UNIT –I
INTRODUCTION

(1) Purpose of Testing:

(i) What we do:

- Testing consumes at least half of the labor expended to produce a working program.
- Few programmers like testing and even fewer like test design—especially if test design and testing take longer than program design and coding.
- This attitude is understandable.
- Software is ephemeral: you can't point to something physical.
- I think, deep down, most of us don't believe in software—at least not the way we believe in hardware.
- If software is insubstantial, then how much more insubstantial does software testing seem? There isn't even some debugged code to point to when we're through with test design.
- The effort put into testing seems wasted if the tests don't reveal bugs.
- There's another, deeper, problem with testing that's related to the reason we do it (MILL78B, MYER79). It's done to catch bugs.
- There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs.
- So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it.
- Therefore, testing and test design amount to an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors.
- Punishment for what? For being human? Guilt for what? For not achieving inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For not being telepathic? For not solving human communication problems that have been kicked around by philosophers and theologians for 40 centuries?
- The statistics show that programming, done well, will still have one to three bugs per hundred statements (AKIY71, ALBE76, BOEH75B, ENDR75, RADA81, SHOO75, THAY76, WEIS85B).*
- Certainly, if you have a 10% error rate, then you either need more programming education or you deserve reprimand *and* guilt.**
- There are some persons who claim that they can write bug-free programs. There's a saying among sailors on the Chesapeake Bay, whose sandy, shifting bottom outdates charts before they're printed, "If you haven't run aground on the Chesapeake, you haven't sailed the Chesapeake much."
- So it is with programming and bugs: I have them, you have them, we all have them—and the point is to do what we can to prevent them and to discover them as early as possible, but not to feel guilty about them.
- Programmers! Cast out your guilt! Spend half your time in joyous testing and debugging! Thrill to the excitement of the chase! Stalk bugs with care, methodology, and reason. Build traps for them.
- Be more artful than those devious bugs and taste the joys of guiltless programming! Testers! Break that software (as you must) and drive it to the ultimate—but don't enjoy the programmer's pain.

(ii) Productivity and quality in Software:

- Consider the manufacture of a mass-produced widget. Whatever the design cost, it is a small part of the total cost when amortized over a large production run.
- Once in production, every manufacturing stage is subjected to quality control and testing from component source inspection to final testing before shipping.
- If flaws are discovered at any stage, the widget or part of it will either be discarded or cycled back for rework and correction.
- The assembly line's productivity is measured by the sum of the costs of the materials, the rework, and the discarded components, and the cost of quality assurance and testing.
- There is a trade-off between quality-assurance costs and manufacturing costs. If insufficient effort is spent in quality assurance, the reject rate will be high and so will the net cost.
- Conversely, if inspection is so good that all faults are caught as they occur, inspection costs will dominate, and again net cost will suffer.
- The manufacturing process designers attempt to establish a level of testing and quality assurance that minimizes net cost for a given quality objective.
- 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 spaceships, nuclear reactors, and aircraft, where failures threaten life.
- The relation between productivity and quality for software is very different from that for manufactured goods.
- The "manufacturing" cost of a software copy is trivial: the cost of the tape or disc and a few minutes of computer time.
- Furthermore, software "manufacturing" quality assurance is automated through the use of check sums and other error-detecting methods.
- Software costs are dominated by development.
- Software maintenance is unlike hardware maintenance. It is not really "maintenance" but an extended development in which enhancements are designed and installed and deficiencies corrected.
- The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
- The main difference then between widget productivity and software productivity is that for hardware quality is only one of several productivity determinants, whereas for software, quality and productivity are almost indistinguishable.

(iii) Goals for testing:

- Testing and test design, as parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs.
- Finally, tests should provide clear diagnoses so that bugs can be easily corrected. Bug prevention is testing's first goal.
- A prevented bug is better than a detected and corrected bug because if the bug is prevented, there's no code to correct.
- Moreover, no retesting is needed to confirm that the correction was valid, no one is embarrassed, no memory is consumed, and prevented bugs can't wreck a schedule.
- More than the act of testing, the act of *designing* tests is one of the best bug preventers known.
- The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded—indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the rest.

- For this reason, Dave Gelperin and Bill Hetzel (GELP87) advocate "Test, then code." The ideal test activity would be so successful at bug prevention that actual testing would be unnecessary because all bugs would have been found and fixed during test design.*
- Unfortunately, we can't achieve this ideal. Despite our effort, there will be bugs because we are human.
- To the extent that testing fails to reach its primary goal, *bug prevention*, it must reach its secondary goal, *bug discovery*. Bugs are not always obvious.
- A bug is manifested in deviations from expected behavior. A test design must document expectations, the test procedure in detail, and the results of the actual test—all of which are subject to error.
- But knowing that a program is incorrect does not imply knowing the bug. Different bugs can have the same manifestations, and one bug can have many symptoms.
- The symptoms and the causes can be disentangled only by using many small detailed tests.

(iv) Phases in a Tester's Mental Life:

(a) Why Testing:

- What's the purpose of testing? There's an attitudinal progression characterized by the following five phases:
  PHASE 0—There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose.
  PHASE 1—The purpose of testing is to show that the software works.
  PHASE 2—The purpose of testing is to show that the software doesn't work.
  PHASE 3—The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value.
  PHASE 4—Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

(b) Phase 0 Thinking:

- I called the inability to distinguish between testing and debugging "phase 0" because it denies that testing matters, which is why I denied it the grace of a number. See Section 2.1 in this chapter for the difference between testing and debugging. If phase 0 thinking dominates an organization, then there can be no effective testing, no quality assurance, and no quality. Phase 0 thinking was the norm in the early days of software development and dominated the scene until the early 1970s, when testing emerged as a discipline.
- Phase 0 thinking was appropriate to an environment characterized by expensive and scarce computing resources, low-cost (relative to hardware) software, lone programmers, small projects, and throwaway software. Today, this kind of thinking is the greatest cultural barrier to good testing and quality software. But phase 0 thinking is a problem for testers and developers today because many software managers learned and practiced programming when this mode was the norm—and it's hard to change how you think.

(c) Phase 1 Thinking-The Software Works

- Phase I thinking represented progress because it recognized the distinction between testing and debugging. This thinking dominated the leading edge of testing until the late 1970s when its fallacy was discovered. This recognition is attributed to Myers (MYER79) who observed that it is self-corrupting. It only takes one failed test to show that software doesn't work, but even an infinite number of tests won't prove that it does. The objective of phase 1 thinking is unachievable. The process is corrupted because the probability of showing that the software works *decreases* as testing increases; that is, the more you test, the likelier you are to find a bug. Therefore, if your objective is to demonstrate a high probability of working, that objective is best achieved by not testing at all! Although this conclusion may seem silly to the

conscious, rational mind, it is the kind of syllogism that our unconscious mind loves to implement.

(d) Phase 2 Thinking-The Software Doesn't Work:

- When, as testers, we shift our goal to phase 2 thinking we are no longer working in cahoots with the designers, but against them. The difference between phase 1 and 2 thinking is illustrated by analogy to the difference between bookkeepers and auditors. The bookkeeper's goal is to show that the books balance, but the auditor's goal is to show that despite the appearance of balance, the bookkeeper has embezzled. Phase 2 thinking leads to strong, revealing tests.
- While one failed test satisfies the phase 2 goal, phase 2 thinking also has limits. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. Phase 2 thinking leads to a never-ending sequence of ever more diabolical tests. Taken to extremes, it too never ends, and the result is reliable software that never gets shipped. The trouble with phase 2 thinking is that we don't know when to stop.

(e) Phase 3 Thinking-Test for Risk Reduction:

- Phase 3 thinking is nothing more than accepting the principles of statistical quality control. I say "accepting" rather than "implementing" because it's not obvious how statistical quality control should be applied to software. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does. Testing, pass or fail, reduces our perception of risk about a software product. The more we test, the more we test with harsh tests, the more confidence we have in the product. We'll risk release when that confidence is high enough.*

(f) Phase 4 Thinking-A State of Mind:

- The phase 4 thinker's knowledge of what testing can and can't do, combined with knowing what makes software testable, results in software that doesn't need much testing to achieve the lower-phase goals. Testability is the goal for two reasons. The first and obvious reason is that we want to reduce the labor of testing. The second and more important reason is that testable code has fewer bugs than code that's hard to test. The impact on productivity of these two factors working together is multiplicative. What makes code testable? One of the main reasons to learn test techniques is to answer that question.

(g) Cumulative Goals:

- The above goals are cumulative. Debugging depends on testing as a tool for probing hypothesized causes of symptoms. There are many ways to break software that have nothing to do with the software's functional requirements: phase 2 tests alone might never show that the software does what it's supposed to do. It's impractical to break software until the easy demonstrations of workability are behind you. Use of statistical methods as a guide to test design, as a means to achieve good testing at acceptable risks, is a way of fine-tuning the process. It should be applied only to large, robust products with few bugs. Finally, a state of mind isn't enough: even the most testable software must be debugged, must work, and must be hard to break.

(v) Test Design:

- Although programmers, testers, and programming managers know that code must be designed and tested, many appear to be unaware that tests themselves must be designed and tested—designed by a process no less rigorous and no less controlled than that used for code.

- Too often, test cases are attempted without prior analysis of the program's requirements or structure. Such test design, if you can call it that, is just a haphazard series of ad-lib cases that are not documented either before or after the tests are executed.
- Because they were not formally designed, they cannot be precisely repeated, and no one is sure whether there was a bug or not. After the bug has been ostensibly corrected, no one is sure that the retest was identical to the test that found the bug.
- Ad-lib tests are useful during debugging, where their primary purpose is to help locate the bug, but adlib tests done in support of debugging, no matter how exhausting, are not substitutes for *designed* tests.
- The test-design phase of programming should be explicitly identified. Instead of "design, code, desk check, test, and debug," the programming process should be described as: "design, test design, code, test code, program inspection, test inspection, test debugging, test execution, program debugging, testing."
- Giving test design an explicit place in the scheme of things provides more visibility to that amorphous half of the labor that often goes under the name "test and debug." It makes it less likely that test design will be given short shrift when the budget's small and the schedule's tight and there's a vague hope that maybe this time, just this once, the system will come together without bugs.

(vi) Testing Isn't Everything:

- This is a book on testing techniques, which are only *part* of our weaponry against bugs. Research and practice (BASI87, FAGA76, MYER78, WEIN65, WHIT87) show that other approaches to the creation of good software are possible and essential. Testing, I believe, is still our most potent weapon, but there's evidence (FAGA76) that other methods may be as effective: but you can't implement inspections, say, *instead* of testing because testing and inspections catch or prevent different kinds of bugs. Today, if we want to prevent all the bugs that we can and catch those that we don't prevent, we must review, inspect, read, do walkthroughs, *and then test*. We don't know today the mix of approaches to use under what circumstances. Experience shows that the "best mix" *very much* depends on things such as development environment, application, size of project, language, history, and culture. The other major methods in decreasing order of effectiveness are as follows:
- **Inspection Methods**—In this category I include walkthroughs, desk checking, formal inspections (FAGA76), and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overlap.
- **Design Style**—By this term I mean the stylistic criteria used by programmers to define what they mean by a "good" program. Sticking to outmoded style, such as "tight" code or "optimizing" for performance destroys quality. Conversely, adopting stylistic objectives such as testability, openness, and clarity can do much to prevent bugs.
- **Static Analysis Methods**—These methods include anything that can be done by formal analysis of source code during or in conjunction with compilation. Syntax checking in early compilers was rudimentary and was part of the programmer's "testing," Compilers have taken that job over (thank the Lord). **Strong typing** and **type checking** eliminate an entire category of bugs. There's a lot more that can be done to detect errors by static analysis. It's an area of intensive research and development. For example, much of **data-flow anomaly** detection (see Chapters 5 and 8), which today is part of testing, will eventually be incorporated into the compiler's static analysis.
- **Languages**—The source language can help reduce certain kinds of bugs. Languages continue to evolve, and preventing bugs is a driving force in that evolution. Curiously, though, programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the language used.

- **Design Methodologies and Development Environment**—The design methodology (that is, the development process used and the environment in which that methodology is embedded), can prevent many kinds of bugs. For example, configuration control and automatic distribution of change information prevents bugs which result from a programmer's unawareness that there were changes.

(vii) The Pesticide Paradox and the Complexity Barrier:

- You're a poor farmer growing cotton in Alabama and the boll weevils are destroying your crop. You mortgage the farm to buy DDT, which you spray on your field, killing 98% of the pest, saving the crop. The next year, you spray the DDT early in the season, but the boll weevils still eat your crop because the 2% you didn't kill last year were resistant to DDT. You now have to mortgage the farm to buy DDT *and* Malathion; then next year's boll weevils will resist both pesticides and you'll have to mortgage the farm yet again. That's the pesticide paradox* for boll weevils and also for software testing.
- *First Law: The Pesticide Paradox*—Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. That's not too bad, you say, because at least the software gets better and better. Not quite!
- *Second Law: The Complexity Barrier*—Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity.
- By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but this time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

(2) Some Dichotomies:

(i) Testing Versus Debugging:

- Testing and debugging are often lumped under the same heading, and it's no wonder that their roles are often confused: for some, the two words are synonymous; for others, the phrase "test and debug" is treated as a single word. The **purpose of testing** is to show that a program has bugs. **The purpose of debugging** is 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:

  1. Testing starts with known conditions, uses predefined procedures, and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts from possibly unknown initial conditions, and the end cannot be predicted, except statistically.

  2. Testing can and should be planned, designed, and scheduled. The procedures for, and duration of, debugging cannot be so constrained.

  3. Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.

  4. Testing proves a programmer's failure. Debugging is the programmer's vindication.

  5. Testing, as executed, should strive to be predictable, dull, constrained, rigid, and inhuman. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.

  6. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.

  7. Testing can often be done by an outsider. Debugging must be done by an insider.

8. Although there is a robust theory of testing that establishes theoretical limits to what testing can and can't do, debugging has only recently been attacked by theorists—and so far there are only rudimentary results.

9. Much of test execution and design can be automated. Automated debugging is still a dream.

(ii) 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. The software's user should be concerned only with functionality and features, and the program's implementation details should not matter. Functional testing takes the user's point of view.
- **Structural testing** does look at the implementation details. Such things as programming style, control method, source language, database design, and coding details dominate structural testing; but the boundary between function and structure is fuzzy. Good systems are built in layers—from the outside to the inside. The user sees only the outermost layer, the layer of pure function. Each layer inward is less related to the system's functions and more constrained by its structure: so what is structure to one layer is function to the next. For example, the user of an online system doesn't know that the system has a memory-allocation routine. For the user, such things are structural details. The memory-management routine's designer works from a specification for that routine. The specification is a definition of "function" at that layer. The memory-management routine uses a link-block subroutine. The memory-management routine's designer writes a "functional" specification for a link-block subroutine, thereby defining a further layer of structural detail and function. At deeper levels, the programmer views the operating system as a structural detail, but the operating system's designer treats the computer's hardware logic as the structural detail.
- Most of this book is devoted to models of programs and the tests that can be designed by using those models. A given model, and the associated tests may be first introduced in a structural context but later used again in a functional context, or vice versa. The initial choice of how to present a model was based on the context that seemed most natural for that model and in which it was likeliest that the model would be used for test design. Just as you can't clearly distinguish function from structure, you can't fix the utility of a model to structural tests or functional tests. If it helps you design effective tests, then use the model in whatever context it seems to work.
- There's no controversy between the use of structural versus functional tests: both are useful, both have limitations, both target different kinds of bugs. Functional tests can, in principle, 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. The art of testing, in part, is in how you choose between structural and functional tests.

(iii) The Designer Versus the Tester:

- If testing were wholly based on functional specifications and independent of implementation details, then the designer and the tester could be completely separated. Conversely, to design a test plan based only on a system's structural details would require the software designer's knowledge, and hence only she could design the tests. The more you know about the design, the likelier you are to eliminate useless tests, which, despite functional differences, are actually handled by the same routines over the same paths; but the more you know about the design, the likelier you are to have the same misconceptions as the designer. Ignorance of structure is the independent tester's best friend and worst enemy. The naive tester has no preconceptions about what is or is not possible and will, therefore, design tests that the program's designer would never think of—and many tests that never should be

thought of. Knowledge, which is the designer's strength, brings efficiency to testing but also blindness to missing functions and strange cases. Tests designed and executed by the software's designers are by nature biased toward structural considerations and therefore suffer the limitations of structural testing. Tests designed and executed by an independent tester are bias-free and can't be finished. Part of the artistry of testing is to balance knowledge and its biases against ignorance and its inefficiencies.

- In this book I discuss the "tester," "test-team member," or "test designer" in contrast to the "programmer" and "program designer," as if they were distinct persons. As one goes from **unit testing** to **unit integration**, to **component testing** and integration, to **system testing**, and finally to formal **system feature testing**, it is increasingly more effective if the "tester" and "programmer" are different persons. The techniques presented in this book can be used for all testing—from unit to system. When the technique is used in system testing, the designer and tester are probably different persons; but when the technique is used in unit testing, the tester and programmer merge into one person, who sometimes acts as a programmer and sometimes as a tester.
- You must be a constructive schizophrenic. Be clear about the difference between your role as a programmer and as a tester. The tester in you must be suspicious, uncompromising, hostile, and compulsively obsessed with destroying, utterly destroying, the programmer's software. The tester in you is your Mister Hyde—your Incredible Hulk. He must exercise what Gruenberger calls "low cunning." (HETZ73) The programmer in you is trying to do a job in the simplest and cleanest way possible, on time, and within budget. Sometimes you achieve this by having great insights into the programming problem that reduce complexity and labor and are almost correct. And with that tester/Hulk lurking in the background of your mind, it pays to have a healthy paranoia about bugs. Remember, then, that when I refer to the "test designer" and "programmer" as separate persons, the extent to which they are separated depends on the testing level and the context in which the technique is applied. This saves me the effort of writing about the same technique twice and you the tedium of reading it twice.

(iv) Modularity Versus Efficiency:

- Both tests and systems can be modular. A **module** is a discrete, well-defined, small component of a system. The smaller the component, the easier it is to understand; but every component has interfaces with other components, and *all* interfaces are sources of confusion. The smaller the component, the likelier are interface bugs. Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand. Part of the artistry of software design is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.
- Testing can and should likewise be organized into modular components. Small, independent test cases have the virtue of easy repeatability. If an error is found by testing, only the small test, not a large component that consists of a sequence of hundreds of interdependent tests, need be rerun to confirm that a test design bug has been fixed. Similarly, if the test has a bug, only that test need be changed and not a whole test plan. But microscopic test cases require individual setups and each such setup (e.g., data, inputs) can have bugs. As with system design, artistry comes into test design in setting the scope of each test and groups of tests so that test design, test debugging, and test execution labor are minimized without compromising effectiveness.

(v) Small Versus Large:

- I often write small analytical programs of a few hundred lines that, once used, are discarded. Do I use formal test techniques, quality assurance, and all the rest I so passionately advocate? Of course not, and I'm not a hypocrite. I do what everyone does in similar

circumstances: I design, I code, I test a few cases, debug, redesign, recode, and so on, much as I did 30 years ago. I can get away with such (slovenly) practices because I'm programming for a very small, intelligent, forgiving, user population—me. It's the ultimate of small programs and it is most efficiently done by intuitive means and complete lack of formality.

- Let's up the scale to a larger package. I'm still the only programmer and user, but now, the package has thirty components averaging 750 statements each, developed over a period of 5 years. Now I *must* create and maintain a data dictionary and do thorough unit testing. But I'll take my own word for it and not bother to retain all those test cases or to exercise formal configuration control.
- You can extrapolate from there or draw on your experiences. **Programming in the large** (DERE76) means constructing programs that consist of many components written by many different persons. **Programming in the small** is what we do for ourselves in the privacy of our own offices or as homework exercises in an undergraduate programming course. Size brings with it nonlinear scale effects, which are imperfectly understood today. Qualitative changes occur with size and so must testing methods and quality criteria. A primary example is the notion of **coverage**—a measure of test completeness. Without worrying about exactly what these terms mean, 100% coverage is essential for unit testing, but we back off this requirement as we deal with ever larger software aggregates, accept 75%-85% for most systems, and possibly as low as 50% for huge systems of 10 million lines of code or so.

(vi) The Builder Versus the Buyer:

- Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. Many organizations today recognize the virtue of independent software development and operation because it leads to better software, better security, and better testing. Independent software development does not mean that all software should be bought from software houses or consultants but that the software developing entity and the entity that pays for the software be separated enough to make accountability clear. I've heard of cases where the software development group and the operational group within the same company negotiate and sign formal contracts with one another—with lawyers present. If there is no separation between builder and buyer, there can be no accountability. If there is no accountability, the motivation for software quality disappears and with it any serious attempt to do proper testing.
- Just as programmers and testers can merge and become one, so can builder and buyer. There are several other persons in the software development cast of characters who, like the above, can also be separated or merged:

  1. The **builder**, who designs for and is accountable to
  2. The **buyer**, who pays for the system in the hope of profits from providing services to
  3. The **user**, the ultimate beneficiary or victim of the system. The user's interests are guarded by
  4. The **tester**, who is dedicated to the builder's destruction and
  5. The **operator**, who has to live with the builder's mistakes, the buyer's murky specifications, the tester's oversights, and the user's complaints.

(3) A Model For Testing:

(i) The Project:

- Testing is applied to anything from subroutines to systems that consist of millions of statements. The archetypical system is one that allows the exploration of all aspects of testing without the complications that have nothing to do with testing but affect any very large project.

- It's medium-scale programming. Testing the interfaces between different parts of your own mind is very different from testing the interface between you and other programmers separated from you by geography, language, time, and disposition.
- Testing a one-shot routine that will be run only a few times is very different from testing one that must run for decades and may be modified by some unknown future programmer.
- Although all the problems of the solitary routine occur for the routine that is embedded in a system, the converse is not true: many kinds of bugs just can't exist in solitary routines.
- There is an implied context for the test methods discussed in this book—a real-world context characterized by the following model project:
- **Application**—The specifics of the application are unimportant. It is a real-time system that must provide timely responses to user requests for services. It is an online system connected to remote terminals.
- **Staff**—The programming staff consists of twenty to thirty programmers—big enough to warrant formality, but not too big to manage—big enough to use specialists for some parts of the system's design.
- **Schedule**—The project will take 24 months from the start of design to formal acceptance by the customer. Acceptance will be followed by a 6-month cutover period. Computer resources for development and testing will be almost adequate.
- **Specification**—The specification is good. It is functionally detailed without constraining the design, but there are undocumented "understandings" concerning the requirements.
- **Acceptance Test**—The system will be accepted only after a formal acceptance test. The application is not new, so part of the formal test already exists. At first the customer will intend to design the acceptance test, but later it will become the software design team's responsibility.
- **Personnel**—The staff is professional and experienced in programming and in the application. Half the staff has programmed that computer before and most know the source language. One-third, mostly junior programmers, have no experience with the application. The typical programmer has been employed by the programming department for 3 years. The climate is open and frank. Management's attitude is positive and knowledgeable about the realities of such projects.
- **Standards**—Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards. Documentation is good. There is an internal, semiformal, quality-assurance function. The database is centrally developed and administered.
- **Objectives**—The system is the first of many similar systems that will be implemented in the future. No two will be identical, but they will have 75% of the code in common. Once installed, the system is expected to operate profitably for more than 10 years.
- **Source**—One-third of the code is new, one-third extracted from a previous, reliable, but poorly documented system, and one-third is being rehosted (from another language, computer, operating system—take your pick).
- **History**—One programmer will quit before his components are tested. Another programmer will be fired before testing begins: excellent work, but poorly documented. One component will have to be redone after unit testing: a superb piece of work that defies integration. The customer will insist on five big changes and twenty small ones. There will be at least one nasty problem that nobody—not the customer, not the programmer, not the managers, nor the hardware vendor—suspected. A facility and/or hardware delivery problem will delay testing for several weeks and force second- and third-shift work. Several important milestones will slip but the delivery date will be met.

- Our model project is a typical well-run, successful project with a share of glory and catastrophe—neither a utopian project nor a slice of hell.

(ii)Overview:

- The process starts with a program embedded in an environment, such as a computer, an operating system, or a calling program. We understand human nature and its suceptibility to error. This understanding leads us to create three models: a model of the environment, a model of the program, and a model of the expected bugs. From these models we create a set of tests, which are then executed. The result of each test is either expected or unexpected. If unexpected, it may lead us to revise the test, our model or concept of how the program behaves, our concept of what bugs are possible, or the program itself. Only rarely would we attempt to modify the environment.

(iii)The Environment:

- A **program's environment** is the hardware and software required to make it run. For online systems the environment may include communications lines, other systems, terminals, and operators. The environment also includes all programs that interact with—and are used to create—the program under test, such as operating system, loader, linkage editor, compiler, utility routines.
- Programmers should learn early in their careers that it's not smart to blame the environment (that is, hardware and firmware) for bugs. Hardware bugs are rare. So are bugs in manufacturer-supplied software. This isn't because logic designers and operating system programmers are better than application programmers, but because such hardware and software is stable, tends to be in operation for a long time, and most bugs will have been found and fixed by the time programmers use that hardware or software.* Because hardware and firmware are stable, we don't have to consider all of the environment's complexity. Instead, we work with a simplification of it, in which only the features most important to the program at hand are considered. Our model of the environment includes our *beliefs* regarding such things as the workings of the computer's instruction set, operating system macros and commands, and what a higher-order language statement will do. If testing reveals an unexpected result, we may have to change our beliefs (our model of the environment) to find out what went wrong. But sometimes the environment could be wrong: the bug could be in the hardware or firmware after all.

(iv)The Program:

- Most programs are too complicated to understand in detail. We must simplify our concept of the program in order to test it. So although a real program is exercised on the test bed, in our brains we deal with a simplified version of it—a version in which most details are ignored. If the program calls a subroutine, we tend not to think about the subroutine's details unless its operation is suspect. Similarly, we may ignore processing details to focus on the program's control structure or ignore control structure to focus on processing. As with the environment, if the simple model of the program does not explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

(v)Bugs:

- Bugs are more insidious than ever we expect them to be. Yet it is convenient to categorize them: initialization, call sequence, wrong variable, and so on. Our notion of what is or isn't a bug varies. A bad specification may lead us to mistake good behavior for bugs, and vice versa. An unexpected test result may lead us to change our notion of what a bug is—that is to say, our model of bugs.
- While we're on the subject of bugs, I'd like to dispel some optimistic notions that many programmers and testers have about bugs. Most programmers and testers have beliefs

about bugs that express a naivete that ranks with belief in the tooth fairy. If you hold any of the following beliefs, then disabuse yourself of them because as long as you believe in such things you will be unable to test effectively and unable to justify the dirty tests most programs need.

- **Benign Bug Hypothesis**—The belief that bugs are nice, tame, and logical. Only weak bugs have a logic to them and are amenable to exposure by strictly logical means. Subtle bugs have no definable pattern—they are wild cards.
- **Bug Locality Hypothesis**—The belief that a bug discovered within a component affects only that component's behavior; that because of structure, language syntax, and data organization, the symptoms of a bug are localized to the component's designed domain. Only weak bugs are so localized. Subtle bugs have consequences that are arbitrarily far removed from the cause in time and/or space from the component in which they exist.
- **Control Bug Dominance**—The belief that errors in the control structure of programs dominate the bugs. While many easy bugs, especially in components, can be traced to **control-flow** errors, **data-flow** and data-structure errors are as common. Subtle bugs that violate data-structure boundaries and data/code separation can't be found by looking only at control structures.
- **Code/Data Separation**—The belief, especially in HOL programming, that bugs respect the separation of code and data.* Furthermore, in real systems the distinction between code and data can be hard to make, and it is exactly that blurred distinction that permit such bugs to exist.
- **Lingua Salvator Est**—The hopeful belief that language syntax and semantics (e.g., structured coding, strong typing, complexity hiding) eliminates most bugs. True, good language features do help prevent the simpler component bugs but there's no statistical evidence to support the notion that such features help with subtle bugs in big systems.
- **Corrections Abide**—The mistaken belief that a corrected bug remains corrected. Here's a generic counterexample. A bug is believed to have symptoms caused by the interaction of components A and B but the real problem is a bug in C, which left a residue in a data structure used by both A and B. The bug is "corrected" by changing A and B. Later, C is modified or removed and the symptoms of A and B recur. Subtle bugs are like that.
- **Silver Bullets**—The mistaken belief that X (language, design method, representation, environment—name your own) grants immunity from bugs. Easy-to-moderate bugs may be reduced, but remember the pesticide paradox.
- **Sadism Suffices**—The common belief, especially by independent testers, that a sadistic streak, low cunning, and intuition are sufficient to extirpate most bugs. You only catch easy bugs that way. Tough bugs need methodology and techniques, so read on.
- **Angelic Testers**—The ludicrous belief that testers are better at test design than programmers are at code design.*

(vi)Tests:

- Tests are formal procedures. Inputs must be prepared, outcomes predicted, tests documented, commands executed, and results observed; all these steps are subject to error. There is nothing magical about testing and test design that immunizes testers against bugs. An unexpected test result is as often cause by a test bug as it is by a real bug.* Bugs can creep into the documentation, the inputs, and the commands and becloud our observation of results. An unexpected test result, therefore, may lead us to revise the tests. Because the tests are themselves in an environment, we also have a mental model of the tests, and instead of revising the tests, we may have to revise that mental model.

(vii)Testing and Levels:

- We do three distinct kinds of testing on a typical software system: **unit/ component testing, integration testing**, and **system testing**. The objectives of each class is different and therefore, we can expect the mix of test methods used to differ. They are:
- **Unit, Unit Testing**—A **unit** is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a **test harness** or **driver**. A **unit** is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. **Unit testing** is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a **unit bug**.
- **Component, Component Testing**—A **component** is an **integrated aggregate** of one or more units. A unit is a component, a component with subroutines it calls is a component, etc. By this (recursive) definition, a component can be anything from a unit to an entire system. **Component testing** is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure.
- When our tests reveal such problems, we say that there is a **component bug**. **Integration, Integration Testing**—**Integration** is a *process* by which components are aggregated to create larger components. **Integration testing** is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B have both passed their component tests.
- Integration testing is aimed as showing inconsistencies between A and B. Examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated objects, which is just higher level component testing. Integration testing is specifically aimed at exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B, and finally, component testing for the "new" component (A,B).*
- **System, System Testing**—A **system** is a big component. **System testing** is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.
- This book concerns component testing, but the techniques discussed here also apply to integration and system testing. There aren't any special integration and system testing techniques but the mix of effective techniques changes as our concern shifts from components to integration, to system. How and where integration and system testing will be covered is discussed in the preface to this book. You'll find comments on techniques concerning their relative effectiveness as applied to component, integration, and system testing throughout the book. Such comments are intended to guide your selection of a mix of techniques that best matches your testing concerns, be it component, integration, or system, or some mixture of the three.

(viii)The Role of Models:

- Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. Each model is used either until we accept the behavior as correct or until the model is no longer sufficient for the purpose. Unexpected test results always force a revision of some mental model, and in turn may lead to a revision of

whatever is being modeled. The revised model may be more detailed, which is to say more complicated, or more abstract, which is to say simpler. The art of testing consists of creating, selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behavior.

(4) The Consequences of Bugs:

(i) The Importance of Bugs:

- The importance of a bug depends on frequency, correction cost, installation cost, and consequences.
- **Frequency**—How often does that kind of bug occur? See Table 2.1 on page 57 for bug frequency statistics. Pay more attention to the more frequent bug types.
- **Correction Cost**—What does it cost to correct the bug after it's been found? That cost is the sum of two factors: (1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size. The larger the system the more it costs to correct the same bug.
- **Installation Cost**—Installation cost depends on the number of installations: small for a single-user program, but how about a PC operating system bug? Installation cost can dominate all other costs—fixing one simple bug and distributing the fix could exceed the entire system's development cost.
- **Consequences**—What are the consequences of the bug? You might measure this by the mean size of the awards made by juries to the victims of your bug.
- A reasonable metric for bug importance is:
  importance($) = frequency*(correction_cost + installation_cost + consequential_cost)
- Frequency tends not to depend on application or environment, but correction, installation, and consequential costs do. As designers, testers, and QA workers, you must be interested in bug importance, not raw frequency. Therefore you must create your own importance model. This chapter will help you do that.

(ii) How Bugs Affect Us-Consequences:

- Bug consequences range from mild to catastrophic. Consequences should be measured in human rather than machine terms because it is ultimately for humans that we write programs. If you answer the question, "What are the consequences of this bug?" in machine terms by saying, for example, "Bit so-and-so will be set instead of reset," you're avoiding responsibility for the bug. Although it may be difficult to do in the scope of a subroutine, programmers should try to measure the consequences of their bugs in human terms. Here are some consequences on a scale of one to ten:
- **1.** *Mild*—The symptoms of the bug offend us aesthetically; 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. Names are truncated or arbitrarily modified. Bills for $0.00 are sent. Operators must use unnatural command sequences and must trick the system into a proper response for unusual bug-related cases.
- **4.** *Disturbing*—It refuses to handle legitimate transactions. The automatic teller machine won't give you money. My credit card is declared invalid.
- **5.** *Serious*—It loses track of transactions: not just the transaction itself (your paycheck), but the fact that the transaction occurred. Accountability is lost.
- **6.** *Very Serious*—Instead of losing your paycheck, the system credits it to another account or converts deposits into withdrawals. The bug causes the system to do the wrong transaction.

- **7.** *Extreme*—The problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
- **8.** *Intolerable*—Long-term, unrecoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
- **9.** *Catastrophic*—The decision to shut down is taken out of our hands because the system fails.
- **10.** *Infectious*—What can be worse than a failed system? One that corrupts other systems even though it does not fail in itself; that erodes the social or physical environment; that melts nuclear reactors or starts wars; whose influence, because of malfunction, is far greater than expected; a system that kills.
- Any of these consequences could follow from that wrong bit. Programming is a serious business, and testing is more serious still. It pays to have nightmares about undiscovered bugs once in a while (SHED80). When was the last time one of your bugs violated someone's human rights?

(iii) Flexible Severity Rather Than Absolutes:

- Many programmers, testers, and quality assurance workers have an absolutist attitude toward bugs. "Everybody *knows* that a program must be *perfect* if it's to work: if there's a bug, it *must* be fixed." That's untrue, of course, even though the myth continues to be foisted onto an unwary public. Ask the person in the street and chances are that they'll parrot that myth of ours. That's trouble for us because we can't do it now and never could. It's *our* myth because we, the computer types, created it and continue to perpetuate it. Software never was perfect and won't get perfect. But is that a license to create garbage? The missing ingredient is our reluctance to quantify quality. If instead of saying that software has either 0 quality (there is at least one bug) or 100% (perfect quality and no bugs), we recognize that quality can be measured on some scale, say from 0 to 10. Quality can be measured as a combination of factors, of which the number of bugs and their severity is only one component. The details of how this is done is the subject of another book; but it's enough to say that many organizations have designed and use satisfactory, quantitative, quality metrics. Because bugs and their symptoms play a significant role in such metrics, as testing progresses you can see the quality rise from next to zero to some value at which it is deemed safe to ship the product.
- Examining these metrics closer, we see that how the parts are weighted depends on environment, application, culture, and many other factors.
- Let's look at a few of these:
- **Correction Cost**—The cost of correcting a bug has almost nothing to do with symptom severity. Catastrophic, life-threatening bugs could be trivial to fix, whereas minor annoyances could require major rewrites to correct.
- **Context and Application Dependency**—The severity of a bug, for the same bug with the same symptoms, depends on context. For example, a roundoff error in an orbit calculation doesn't mean much in a spaceship video game but it matters to real astronauts.
- **Creating Culture Dependency**—What's important depends on the creators of the software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their products than, say, games software vendors.
- **User Culture Dependency**—What's important depends on the user culture. An R&D shop might accept a bug for which there's a workaround; a banker would go to jail for that same bug; and naive users of PC software go crazy over bugs that pros ignore.
- **The Software Development Phase**—Severity depends on development phase. Any bug gets more severe as it gets closer to field use and more severe the longer it's been around—

more severe because of the dramatic rise in correction cost with time. Also, what's a trivial or subtle bug to the designer means little to the maintenance programmer for whom all bugs are equally mysterious.

(iv) The Nightmare List and When to Stop Testing:

- In George Orwell's novel, *1984*, there's a torture chamber called "room 101"—a room that contains your own special nightmare. For me, sailing through 4-foot waves, the boat heeled over, is exhilarating; for my seasick passengers, that's room 101. For me, rounding Cape Horn in winter, with 20-foot waves in a gale is a room 101 but I've heard round-the-world sailboat racers call such conditions "bracing."
- The point about bugs is that you or your organization must define your own nightmares. I can't tell you what they are, and therefore I can't ascribe a severity to bugs. Which is why I treat all bugs as equally as I can in this book. And when I slip and express a value judgment about bugs, recognize it for what it is because I can't completely rid myself of my own values.
- How should you go about quantifying the nightmare? Here's a workable procedure:
- **1.** List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms. For end users and the population at large, the categories of Section 2.2 above are a starting point. For programmers the nightmare may be closer to home, such as: "I might get a bad personal performance rating."
- **2.** Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare, but if your scope extends to the public, it could be the cost of lawsuits, lost business, or nuclear reactor meltdowns.
- **3.** Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
- **4.** Based on your experience, measured data (the best source to use), intuition, and published statistics postulate the kinds of bugs that are likely to create the symptoms expressed by each nightmare. Don't go too deep because most bugs are easy. This is a bug design process. If you can "design" the bug by a one-character or one statement change, then it's a good target. If it takes hours of sneaky thinking to characterize the bug, then either it's an unlikely bug or you're worried about a saboteur in your organization, which could be appropriate in some cases. Most bugs are simple goofs once you find and understand them.
- **5.** For each nightmare, then, you've developed a list of possible causative bugs. Order that list by decreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across all nightmares:
- **6.** Rank the bug types in order of decreasing importance to you.
- **7.** Design tests (based on your knowledge of test techniques) and design your quality assurance inspection process by using the methods that are most effective against the most important bugs.
- **8.** If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmare is possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities. As you test, revise the probabilities and reorder the nightmare list. Taking whatever information you get from testing and working it back through the exercise leads you to revise your subsequent test strategy, either on this project if it's big enough or long enough, or on subsequent projects.
- **9.** Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

- The above prescription can be implemented as a formal part of the software development process, or it can be adopted as a guideline or philosophical point of view. The idea is not that you implement elaborate metrics (unless that's appropriate) but that you recognize the importance of the feedback that testing provides to the testing process itself and, more important, to the kinds of tests you will design.
- The mature tester's problem has never been how to design tests. If you understand testing techniques, you will know how to design several different infinities of justifiable tests. The tester's central problem is how to best cull a reasonable, finite, number of tests from that multifold infinity—a test suite that, as experience and logic leads us to predict, will have a high probability of putting the nightmares to rest—that is to say, an effective, revealing, set of tests. Look at the pesticide paradox again and observe the following consequence:
- Corollary to the First Law—Test suites wear out.
- Yesterday's elegant, revealing, effective, test suite will wear out because programmers and designers, given feedback on their bugs, do modify their programming habits and style in an attempt to reduce the incidence of bugs they know about. Furthermore, the better the feedback, the better the QA, the more responsive the programmers are, the faster those suites wear out. Yes, the software is getting better, but that only allows you to approach closer to, or to leap over, the previous complexity barrier. True, bug statistics tell you nothing about the coming release, only the bugs of the previous release—but that's better than basing your test technique strategy on general industry statistics or on myths. If you don't gather bug statistics, organized into some rational taxonomy, you don't know how effective your testing has been, and worse, you don't know how worn out your test suite is. The consequences of that ignorance is a brutal shock.
- How many horror stories do you want to hear about the sophisticated outfit that tested long, hard, and diligently—sent release 3.4 to the field, confident that it was the best tested product they had ever shipped—only to have it bomb more miserably than any prior release?

(5) A Taxonomy For Bugs:

(i) General:

- There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer's state of mind. For example, a one-character error in a source statement changes the statement, but unfortunately it passes syntax checking. As a result, data are corrupted in an area far removed from the actual bug. That in turn leads to an improperly executed function. Is this a typewriting error, a coding error, a data error, or a functional error? If the bug is in our own program, we're tempted to blame it on typewriting;** if in another programmer's code, on carelessness. And if our job is to critique the system, we might say that the fault is an inadequate internal data-validation mechanism. A detailed taxonomy is presented in the appendix.
- The major categories are: requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing. A first breakdown is provided in Table 2. 1, whereas in the appendix the breakdown is as fine as makes sense. Bug taxonomy, as testing, is potentially infinite. More important than adopting the "right" taxonomy is that you adopt *some* taxonomy and that you use it as a statistical framework on which to base your testing strategy. Because there's so much effort required to develop a taxonomy, don't redo my work—you're invited to adopt the taxonomy of the appendix (or any part thereof) and are hereby authorized to copy it (with appropriate attribution) without guilt or fear of being sued by me for plagiarism. If my taxonomy doesn't turn you on, adopt the IEEE taxonomy (IEEE87B).

(ii) Requirements, Features, and Functionality Bugs:

(a) Requirements and Specifications:

- Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specification may assume, but not mention, other specifications and prerequisites that are known to the specifier but not to the designer. And specifications that don't have these flaws may change while the design is in progress. Features are modified, added, and deleted. The designer has to hit a moving target and occasionally misses.
- Requirements, especially as expressed in a specification (or often, as *not* expressed because there is no specification) are a major source of expensive bugs. The range is from a few percent to more than 50%, depending on application and environment. What hurts most about these bugs is that they're the earliest to invade the system and the last to leave. It's not unusual for a faulty requirement to get through all development testing, beta testing, and initial field use, only to be caught after hundreds of sites have been installed.

(b) Feature Bugs:

- Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. A missing feature or case is the easiest to detect and correct. A wrong feature could have deep design implications. Extra features were once considered desirable. We now recognize that "free" features are rarely free. Any increase in generality that does not contribute to reliability, modularity, maintainability, and robustness should be suspected. Gratuitous enhancements can, if they increase complexity, accumulate into a fertile compost heap that breeds future bugs, and they can create holes that can be converted into security breaches. Conversely, one cannot rigidly forbid additional features that might be a consequence of good design. Removing the features might complicate the software, consume more resources, and foster more bugs.

(c) Feature Interaction:

- Providing clear, correct, implementable, and testable feature specifications is not enough. Features usually come in groups of related features. The features of each group and the interaction of features within each 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. Call holding allows you to put a new incoming call on hold while you continue talking to the first caller. Call forwarding allows you to redirect incoming calls to some other telephone number. Here are some simple feature interaction questions: How about holding a third call when there is already a call on hold? Forwarding forwarded calls (i.e., the number forwarded to is also forwarding calls)? Forwarding calls in a loop? Holding while forwarding is active? Initiating forwarding when there is a call on hold? Holding for forwarded calls when the telephone forwarded to does (doesn't) have forwarding? . . . If you think these variations are brain twisters, how about feature interactions for your income tax return, say between federal, state, and local tax laws? Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore feature interaction bugs. We have very little statistics on these bugs, but the trend seems to be that as the earlier, simpler, bugs are removed, feature interaction bugs emerge as a major category. Other than deliberately preventing some interactions and testing the important combinations, we have no magic remedy for these problems.

(d) Specification and Feature Bug Remedies:

- Most feature bugs are rooted in human-to-human communication problems. One solution is the use of high-level, formal specification languages or systems (BELF76, BERZ85,

DAVI88A, DAV18813, FISC79, HAYE85, PROG88, SOFT88, YEHR80). Such languages and systems provide short-term support but, in the long run, do not solve the problem.

- **Short-Term Support**—Specification languages (we'll call them all "languages" hereafter, even though some may be interactive dialogue systems) facilitate formalization of requirements and (partial)* inconsistency and ambiguity analysis. With formal specifications, partially to fully automatic test case generation is possible. Generally, users and developers of such products have found them to be cost-effective.
- **Long-Term Support**—Assume that we have a great specification language and that it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria. A specification written in that language could theoretically be compiled into object code (ignoring efficiency and practicality issues). But this is just programming in HOL squared. The specification problem has been shifted to a higher level but not eliminated. Theoretical considerations aside, given a system which can generate functional tests from specifications, the likeliest impact is a further complexity escalation facilitated by the reduction of another class of bugs (the complexity barrier law).
- The long-term impact of formal specification languages and systems will probably be that they will influence the design of ordinary programming languages so that more of *current* specification can be formalized. This approach will reduce, but not eliminate, specification bugs. The pesticide paradox will work again to eliminate the kinds of specification bugs we now have (simple ambiguities and contradictions), leaving us a residue of tougher specification bugs that will need an even higher order specification system to expose.

(e) Testing Techniques:

- Most **functional test techniques**—that is, those techniques which are based on a behavioral description of software, such as **transaction flow testing** (Chapter 4), **syntax testing** (Chapter 9), **domain testing** (Chapter 6), **logic testing** (Chapter 10), and **state testing** (Chapter 11) are useful in testing functional bugs. They are also useful in testing for requirements and specification bugs to the extent that the requirements can be expressed in terms of the model on which the technique is based.

(iii) Structural Bugs:

(a) 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 switches, **spaghetti code**, and worst of all, **pachinko code**.
- Although much of testing and software design literature focuses on control flow bugs, they are not as common in new software as the literature might lead one to believe. One reason for the popularity of control-flow problems in the literature is that this area is amenable to theoretical treatment. Fortunately, most control-flow bugs (in new code) are easily tested and caught in unit testing.
- Another source of confusion and therefore research concern is that novice programmers working on toy problems do tend to have more control-flow bugs than experienced programmers. A third reason for concern with control-flow problems is that dirty old code, especially assembly language and COBOL code, can be dominated by control-flow bugs. In fact, a good reason to rewrite an application from scratch is that the old control structure has become so complicated and so arbitrary after decades of rework that no one dare modify it further and, further, it defies testing.
- Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically, path testing (Chapter 3), combined with a bottom-line functional test based on a specification. These bugs are partially prevented by language choice (e.g., languages

that restrict control-flow options) and style, and most important, lots of memory. Experience shows that many control-flow problems result directly from trying to "squeeze" 8 pounds of software into a 4-pound bag (i.e., 8K object into 4K). Squeezing for short execution time is as bad.

(b) Logic Bugs:

- Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations, include nonexistent cases, improper layout of cases, "impossible" cases that are not impossible, a "don't-care" case that matters, improper negation of a boolean expression (for example, using "greater than" as the negation of "less than"), improper simplification and combination of cases, overlap of exclusive cases, confusing "exclusive OR" with "inclusive OR."
- Another problematic area concerns misunderstanding the semantics of the order in which a boolean expression is evaluated for specific compilers, especially in the context of deeply nested IF-THEN-ELSE constructs. For example, the truth or falsity of a logical expression is determined after evaluating a few terms, so evaluation of further terms (usually) stops, but the programmer expects that further terms will be evaluated. In other words, although the boolean expression appears as a single statement, the programmer does not understand that its components will be evaluated sequentially. See index entries on **predicate coverage** for more information.
- If these bugs are part of logical (i.e., boolean) processing not related to control flow, then they are categorized as processing bugs. If they are part of a logical expression (i.e., **control-flow predicate**) which is used to direct the control flow, then they are categorized as control-flow bugs.
- Logic bugs are not really different in kind from arithmetic bugs. They are likelier than arithmetic bugs because programmers, like most people, have less formal training in logic at an early age than they do in arithmetic. The best defense against this kind of bug is a systematic analysis of cases. Logic-based testing (Chapter 10) is helpful.

(c) Processing Bugs:

- Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection, and general processing. Many problems in this area are related to incorrect conversion from one data representation to another. This is especially true in assembly language programming. Other problems include ignoring overflow, ignoring the difference between positive and negative zero, improper use of greater-than, greater-than-or-equal, less-than, less-than-or-equal, assumption of equality to zero in floating point, and improper comparison between different formats as in ASCII to binary or integer to floating point.
- Although these bugs are frequent (12%), they tend to be caught in good unit testing and also tend to have localized effects. Selection of covering test cases, especially domain-testing methods (Chapter 6) are the testing remedies for this kind of bug.

(d) Initialization Bugs:

- Initialization bugs are common, and experienced programmers and testers know they must look for them. Both improper and superfluous initialization occur. The latter tends to be less harmful but can affect performance. Typical bugs are as follows: forgetting to initialize working space, registers, or data areas before first use or assuming that they are initialized elsewhere; a bug in the first value of a loop-control parameter; accepting an initial value without a validation check; and initializing to the wrong format, data representation, or type.
- The remedies here are in the kinds of tools the programmer has. The source language also helps. Explicit declaration of all variables, as in Pascal, helps to reduce some initialization problems. Preprocessors, either built into the language or run separately, can detect some,

but not all, initialization problems. The test methods of Chapter 5 are helpful for test design and for debugging initialization problems.

(e) Data Flow Bugs and Anomalies:

- Most initialization bugs are a special case of data-flow anomalies. A **data-flow anomaly** occurs when 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 data and then not storing or using the result, or initializing twice without an intermediate use. Although part of data-flow anomaly detection can be done by the compiler based on information known at compile time, much can be detected only by execution and therefore is a subject for testing. It is generally recognized today that data-flow anomalies are as important as control-flow anomalies. The methods of Chapters 5 and 12 will help you design tests aimed at data-flow problems.

(iv) Data Bugs:

(a) General:

- 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. Underestimating the frequency of data bugs is caused by poor bug accounting. In some projects, bugs in data declarations are just not counted, and for that matter, data declaration statements are not counted as part of the code. The separation of code and data is, of course, artificial because their roles can be interchanged at will. At the extreme, one can write a twenty-instruction program that can simulate any computer (a Turing machine) and have all "programs" recorded as data and manipulated as such. Furthermore, this can be done in any language on any computer—but who would want to?
- Software is evolving toward programs in which more and more of the control and processing functions are stored in tables. I call this the third law:
- *Third Law*—Code migrates to data.
- Because of this law there is an increasing awareness that bugs in code are only half the battle and that data problems should be given equal attention. The bug statistics of Table 2.1 support this concept; that is, structural bugs and data bugs each have frequencies of about 25%. If you examine a piece of contemporary source code, you may find that half of the statements are data declarations. Although these statements do not result in executable code, because they are specified by humans, they are as subject to error as operative statements. If a program is designed under the assumption that a certain data object will be set to zero and it isn't, the operative statements of the program are not at fault. Even so, there is still an initialization bug, which, because it is in a data statement, could be harder to find than if it had been a bug in executable code.
- This increase in the proportion of the source statements devoted to data definition is a direct consequence of two factors: (1) the dramatic reduction in the cost of main memory and disc storage, and (2) the high cost of creating and testing software. Generalized software controlled by tables is not efficient. Computer costs, especially memory costs, have decreased to the point where the inefficiencies of generalized table-driven code are not usually significant. The increasing cost of software as a percentage of system cost has shifted the emphasis in the software industry away from single-purpose, unique software to an increased reliance on prepackaged, generalized programs. This trend is evident in the computer manufacturers' software, in the existence of a healthy proprietary software industry, and in the emergence of languages and programming environments that support code reusability (e.g., object-oriented languages). Generalized packages must satisfy a wide range of options, host configurations, operating systems, and computers. The designer of a

generalized package achieves generality, in part, by making many things parametric, such as array sizes, memory partition, and file structure. It is not unusual for a big application package to have several hundred parameters. Setting the parameter values particularizes the program to the specific installation. The parameters are interrelated, and errors in those relations can cause illogical conditions and, therefore, bugs.

- Another source of database complexity increase is the use of control tables in lieu of code. The simplest example is the use of tables that turn processing options on and off. A more complicated form of control table is used when a system must execute a set of closely related processes that have the same control structure but are different in details. An early example is found in telephony, where the details of controlling a telephone call are table-driven. A generalized call-control processor handles calls from and to different kinds of lines. The system is loaded with a set of tables that corresponds to the protocols required for that telephone exchange. Another example is the use of generalized device-control software which is particularized by data stored in device tables. The operating system can be used with new, undefined devices, if those devices' parameters can fit into a set of very broad values. The culmination of this trend is the use of complete, internal, transaction-control languages designed for the application. Instead of being coded as computer instructions or language statements, the steps required to process a transaction are stored as a sequence of constants in a transaction-processing table. The state of the transaction, that is, the current processing step, is stored in a transaction-control block. The generalized transaction-control processor uses the combination of transaction state and the control tables to direct the transaction to the next step. The transaction-control table is actually a program which is processed interpretively by the transaction-control processor. That program may contain the equivalent of addressing, conditional branch instructions, looping statements, case statements, and so on. In other words, a **hidden programming language** has been created. It is an effective design technique because it enables fixed software to handle many different transaction types, individually and simultaneously. Furthermore, modifying the control tables to install new transaction types is usually easier than making the same modifications in code.
- In summary, current programming trends are leading to the increasing use of undeclared, internal, specialized programming languages. These are languages—make no mistake about that—even if they are simple compared to normal programming languages; but the syntax of these languages is rarely debugged. There's no compiler for them and therefore no source syntax checking. The programs in these languages are inserted as octal or hexadecimal codes—as if we were programming back in the early days of UNIVAC-I. Large, low-cost memory will continue to strengthen this trend and, consequently, there will be an increased incidence of code masquerading as data. Bugs in this kind of hidden code are at least as difficult to find as bugs in normal code. The first step in the avoidance of data bugs—whether the data are used as pure data, as parameters, or as hidden code—is the realization that *all* source statements, including data declarations, must be counted, and that all source statements, whether or not they result in object code, are bug-prone.
- The categories used for data bugs are different from those used for code bugs. Each way of looking at data provides a different perspective. These categories for data bugs overlap and are no stricter than the categories used for bugs in code.

(b) Dynamic Versus Static:

- Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, 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. Failure to initialize a shared object properly can lead to data-dependent bugs caused by residues from a previous use of that object by another transaction. Note that the culprit transaction is long

gone when the bug's symptoms are discovered. Because the effect of corruption of dynamic data can be arbitrarily far removed from the cause, such bugs are among the most difficult to catch. The design remedy is complete documentation of all shared-memory structures, defensive code that does thorough data-validation checks, and centralized-resource managers.

- The basic problem is leftover garbage in a shared resource. This can be handled in one of three ways: (1) cleanup after use by the user, (2) common cleanup by the resource manager, and (3) no cleanup. The latter is the method usually used. Therefore, resource users must program under the assumption that the resource manager gives them garbage-filled resources. Common cleanup is used in very secure systems where subsequent users of a resource must never be able to read data left by a previous user in another security or privacy category.
- Static data are fixed in form and content. Whatever their purpose, they appear in the source code or data base, directly or indirectly, as, for example, a number, a string of characters, or a bit pattern. Static data need not be explicit in the source code. Some languages provide **compile-time processing**, which is especially useful in general-purpose routines that are particularized by interrelated parameters. Compile-time processing is an effective measure against parameter-value conflicts. Instead of relying on the programmer to calculate the correct values of interrelated parameters, a program executed at compile time (or assembly time) calculates the parameters' values. If compile-time processing is not a language feature, then a specialized preprocessor can be built that will check the parameter values and calculate those values that are derived from others. As an example, a large commercial telecommunications system has several hundred parameters that dictate the number of lines, the layout of all storage media, the hardware configuration, the characteristics of the lines, the allowable user options for those lines, and so on. These are processed by a site-adapter program that not only sets the parameter values but builds data declarations, sizes arrays, creates constants, and inserts processing routines from a library. A bug in the site adapter, or in the data given to the site adapter, can result in bugs in the static data used by the object programs for that site.
- Another example is the postprocessor used to install many personal computer software packages. Here the configuration peculiarities are handled by generalized table-driven software, which is particularized at run (actually, installation) time.
- Any preprocessing (or postprocessing) code, any code executed at compile or assembly time or before, at load time, at installation time, or some other time can lead to faulty static data and therefore bugs—even though such code (and the execution thereof) does not represent object code at run time. We tend to take compilers, assemblers, utilities, loaders, and configurators for granted and do not suspect them to be bug sources. This is not a bad assumption for standard utilities or translators. But if a highly parameterized system uses site-adapter software or preprocessors or compile-time/assembly-time processing, and if such processors and code are developed concurrently with the working software of the application—watch out!
- Software used to produce object code is suspect until validated. All new software must be rigorously tested even if it isn't part of the application's mainstream. Static data can be just as wrong as any other kind and can have just as many bugs. Do not treat a routine that creates static data as "simple" because it "just stuffs a bunch of numbers into a table." Subject such code to the same testing rigor that you apply to running code.*
- The design remedy for the preprocessing situation is in the source language. If the language permits compile-time processing that can be used to particularize parameter values and data structures, and if the syntax of the compile-time statements is identical to the syntax of the

rest of the language, then the code will be subjected to the same validation and syntax checking as ordinary code. Such language facilities eliminate the need for most specialized preprocessors, table generators, and site adapters. For postprocessors, there is no magic, other than to recognize that users judge developers by the entire picture, installation software included.

(c) Information, Parameter, and Control:

- Static or dynamic data can serve in one of three roles, or in a combination of roles: as a parameter, for control, or for information. What constitutes control or information is a matter of perspective and can shift from one processing level to another. A scheduler receives a request to start a process. To the scheduler the identity of the process is information to be processed, but at another level it is control. My name is used to generate a hash code that will be used to access a disc record. My name is information, but to the disc hardware its translation into an address is control (e.g., move to track so-and-so).
- Information is usually dynamic and tends to be local to a single transaction or task. As such, errors in information (when data are treated as information, that is) may not be serious bugs. The bug, if any, is in the lack of protective data-validation code or in the failure to protect the routine's logic from out-of-range data or data in the wrong format. The only way we can be sure that there is data-validation code in a routine is to put it there. Assuming that the other routine will validate data invites latent bugs and maintenance problems. The program evolves and changes, and it is forgotten that the modified routine did the data validation for several other routines. *If* a routine is vulnerable to bad data, the only sane thing to do is to block such data within the routine; but it's even better to redesign it so that it is no longer vulnerable.
- Inadequate data validation often leads to finger pointing. The calling routine's author is blamed, the called routine's author blames back, they both blame the operators. This scenario leads to a lot of ego confrontation and guilt. "If only the other programmers did their job correctly," you say, "we wouldn't need all this redundant data validation and defensive code. I have to put in this extra junk because I'm surrounded by slobs!" This attitude is understandable, but not productive. Furthermore, if you really feel that way, you're likely to feel guilty about it. Don't blame your fellow programmer and don't feel guilt. Nature has conspired against us but given us a scapegoat. One of the unfortunate side effects of large-scale integrated circuitry stems from the use of microscopic logic elements that work at very low energy levels. Modern circuitry is vulnerable to electronic noise, electromagnetic radiation, cosmic rays, neutron hits, stray alpha particles, and other noxious disturbances. No kidding—alpha-particle hits that can change the value of a bit are a serious problem, and the semiconductor manufacturers are spending a lot of money and effort to reduce the random modification of data by alpha particles. Therefore, even if your fellow programmers did thorough, correct data validation, dynamic data, static data, parameters, and code can be corrupted. Program without rancor and guilt! Put in the data-validation checks and blame the necessity on sun spots and alpha particles!*

(d) Contents, Structure, and Attributes:

- Data specifications consist of three parts:
- **Contents**—The 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**—The size and shape and numbers that describe the data object, that is, the memory locations used to store the content (e.g., 16 characters aligned on a word boundary, 122 blocks of 83 characters each, bits 4 through 14 of word 17). Structures can have substructures and can be arranged into superstructures. A hunk of memory may have

several different structures defined over it—e.g., a two-dimensional array treated elsewhere as *N* one-dimensional arrays.

- **Attributes**—The specification of meaning, that is, the semantics associated with the contents of a data object (e.g., an integer, an alphanumeric string, a subroutine).
- The severity and subtlety of bugs increases as we go from content to attributes because things get less formal in that direction. Content has been dealt with earlier in this section. Structural bugs can take the form of declaration bugs, but these are not the worst kind of structural bugs. A serious potential for bugs occurs when data are used with different structures. Here is a piece of clever design. The programmer has subdivided the problem into eight cases and uses a 3-bit field to designate the case. Another programmer has four different cases to consider and uses a 2-bit field for the purpose. A third programmer is interested in the combination of the other two sets of cases and treats the whole as a 5-bit field that leads to thirty-two combined cases. We cannot judge, out of context, whether this is a good design or an abomination, but we can note that there is a different structure in the minds of the three programmers and therefore a potential for bugs. The practice of interpreting a given memory location under several different structures is not intrinsically bad. Often, the only alternative would be increased memory and many more data transfers.
- Attributes of data are the meanings we associate with data. Although some bugs are related to misinterpretation of integers for floating point and other basic representation problems, the more subtle attribute-related bugs are embedded in the application. Consider a 16-bit field. It could represent, among other things, a number, a loop-iteration count, a control code, a pointer, or a link field. Each interpretation is a different attribute. There is no way for the computer to know that it is proper or improper to add a control code to a link field to yield a loop count. We have used the same data with different meanings. In modern parlance, we have changed the **data type**. It is generally incorrect to logically or arithmetically combine objects whose types are different. Conversely, it is almost impossible to create an efficient system without doing so. Shifts in interpretation usually occur at interfaces, especially the human interface that is behind every software interface. See GANN76 for a summary of **type bugs**.
- The preventive measures for data-type bugs are in the source language, documentation, and coding style. Explicit documentation of the contents, structure, and attributes of all data objects is essential. The database documentation should be centralized. All alternate interpretation of a given data object should be listed along with the identity of all routines that have access to that object. A proper **data dictionary** (which is what the database documentation is called) can be as large as the narrative description of the code. The data dictionary and the database it represents must also be designed. This design is done by a high-level design process, which is as important as the design of the software architecture. My point of view here is dogmatic. Routines should not be administratively treated as if they have their “own” data declarations.* All data structures should be globally defined and centrally administered. Exceptions, such as a private work area, should be individually justified. Such private data structures must never be used by any other routine but the structure must still be documented in the data dictionary.
- It’s impossible to properly test software of any size (say 10,000+ statements) without central database management and a configuration-controlled data dictionary. I was once faced with such a herculean challenge. My first step was to try to create the missing data dictionary preparatory to any attempt to define tests. The act of dragging the murky bottoms of a hundred minds for hidden data declarations and semiprivate space in an attempt to create a data dictionary revealed so many data bugs that it was obvious that the system would defy

integration. I never did get to design tests for that project—it collapsed; and a new design was started surreptitiously from scratch.

- The second remedy is in the source language. **Strongly typed languages** prevent the inadvertent mixed manipulation of data that are declared as different types. A conversion in usage from pointer type to counter type, say, requires an explicit statement that will do the conversion. Such statements may or may not result in object code. Conversion from floating point to integer, would, of course, require object code, but conversion from pointer to counter might not. Strong typing forces the explicit declaration of attributes and provides compiler facilities to check for mixed-type operations. The ability of the user to specify types, as in Pascal, is mandatory. These data-typing facilities force the specification of data attributes into the source code, which makes them more amenable to automatic verification by the compiler and to test design than when the attributes are described in a separate data dictionary. In assembly language programming, or in source languages that do not have user-defined types, the remedy is the use of **field-access macros**. No programmer is allowed to directly access a field in the database. Access can be obtained only through the use of a field-access macro. The macro code does all the extraction, stripping, justification, and type conversion necessary. If the database structure has to be changed, the affected field-access macros are changed, but the source code that uses the macros does not (usually) have to be changed. The attributes of the data are documented with the field-access macro documentation. Another advantage of this approach is that the data dictionary can be automatically produced from the specifications of the field-access macro library.

(v) Coding Bugs:

- Coding errors of all kinds can create any of the other kinds of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. Failure to catch a syntax error is a bug in the translator. A good translator will also catch undeclared data, undeclared routines, dangling code, and many initialization problems. Any programming error caught by the translator (assembler, compiler, or interpreter) does not substantially affect test design and execution because testing cannot start until such errors are corrected. Whether it takes a programmer one, ten, or a hundred passes before a routine can be tested should concern software management (because it is a programming productivity issue) but not test design (which is a quality-assurance issue). But if a program has many source-syntax errors, we should expect many logic and coding bugs—because a slob is a slob is a slob.
- Given good source-syntax checking, the most common pure coding errors are typographical, followed by errors caused by not understanding the operation of an instruction or statement or the by-products of an instruction or statement. Coding bugs are the wild cards of programming. Unlike logic or process bugs, which have their own perverse rationality, wild cards are arbitrary.
- The most common kind of coding bug, and often considered the least harmful, are documentation bugs (i.e., erroneous comments). Although many documentation bugs are simple spelling errors or the result of poor writing, many are actual errors—that is, misleading or erroneous comments. We can no longer afford to discount such bugs because their consequences are as great as “true” coding errors. Today, programming labor is dominated by maintenance. This will increase as software becomes even longer-lived. Documentation bugs lead to incorrect maintenance actions and therefore cause the insertion of other bugs. Testing techniques have nothing to offer for these bugs. The solution lies in inspections, QA, automated data dictionaries, and specification systems.

(vi) Interface, Integration, and System Bugs:

(a) 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. Often there is a person on the other side of the interface. That person may be ingenious or ingenuous, but is frequently malevolent. The primary design criterion for an interface with the outside world should be **robustness**. All external interfaces, human or machine, employ a protocol. Protocols are complicated and hard to understand. The protocol itself may be wrong, especially if it's new, or it may be incorrectly implemented. Other external interface bugs include: invalid timing or sequence assumptions related to external signals; misunderstanding external input and output formats; and insufficient tolerance to bad input data. The test design methods of Chapters 6, 9, and 11 are suited to testing external interfaces.

(b) Internal Interfaces:

- Internal interfaces are in principle not different from external interfaces, but there are differences in practice because the internal environment is more controlled. 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 problems external interfaces have, as well as a few more that are more closely related to implementation details: protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values.
- To the extent that internal interfaces, protocols, and formats are formalized, the test methods of Chapters 6, 9, and 11 will be helpful. The real remedy is in the design and in standards. Internal interfaces should be standardized and not just allowed to grow. They should be formal, and there should be as few as possible. There's a trade-off between the number of different internal interfaces and the complexity of the interfaces. One universal interface would have so many parameters that it would be inefficient and subject to abuse, misuse, and misunderstanding. Unique interfaces for every pair of communicating routines would be efficient, but N programmers could lead to N^2 interfaces, most of which wouldn't be documented and all of which would have to be tested (but wouldn't be). The main objective of integration testing is to test all internal interfaces (BEIZ84).

(c) Hardware Architecture:

- It's easy to forget that hardware exists. You can have a programming career and never see a mainframe or minicomputer. When you are working through successive layers of application executive, operating system, compiler, and other intervening software, it's understandable that the hardware architecture appears abstract and remote. It is neither practical nor economical for every programmer in a large project to know all aspects of the hardware architecture. Software bugs related to hardware architecture originate mostly from misunderstanding how the hardware works. Here are examples: paging mechanism ignored or misunderstood, address-generation error, I/O-device operation or instruction error, I/O-device address error, misunderstood device-status code, improper hardware simultaneity assumption, hardware race condition ignored, data format wrong for device, wrong format expected, device protocol error, device instruction-sequence limitation ignored, expecting the device to respond too quickly, waiting too long for a response, ignoring channel throughput limits, assuming that the device is initialized, assuming that the device is not initialized, incorrect interrupt handling, ignoring hardware fault or error conditions, ignoring operator malice.

- The remedy for hardware architecture and interface problems is two-fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists. Hardware interface testing is complicated by the fact that modern hardware has very few buttons, switches, and lights. Old computers had lots of them, and you could abuse those buttons and switches to create wonderful anomalous interface conditions that could not be simulated any other way. Today's highly integrated black boxes rarely have such controls and, consequently, considerable ingenuity may be needed to simulate and test hardware interface status conditions. Modern hardware is better and cheaper without the buttons and lights, but also harder to test. This paradox can be resolved by hardware that has special test modes and test instructions that do what the buttons and switches used to do. The hardware manufacturers, as a group, have yet to provide adequate features of this kind. Often the only alternative is to use an elaborate hardware simulator instead of the real hardware. Then you're faced with the problem of distinguishing between real bugs and hardware simulator implementation bugs.

(d) Operating System:

- Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by a misunderstanding of what it is the operating system does. And, of course, the operating system could have bugs of its own. Operating systems can lull the programmer into believing that all hardware interface issues are handled by it. Furthermore, as the operating system matures, bugs in it are found and corrected, but some of these corrections may leave quirks. Sometimes the bug is not fixed at all, but a notice of the problem is buried somewhere in the documentation—if only you knew where to look for it.
- The remedy for operating system interface bugs is the same as for hardware bugs: 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 it will localize them and make testing easier.

(e) Software Architecture:

- Software architecture bugs are often the kind that are 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. They tend to be the most difficult kind of bug to find and exhume. Here is a sample of the causes of such bugs: assumption that there will be no interrupts, failure to block or unblock interrupts, assumption that code is reentrant or not reentrant, bypassing data interlocks, failure to close or open an interlock, assumption that a called routine is resident or not resident, assumption that a calling program is resident or not resident, assumption that registers or memory were initialized or not initialized, assumption that register or memory location content did not change, local setting of global parameters, and global setting of local parameters.
- The first line of defense against these bugs is the design. The first bastion of that defense is that there *be* a design for the software architecture. Not designing a software architecture is an unfortunate but common disease. The most elegant test techniques will be helpless in a complicated system whose architecture "just growed" without plan or structure. All test techniques are applicable to the discovery of software architecture bugs, but experience has shown that careful integration of modules and subjecting the final system to a brutal stress test are especially effective (BEIZ84).*

(f) Control and Sequence Bugs:

- System-level control and sequence bugs include: ignored timing; assuming that events occur in a specified sequence; starting a process before its prerequisites are met (e.g., working on data before all the data have arrived from disc); waiting for an impossible combination of

prerequisites; not recognizing when prerequisites have been met; specifying wrong priority, program state, or processing level; missing, wrong, redundant, or superfluous process steps.

- The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence-control mechanisms, such as an internal job control language, are useful. Sequence steps and prerequisites stored in tables and processed interpretively by a sequence-control processor or dispatcher make process sequences easier to test and to modify if bugs are discovered. **Path testing** as applied to **transaction flowgraphs**, as discussed in Chapter 4, is especially effective at detecting system-level control and sequence bugs.

(g) Resource Management Problems:

- Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers. Similarly, external mass storage units such as discs, are subdivided into memory-resource pools. Here are some resource usage and management bugs: required resource not obtained (rare); wrong resource used (common, if there are several resources with the same structure or different kinds of resources in the same pool); resource already in use; race condition in getting a resource; resource not returned to the right pool; fractionated resources not properly recombined (some resource managers take big resources and subdivide them into smaller resources, and Humpty Dumpty isn't always put together again); failure to return a resource (common); **resource deadlock** (a type A resource is needed to get a type B, a type B is needed to get a type C, and a type C is needed to get a type A); resource use forbidden to the caller; used resource not returned; resource linked to the wrong kind of queue; forgetting to return a resource.
- A design remedy that prevents bugs is always preferable to a test method that discovers them. The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.
- Complicated resource structures are often designed in a misguided attempt to save memory and not because they're essential. The software has to handle, say, large-, small-, and medium-length transactions, and it is reasoned that memory will be saved if three different-sized resources are implemented. This reasoning is often faulty because:
- **1.** Memory is cheap and getting cheaper.
- **2.** Complicated resource structures and multiple pools need management software; that software needs memory, and the increase in program space could be bigger than the expected data space saved.
- **3.** The complicated scheme takes additional processing time, and therefore all resources are held in use a little longer. The size of the pools will have to be increased to compensate for this additional holding time.
- **4.** The basis for sizing the resource is often wrong. A typical choice is to make the buffer block's length equal to the length required by an average transaction—usually a poor choice. A correct analysis (see BEIZ78, pp. 301-302) shows that the optimum resource size is usually proportional to the square root of the transaction's length. However, square-root laws are relatively insensitive to parameter changes and consequently the waste of using many short blocks for long transactions or large blocks to store short transactions isn't as bad as naive intuition suggests.
- The second design remedy is to centralize the management of all pools, either through centralized resource managers, common resource-management subroutines, resource-management macros, or a combination of these.
- I mentioned resource loss three times—it was not a writing bug. Resource loss is the most frequent resource-related bug. Common sense tells you why programmers lose resources.

You need the resource to process—so it's unlikely that you'll forget to get it; but when the job is done, the successful conclusion of the task will not be affected if the resource is not returned. A good routine attempts to get resources as soon as possible at a common point and also attempts to return them at a common point; but strange paths may require more resources, and you could forget that you're using several resource units instead of one. Furthermore, an exception-condition handler that responds to system-threatening illogical conditions may bypass the normal exit and jump directly to an executive level—and there goes the resource. The design remedies are to centralize resource fetch-and-return within each routine and to provide macros that return all resources rather than just one. Resource-loss problems are exhumed by path testing (Chapter 3), by transaction-flow testing (Chapter 4), data-flow testing (Chapter 5), and by stress testing (BEIZ84).

(h) Integration Bugs:

- **Integration bugs** are bugs having to do with the integration of, and with the interfaces between, presumably working and tested components. Most of these bugs result from inconsistencies or incompatibilities between components. All methods used to transfer data directly or indirectly between components and all methods by which components share data can host integration bugs and are therefore proper targets for integration testing. The communication methods include data structures, call sequences, registers, semaphores, communication links, protocols, and so on. Integration strategies and special testing considerations are discussed in more detail in BEIZ84. While integration bugs do not constitute a big bug category (9%) they are an expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures, often during the height of system debugging. Test methods aimed at interfaces, especially domain testing (Chapter 6), syntax testing (Chapter 9), and data-flow testing when applied across components (Chapter 5), are effective contributors to the discovery and elimination of integration bugs.

(i) System Bugs:

- **System bugs** is a catch-all phrase covering all kinds of bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating system. System testing as a discipline is discussed in BEIZ84. The only test technique that applies obviously and directly to system testing is transaction-flow testing (Chapter 4); but the reader should keep in mind two important facts: (1) all test techniques can be useful at all levels, from unit to system, and (2) there can be no meaningful system testing until there has been thorough component and integration testing. System bugs are infrequent (1.7%) but very important (expensive) because they are often found only after the system has been fielded and because the fix is rarely simple.

(vii) Test and Test Design Bugs:

(a) Testing:

- Testers have no immunity to bugs (see the footnote on page 20). Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently they can have bugs. The virtue of independent functional testing is that it provides an unbiased point of view; but that lack of bias is an opportunity for different, and possibly incorrect, interpretations of the specification. Although test bugs are not software bugs, it's hard to tell them apart, and much labor can be spent making the distinction. Also, consider the maintenance programmer—does it matter whether she's worked 3 days to chase and fix a real bug or wasted 3 days chasing a chimerical bug that was really a faulty test specification?

(b) Test Criteria:

- The specification is correct, it is correctly interpreted and implemented, and a seemingly proper test has been designed; but the criterion by which the software's behavior is judged is incorrect or impossible. How would you, for example, "prove that the entire system is free of bugs?" If a criterion is quantitative, such as a throughput or processing delay, the act of measuring the performance can perturb the performance measured. The more complicated the criteria, the likelier they are to have bugs.

(c) Remedies:

- The remedies for test bugs are: test debugging, test quality assurance, test execution automation, and test design automation.
- **Test Debugging**—The first remedy for test bugs is testing and debugging the tests. The differences between test debugging and program debugging are not fundamental. Test debugging is usually easier because tests, when properly designed, are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests, and therefore the complicated interactions that usually plague software designers are less frequent. We have no magic prescriptions for test debugging—no more than we have for software debugging.
- **Test Quality Assurance**—Programmers have the right to ask how quality in independent testing and test design is monitored. Should we implement test testers and test—tester tests? This sequence does not converge. Methods for test quality assurance are discussed in *Software System Testing and Quality Assurance* (BEIZ84).
- **Test Execution Automation**—The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers, and the like were all developed to reduce the incidence of programmer and/or operator errors. Test execution bugs are virtually eliminated by various test execution automation tools, many of which are discussed throughout this book. The point is that "manual testing" is self-contradictory. If you want to get rid of test execution bugs, get rid of manual execution.
- **Test Design Automation**—Just as much of software development has been automated (what is a compiler, after all?) much test design can be and has been automated. For a given productivity rate, automation reduces bug count—be it for software or be it for tests.

(viii) Testing and Design Style:

- This is a book on test design, yet this chapter has said a lot about programming style and design. You might wonder why the productivity of one programming group is as much as 10 times higher than that of another group working on the same application, the same computer, in the same language, and under similar constraints. It should be obvious—bad designs lead to bugs, and bad designs are difficult to test; therefore, the bugs remain. Good designs inhibit bugs before they occur and are easy to test. The two factors are multiplicative, which explains the large productivity differences. The best test techniques are useless when applied to abominable code: it is sometimes easier to redesign a bad routine than to attempt to create tests for it. The labor required to produce new code plus the test design and execution labor for the new code can be much less than the labor required to design thorough tests for an undisciplined, unstructured monstrosity. Good testing works best on good code and good designs. And no test technique can ever convert garbage into gold.

FLOW GRAPHS AND PATH TESTING

(1) Basics concepts of path testing:

(i) Motivation and Assumptions:

(a) Path testing

- A sequence of statements which starts at an entry and ends at an exit by passing all the existing junctions, decisions etc is known as path.
- Path testing is a process which involves all the available paths in a program from an entry to an exit in such a way that the entire path is thoroughly tested.
- If the set of paths is properly chosen, then we have achieved some measure of test thoroughness.

(b) Motivation

- Path–testing techniques are the oldest of all structural test techniques.
- Path–testing techniques were also the first techniques to come under theoretical scrutiny.
- There is considerable evidence that path testing was independently discovered and used many times in many different places.
- 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 (i.e., source code).
- It is most often used by programmers to unit–test their own code.

(c) 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. As another example, "IF A is *true* THEN DO X ELSE DO Y", instead of "IF A is *false* THEN . . ."
- We also assume, in path testing, that specifications are correct and achievable, that there are no processing bugs other than those that affect the control flow, and that data are properly defined and accessed.

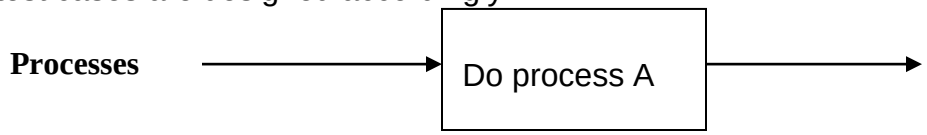
(ii) Control Flowgraphs:

(a) About control flowgraphs:

- The control flowgraph is a graphical representation of a program's control structure.
- A control flowgraph is a form of a flowchart which does not deal with the internal structure of the process rather it shows the data flow and the control flow between the processes.
- It uses the elements process blocks, decisions and junctions.

(i) Process Block

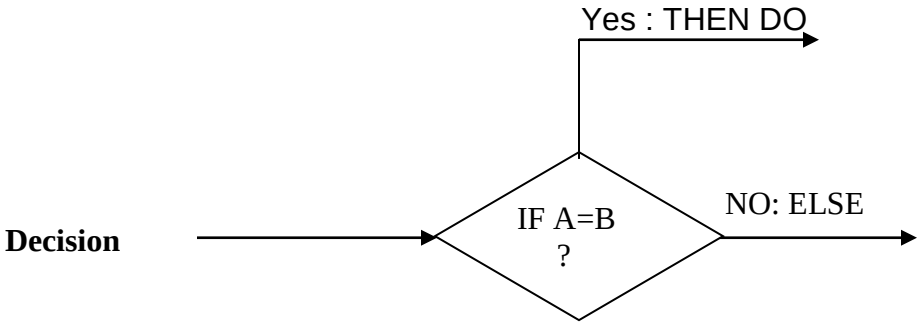
- A process block* is a sequence of program statements uninterrupted by either decisions or junctions.
- Formally, it is a sequence of statements such that if any one statement of the block is executed, then all statements are executed.
- Here once a process block is initiated, every statement within it will be executed.
- Every process has an entry and an exit and consists of a single or series of statements.
- Control flow graph are not concerned with the details of operations in a process block so, the test cases are designed accordingly.

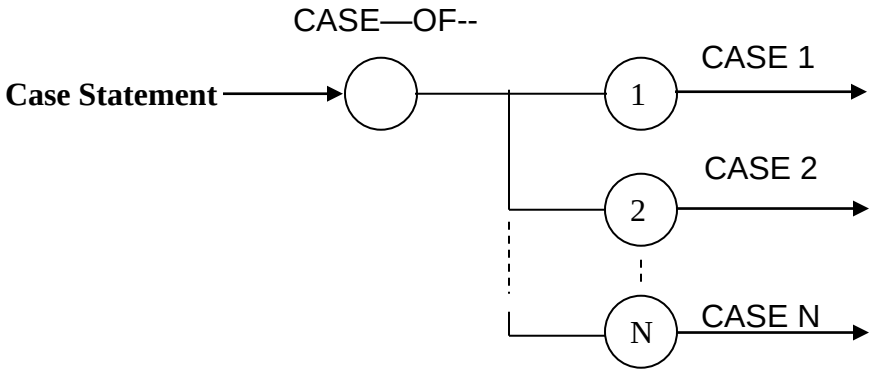


(ii) Decisions and Case Statements:

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

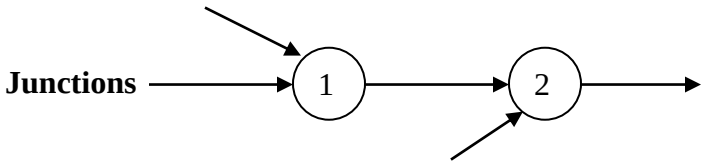
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- The FORTRAN IF and the Pascal IF–THEN–ELSE constructs are decisions, although they also contain processing components.
- While most decisions are two–way or binary**,** some (such as the FORTRAN IF) are three–way branches in control flow.
- The design of test cases is generally easier with two–way branches than with three–way branches, and there are also more powerful test–design tools that can be used.
- Any decision can split the control flow into different way branches.
- This multi way branches can be termed as case statements.
- The designing of test cases for decision and case statements are same.





(iii) Junctions:

- A junction is a point in the program where the control flow can merge.
- That is all the control flows can merge at a point in a program which is known as junction.
- In other words a node with more than one input line is known as junction.
- Examples of junctions are: the target of a jump or skip instruction in assembly language, a label that is the target of a GOTO, the END–IF and CONTINUE statements in FORTRAN, and the Pascal statement labels, END and UNTIL.



Control flowgraph advantages:

- Control flowgraph eliminates the occurrence of some problems which results from expanding the visual complexities.
- Control flowgraph treats all the steps inside a process as a single process entity and shows only data and control flow to and from that entity there by reducing the complexity of structure.
- Control flowgraphs can be referred to as a modern approach for representation of flows.
- Control flowgraph gives the precise and clear view of the program structure, the directions of data flow etc.

Control flowgraph disadvantages:

- Control flowgraph plays an important role in representing the program control structure, but are sparsely available due to the scarcity of control flowgraph generators.
- The information needed to produce a control flowgraph is not provided by most of the compilers.
- Although the control flowgraphs are informative, but causes inconveniency while working.
- Control flowgraph structure is similar to many programming structures and is very difficult to differentiate..

(b) Control Flowgraphs Versus Flowcharts

- Flowchart is a graph which represents the control structure of the program, as well as the internal structure of each and every process or process block.
- Control flowgraph is also a graph which represents the control structure of a program, but it excludes the detailed structure of process blocks.
- All the steps inside a process are shown using flowchart in addition to the control flows, but control flowgraph considers all the steps as a single process entity and shows only the control flows to and from that process entity.
- Flowchart shows the internal flows of each process so, it is difficult to identify the actual control flows between different processes.
- Whereas control flowgraphs shows the control and data flow only between processes, there by complexity is reduced.
- Flowcharts had lost its importance because of the detailed information, it provides which is not in use for process design.
- We can also use flowchart for representing the control and data flows in a traditional way and control flowgraphs as the modern approach for representation of flows.
- Flowcharts can easily be drawn manually using available flowchart generators whereas control flowgraph can be drawn difficult.
- In control flowgraphs, we don't show the details of what is in a process block; indeed, the entire block, no matter how many statements in it, is shown as a single process.
- In flowcharts, conversely, every part of the process block is drawn: if a process block consists of 100 steps, the flowchart may have 100 boxes.
- Flowchart has a box to represent each and every process step which is not the case with control flowgraph, only the outline of process block is shown in control flowgraph.

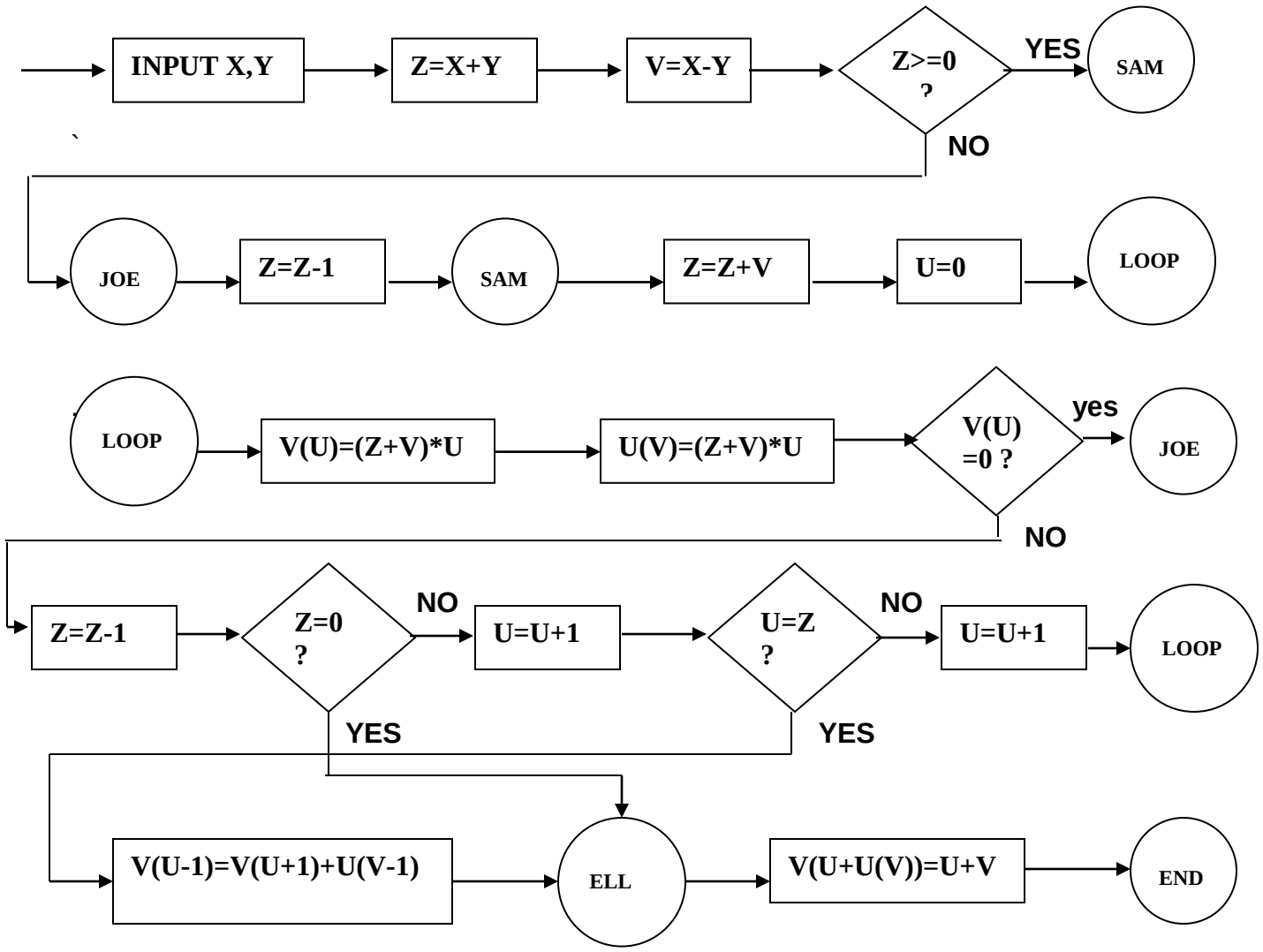
(c) Notational Evolution

- The control flowgraph is a simplified representation of the program's structure.
- To understand its creation and use, we'll go through an example, written in a FORTRAN–like **program design language** (PDL).
- The code is given below.

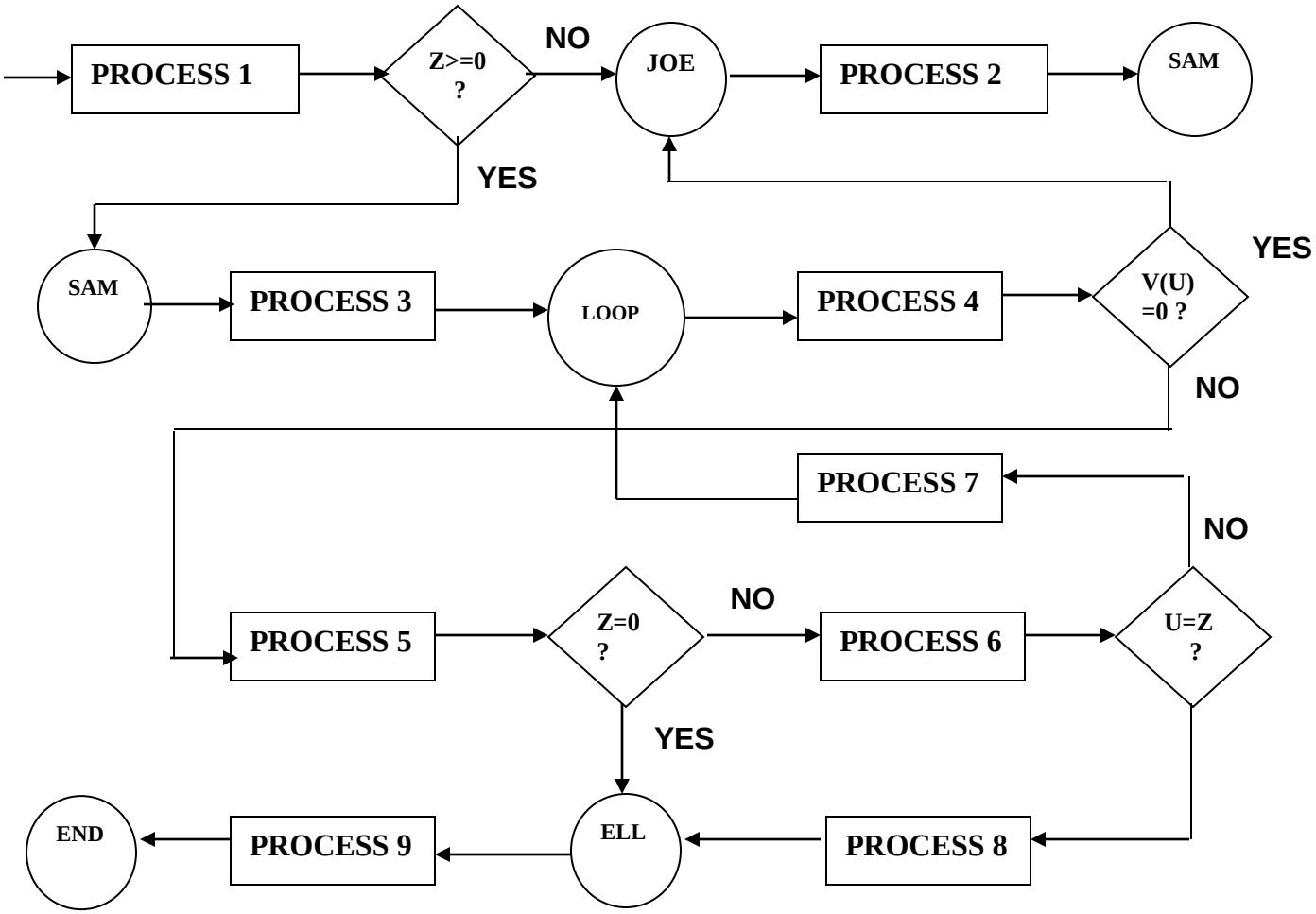
CODE* (PDL)

```
INPUT X,Y
Z:=X+Y
V:=X-Y
IF Z>=0 GOTO SAM
JOE:Z:=Z-1
SAM:Z:=Z+V
FOR U=0 TO Z
V(U),U(V):=(Z+V)*V
IF V(U)=0 GOTO JOE
Z:=Z-1
IF Z=0 GOTO ELL
U:=U+1
NEXT U
V(U+1)+U(V-1)
ELL:V(U+U(V)):=U+V
END
```

- One-to-one Flowchart for the above code is given by



- Control flowgraph for the above example is given by



(d) Flowgraph–Program Correspondence

- A flowgraph is a pictorial representation of a program and not the program itself.
- We can't always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as IF–THEN–ELSE constructs, consist of a combination of decisions, junctions, and processes.
- Furthermore, the translation from a flowgraph element to a statement and vice versa is not always unique.
- A FORTRAN DO has three parts: a decision, an end–point junction, and a process that iterates the DO variable.
- The FORTRAN IF–THEN–ELSE has a decision, a junction, and three processes (including the processing associated with the decision itself).
- Therefore, neither of these statements can be translated into a single flowgraph element.
- Some computers have looping, iterating, and EXECUTE instructions or other instruction options and modes that prevent the direct correspondence between instructions and flowgraph elements.
- Such differences are so familiar to us that we often code without conscious awareness of their existence.
- It is, however, important that the distinction between a program and its flowgraph representation be kept in mind during test design.

- An improper translation from flowgraph to code during coding can lead to bugs, and an improper translation (in either direction) during test design can lead to missing test cases and consequently, to undiscovered bugs.

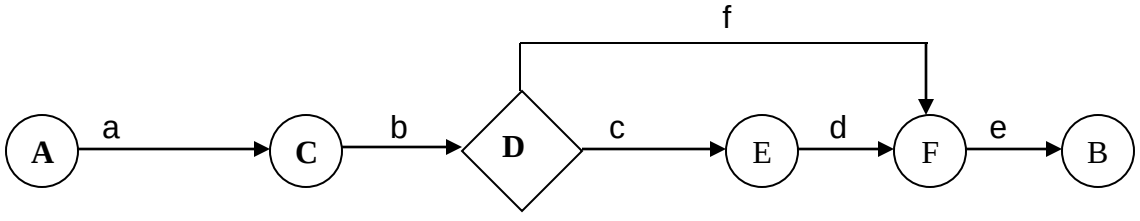
(e) Flowgraph and Flowchart Generation

- The control flowgraph is a simplified version of the earlier flowchart.
- Flowcharts can be (1) hand–drawn by the programmer, (2) automatically produced by a flowcharting program based on a mechanical analysis of the source code, or (3) semiautomatically produced by a flowcharting program based in part on structural analysis of the source code and in part on directions given by the programmer.
- The semiautomatic flowchart is most common with assembly language source code.
- A flowcharting package that provides controls over how statements are mapped into process boxes can be used to produce a flowchart that is reasonably close to the control flowgraph.
- You do this by starting process boxes just after any decision or GOTO target and ending them just before branches or GOTOs.

(iii) Path Testing:

(a) Paths, Nodes, and Links

- A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Every path consists of a set of processes known as links.
- A direct connection between two nodes is also called a “process”.
- Links can be denoted by an arrow and can represented by the lower case letters.
- A path segment is a succession of consecutive links that belongs to some path.
- The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path.
- An alternative way to measure the length of a path is by the number of nodes traversed.
- Nodes are mainly denoted by small circles. A node which has more than one input link is known as a junction, and a node which has more than one output link is referred to as a decision.
- Nodes can be labeled by an alphabets or numbers.
- If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.
- Because links are named by the pair of nodes they join, the name of a path is the name of the nodes along the path.



- There are two different paths from an entry (A) to an exit (B), they are ACDEFB and ACDFB respectively. In these two ACDFB is the shortest path between an entry and an exit.
- In all the nodes (A,B,C,D,E,F), D is the decision which has 2 output links, and F is a junction which has two input links.
- The a,b,c,d,e,f are all the available links.

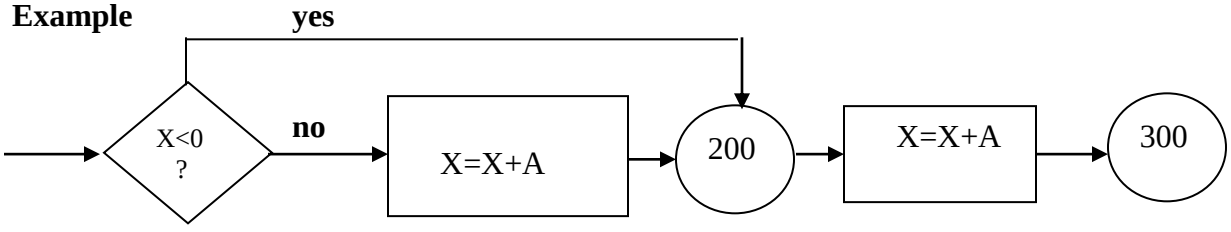
(b) Multi–Entry/Multi–Exit Routines

- Multi-entry means, multiple entry points and multi-exit refers to multiple exit points.
- Generally all routines and programs have a single entry and a single exit.
- There are certain situations in which it is appropriate to change the routine and choose an alternate way to normal control structure.
- There is no justifiable reason which forces you to change the routine.
- You may want to choose an alternate routine, when an illegitimate condition occur and will damage the system's data, if that path is continued further.
- The other reason might be the occurrence of several fluctuations during the processing of same path.
- Hence changing of route is advantageous in such situations by placing an entry point in a routine which sends the flow to appropriate location.
- If a routine can have several different kinds of outcomes, then an exit parameter should be used.
- As there is no direct connection between entry and exit so control flow is managed by reviewing the parameter values of entry and exit in both directions of the routine.
- The main drawback of multi-entry and multi-exit routines is that all the test cases are difficult to cover because the control flow between various processes can't be determined easily due to multiple entry and exit points.

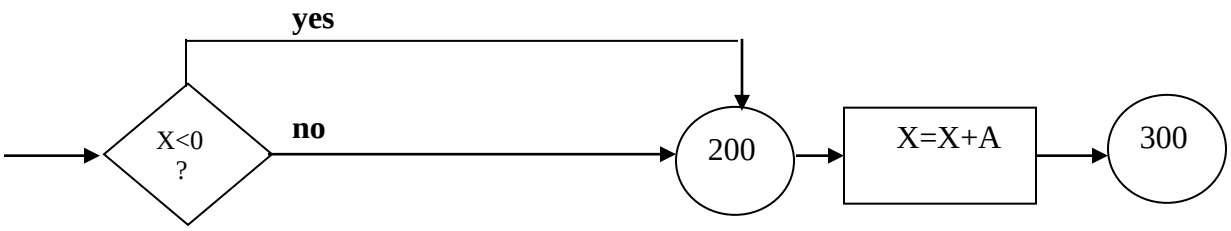
(c) Fundamental Path Selection Criteria

- There are many paths between the entry and exit of a typical routine.
- Path selection mainly deals with the selection of an optimal path between its entry and exit.
- If a routine contains decisions or loops inside it, then there will be more number of paths.
- For example 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.
- If a routine has one loop, each pass through that loop (once, twice, three times, and so on) constitutes a different path through the routine, even though the same code is traversed each time.
- A lavish test approach might consist of testing all paths, but that would not be a complete test, because a bug could create unwanted paths or make mandatory paths unexecutable.
- Complete testing involves
    1. Exercise every path from entry to exit.
    2. Exercise every statement or instruction at least once.
    3. Exercise every branch and case statement, in each direction, at least once.
- If prescription 1 is followed then prescriptions 2 and 3 are automatically followed, but prescription 1 is impractical for most routines.

Example



- For X is less than zero, the output is X+A while X is greater than or equal to zero the output is X+2A because decision doubles the number of paths.
- If we execute all the statements but not the branches in the above example we would get the bug.



- For the above example if X is less than zero the output is correct, but for any positive value the output will be X=X+A which is wrong.
- A static analysis that is an analysis based on examining the source code or structure cannot determine whether a piece of code is or is not reachable.
- Only a dynamic analysis that is an analysis based on the code's behavior while running can determine whether code is reachable or not.

(d) Path–Testing Criteria

- There are three path testing criteria.
- The notation **P_1, P_2, …, P_∞** should alert you to the fact that there is an infinite number of such strategies, but even that's insufficient to exhaust testing.

(i) Path Testing (P_∞):

- Path testing deals with the execution of paths if we have tested all the available control flow paths we have achieved 100% path coverage which is mostly impossible.
- The word coverage refers to combinational value of 100% statement coverage and branch coverage.
- It is represented as (C_1 +C_2), where C_1 refers to statement coverage and C_2 refers to branch coverage.
- Hence this type of coverage is also referred as completed coverage.

(ii) Statement Testing (P_1):

- Statement testing deals with the execution of all the statements inside a program at least once.
- The process of performing possible tests in order to achieve statement testing is called 100% statement coverage.
- Statement coverage is also known as 100% node coverage.
- We denote this by C_1.

(iii) Branch Testing (P_2):

- Branch testing deals with the execution of all the branches at least once in the program.
- The process of performing possible tests in order to achieve branch testing is called 100% branch coverage.
- Branch coverage is also known as link coverage.
- We denote branch coverage by C_2.

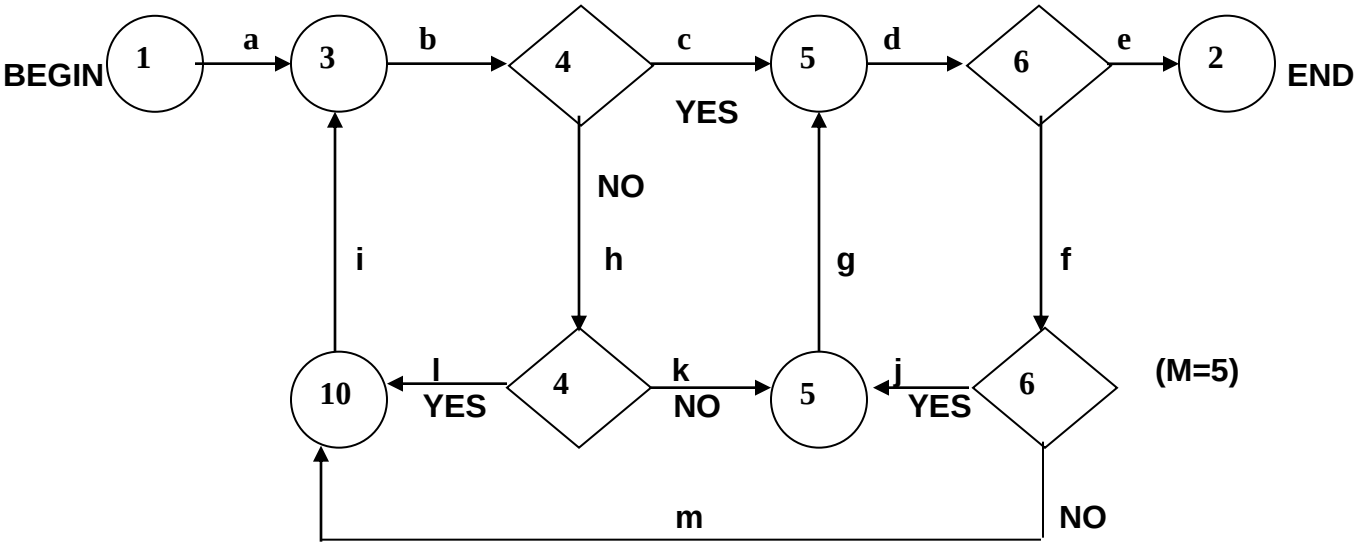
(e) Common Sense and Strategies

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- Statement coverage is established as a minimum testing requirement in the IEEE unit test standard.
- Statement and branch coverage have also been used for more than two decades as minimum mandatory unit test requirements for new code at IBM and other major computer and software companies.
- The justification for insisting on statement and branch coverage isn't based on theory but on common sense.

- Also with our common sense, we can classify code with much probability of having bugs and code with less probability, separately.
- Keeping the code with lower probability of bugs untested may not be wrong because this code will probably have less or no bugs.
- The code with higher probability of bugs is tested thoroughly to remove all the bugs. Even if we are skipping some part of this code it will not create a big one because this portion is tested many times in the entire testing process.

(f) Which Paths

- We must pick enough paths to achieve C1 + C2.
- It's better to take many simple paths than a few complicated paths.
- An example of path selection is given below.



- As we trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
- Start at the beginning and take the most obvious path to the exit—it typically corresponds to the normal path.
- The most obvious path in above figure is (1,3,4,5,6,2), if we name it by nodes, or *abcde* if we name it by links.
- Then take the next most obvious path, *abhkgde.* All other paths in this example lead to loops.
- Take a simple loop first—building, if possible, on a previous path, such as *abhlibcde.*
- Then take another loop, *abcdfjgde.* And finally, *abcdfmibcde.*
- The above paths lead to the following table.

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

- After you have traced a covering path set on the master sheet and filled in the table for every path, check the following.
  1. Does every decision have a YES and a NO in its column? (C2)
  2. Has every case of all case statements been marked? (C2)
  3. Is every three–way branch (less, equal, greater) covered? (C2)
  4. Is every link (process) covered at least once? (C1)*
- Select successive paths as small variations of previous paths.
- Try to chance only one thing at a time that is only one decision's outcome if possible.
- It is better to have several paths, each differing by only one thing, than one path that covers more but along which several things change.
- The abcd segment in the above example is common to many paths

(g) Path selection rules:

(a) Selection of simple path:

- Select an entry/exit path which is simple and assign selected path with either nodes or links.

(b) Selection of additional paths:

- After selection of simple path, the next obvious path is selected.
- This method of selecting successive paths can be done by making small changes to the previous paths.
- Unlike long and complex paths, various small paths are selected which involves gradual variations.
- In path selection Select paths with no loops, Select shorter paths and Select simple and sensible paths.

(c) Selection of Non-functional Sensible paths:

- Select additional paths in such a way that coverage is achieved through the non-functional sensible paths.
- This type of selection should be preferred only if coverage is essential.

(d) Meet the user Requirements:

- All possible paths should be selected in order to meet the requirements of a user.
- This process is repeated until statement (C_1) and branch (C_2) coverages are achieved.
- During this process checking is carried out on each and every decision statement, branch covering, link covering etc.
- Statement coverage and branch coverage ($C_1 + C_2$) does not support loop-related bugs.

(iv) Loops:

(a) The Kinds of Loops

- There are three kinds of loops.
- They are nested, concatenated and horrible loops.

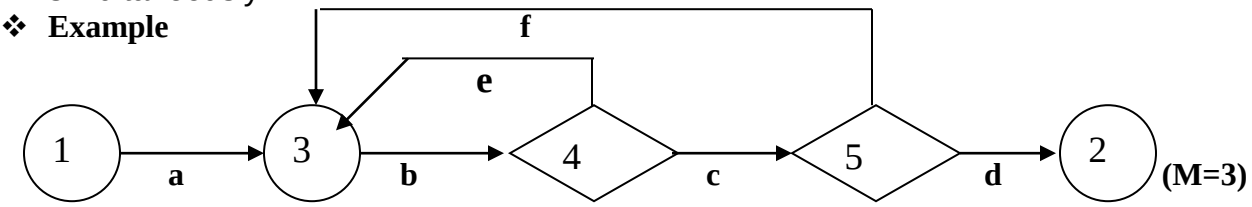
(i) Cases for a Single Loop:

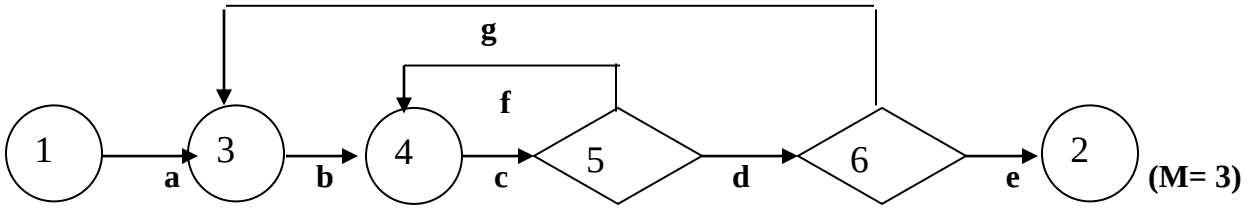
- A single loop can be covered with two cases: looping and not looping.
- The different cases for a single loop are
- Case 1—Single Loop, Zero Minimum, N Maximum, No Excluded Values.
- Case 2—Single Loop, Nonzero Minimum, No Excluded Values.
- Case 3—Single Loops with Excluded Values.

. **(ii) Nested Loops:**

- The nested loops are quite complicated i.e. a loop within another loop is known as nested loop.
- It is very expensive to test the path which contains nested loop because of its complexity.

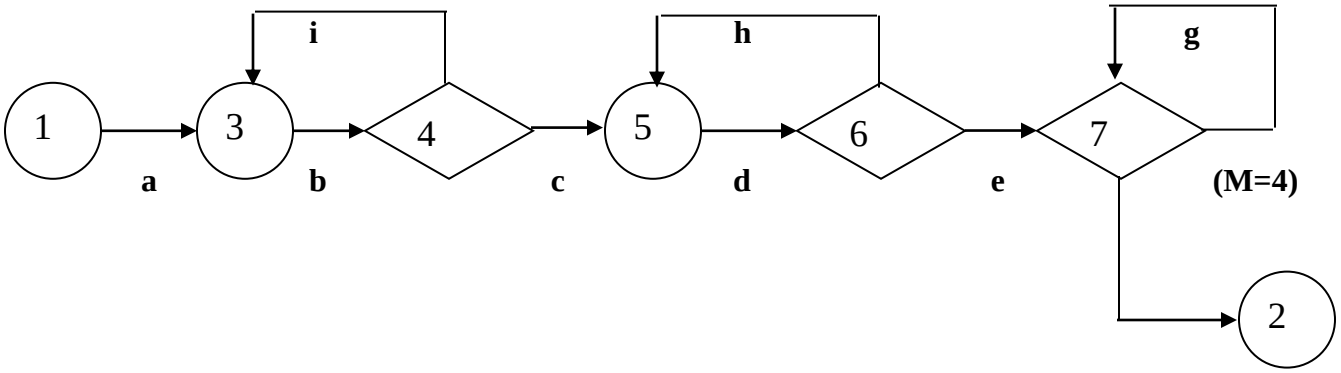
- If you had five tests for one loop, a pair of nested loops would require 25 tests, and three nested loops would require125.
- To overcome this complexity we have to follow some steps.
  1. Start at the innermost loop. Set all the outer loops to their minimum values.
  2. Test the minimum, minimum + 1, typical, maximum – 1, and maximum for the innermost loop, while holding the outer loops at their minimum–iteration–parameter values. Expand the tests as required for out–of–range and excluded values.
  3. If you've done the outermost loop, GOTO step 5, ELSE move out one loop and set it up as in step 2—with all other loops set to typical values.
  4. Continue outward in this manner until all loops have been covered.
  5. Do the five cases for all loops in the nest simultaneously.
- This procedure works out to twelve tests for a pair of nested loops, sixteen for three nested loops, and nineteen for four nested loops.
- Practicality may prevent testing in which all loops achieve their maximum values simultaneously**.**
- **Example**





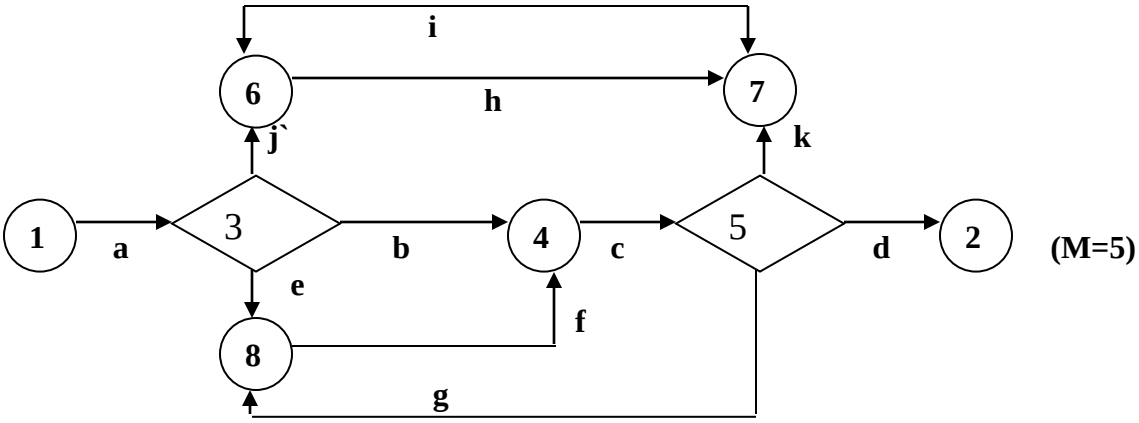
(iii) Concatenated Loops:

- Concatenated loops are the loops which reside one beside the other on the same path.
- In other words, when there exits two adjacent loops on the same path such that, an exit of one loop serves as an entry point for the other loop, then the loops are said to be concatenated.
- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- If one loop's repetition value depends on the repetition value of other loop and both lie on same path they can be termed as nested loops.



(iv) Horrible Loops:

- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- Horrible loops are the complexed of all the three loops. This complex structure of horrible loops makes it very difficult to be tested.
- The design of test cases for horrible loops is indefinite and is too many to execute. Hence horrible loops must be avoided.



(f) Loop–Testing Time

- Any kind of loop can lead to long testing time, especially if all the extreme value cases are to be attempted (MAX – 1, MAX, MAX + 1).
- This situation is obviously worse for nested and dependent concatenated loops.
- In the context of real testing, most tests take a fraction of a second to execute, and even deeply nested loops can be tested in seconds or minutes.
- The unreasonably long test execution times (i.e., hours or centuries) could indicate bugs in the software or the specification.
- Consider nested loops in which testing the combination of extreme values leads to long test times. You have several options:

  1. Show that the combined execution time results from an unreasonable or incorrect specification. Fix the specification.

  2. Prove that although the combined extreme cases are hypothetically possible, they are not possible in the real world. That is, the combined extreme cases cannot occur.

  3. Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.

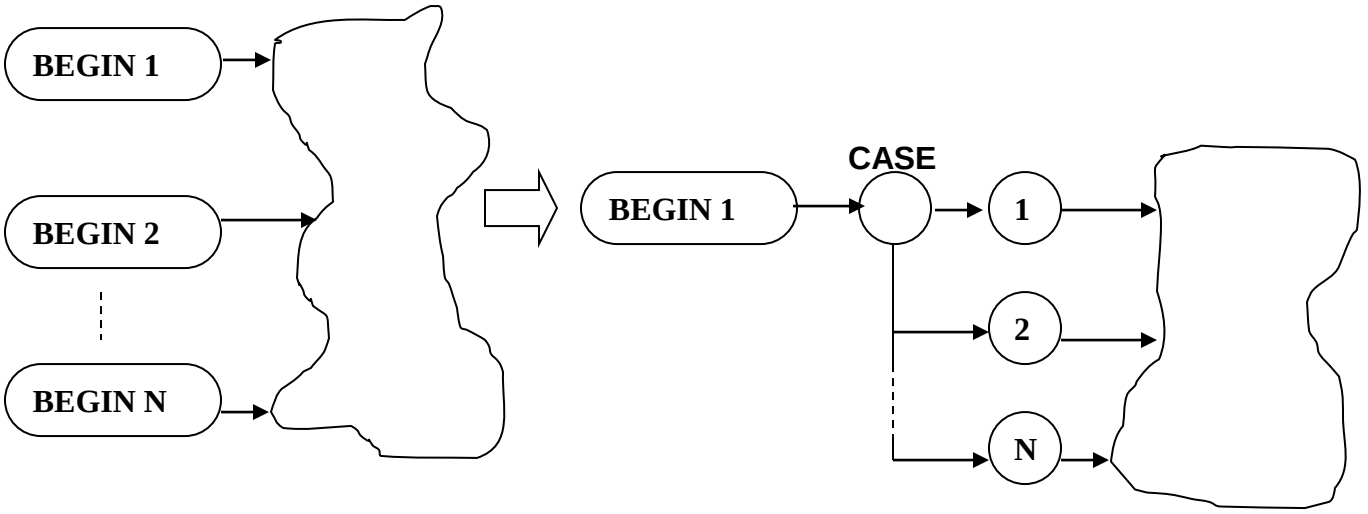
  4. Test with the extreme–value combinations, but use different numbers.

(v) More on Testing Multi–Entry/Multi–Exit Routines:

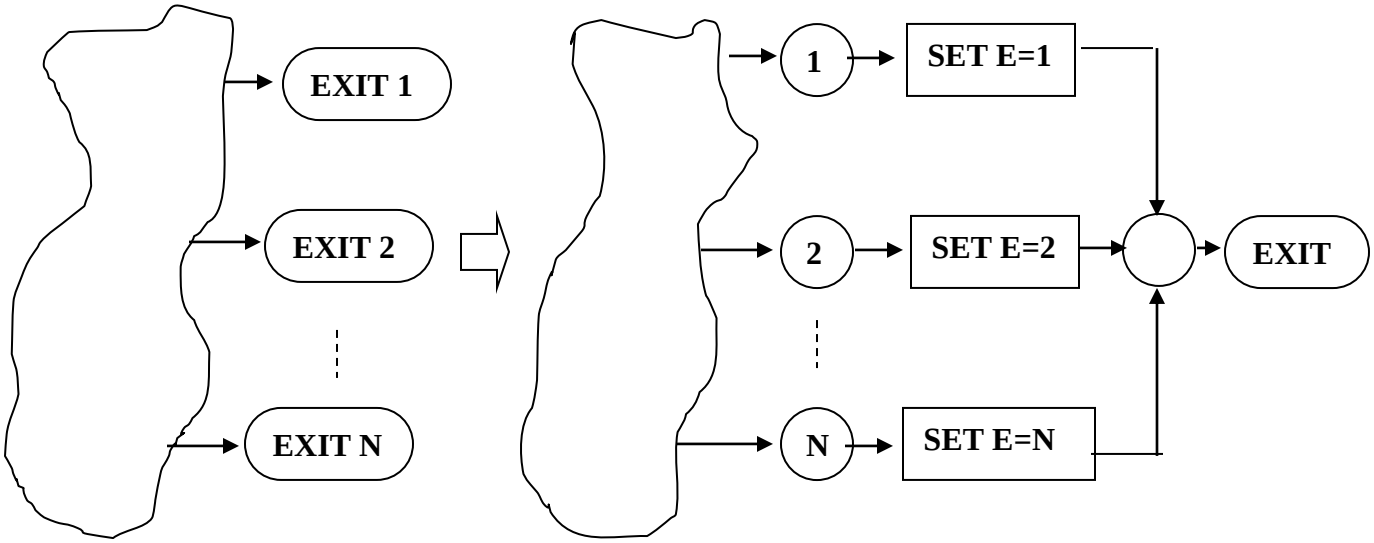
(a) A Weak Approach

- To test the program with multi-entry and multi-exit routines are as follows.
- First, built the fictitious single entry routine and fictitious exit routine with fictitious case statements and processes respectively.
- Secondly concentrate on fictitious common junction. This fictitious code will help you to organize the test case design for multi-entry and multi-exit routines.
- This technique involves a lot of extra work because you must examine the cross-reference listings to find all references to the labels that correspond to the multiple entries.
- All the designers of routines should know how they want to exit, but it's difficult to control an entry that can be initiated by many other programmers.
- The Conversion of Multi-exit or Multi-entry routines is given by the following figures.

(i) A Multi-entry routine is converted to an equivalent single-entry routine with an entry parameter and a controlling case statement.



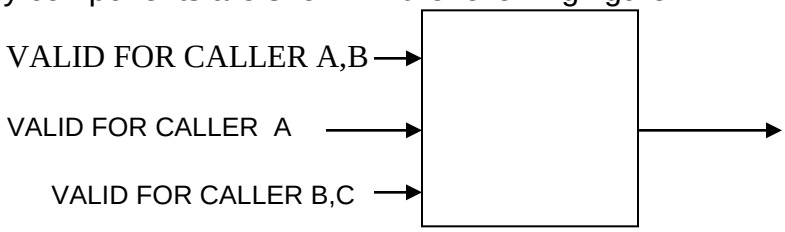
(ii) A Multi-exit routine is converted to an equivalent single-exit routine with an exit parameter.



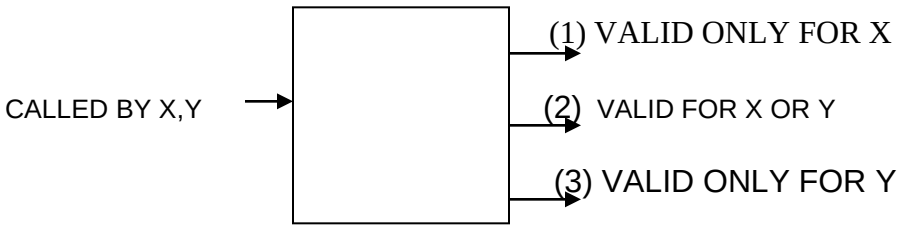
(b) The Integration Testing Issue

- Treating the multi–entry/multi–exit routine by using a fictional entry case statement and a fictional exit parameter is a weak approach because it does not solve the essential testing problem.
- The essential problem is an integration testing issue and has to do with paths within called components.
- For example we have a multi–entry routine with three entrances and three different callers. The first entrance is valid for callers A and B, the second is valid only for caller A, and the third is valid for callers B and C.
- Just testing the entrances doesn't do the job because in integration testing it's the interface, the validity of the call that must be established.
- In integration testing, we would have to do at least two tests for the A and B callers—one for each of their entrances. Note also that, in general, during unit testing we have no idea who the callers are to be.

- Multi-entry components are shown in the following figure.



- Multi-exit routine is shown in the following figure.



- The above multi-exit routine has three exits labeled 1, 2, and 3.
- It can be called by components X or Y. Exits 1 and 2 are valid for the X calls and 2 and 3 are valid for the Y calls.
- Component testing must not only confirm that exits 1 and 2 are taken for the X calls, but that there are no paths for the X calls that lead to exit 3—and similarly for exit 1 and the Y calls.
- But when we are doing unit tests, we do not know who will call this routine with what restrictions. As for the multi–entry routine, we must establish the validity of the exit for every caller.
- Note that we must confirm that not only does the caller take the expected exit, but also that there is no way for the caller to return via the wrong exit.
- When we combine the multi–entry routine with the multi–exit routine, we see that in integration testing we must examine every combination of entry and exit for every caller.
- Since we don't know, during unit design, which combinations will or will not be valid, unit testing must at least treat each such combination as if it were a separate routine.
- Thus, a routine with three entrances and four exits results in twelve routines' worth of unit testing.
- Integration testing is made more complicated in proportion to the number of exits, or fourfold.

(c) The Theory and Tools Issue

- A well-formed software is a software, which has single entry and single exit with a rigid structure.
- Software which does not have this property is called ill-formed.
- The other characteristic of well–formed software is to insist on strict structuring in addition to single–entry/single–exit.
- An assumption that multi-entry and multi-exit routines can't occur in testing theory has been followed.
- Such multi-entry and multi-exit routines come under ill formed routines.
- Before applying the theoretical rules, it is better to confirm whether the software is well-formed or ill-formed.
- Ill-formed (multi-entry and multi-exit) software does not have any structure so, testing of one component does not guarantee the test results for another.
- Even test generators may not be able to generate test cases for ill-formed software.

(d) Strategy Summary

The proper way to test multi–entry or multi–exit routines is:

1. Get rid of them.
2. Completely control those you can't get rid of.
3. Supply the imaginary input case statements, and exit parameters to control flowgraph in order to design test cases for these routines.
4. Do stronger unit testing by treating each and every entry/exit combination considered as a completely different routine.
5. Multi-entry and multi-exit routines are assumed to be more unusual and dangerous so, integration testing is performed with more efforts and concentration.
6. Be sure you understand that test cases designed based on your assumption are suitable for multi-entry and multi-exit routines.

(vi) Effectiveness of Path Testing:

(a) Effectiveness and Limitations

- Unit testing is comparatively stronger than path testing which is stronger than statement and branch testing.
- Unit testing can catch up to 65% of bugs in overall structure, this implies that path testing captures approximately 35% of bugs in the overall structure as per statistical reports.
- Path testing is more effective for unstructured than for structured software.
- Apart from effectiveness, path testing also has certain limitations.

1. Planning to cover does not mean you will cover. Path testing may not cover if you have bugs.
2. Path testing has to be combined with other methods to improve the overall performance in terms of percentage.
3. Unit level path testing does not concentrate on integration issues which may result in interface errors.
4. Database and data–flow errors may not be caught.
5. Illegitimate functions or missed functions cannot be identified during path testing.
6. Not all initialization errors are caught by path testing.
7. Specification errors can't be caught.

(b) A Lot of Work?

- Path testing involves a lot of work that is.
  - Development of control flowgraph.
  - Choosing a route that can cover all the paths, decisions and junctions in a flowgraph.
  - Determining the input values which satisfies each path expression for selecting the respective paths.
  - Writing test cases for loops.
- The statistics indicate that you will spend half of your time testing and debugging—presumably that time includes the time required to design and document test cases.
- Furthermore, the act of careful, complete, systematic, test design will catch as many bugs as the act of testing.
- It is worth that, the test design process, at all levels, and is at least as effective at catching bugs as is running the test designed by that process.

(c)More on How to Do It

- To trace the path from your code, you need a marking pen, a copying machine and a source code list.
- At first you may want to create the control flowgraph and use that as a basis for test design, but as you gain experience with practice, you'll find that you can select the paths directly on the source code without bothering to draw the control flowgraph.

- If you can path trace through code for debugging purposes then you can just as easily trace through code for test design purposes.
- And if you can't trace a path through code, are you a programmer then you do it with code almost the same way as you would with a pictorial control flowgraph.
- Choose your path and mark only the executed statements in case of "if-then-else statements".
- Also mark all the ongoing statements on a path with a marking pen by doing this you will accomplish C_1.
- Place or draw your marking on a master sheet with the marking pen (yellow).
- For achieving C_2 we need to identify and mark all the statements irrespective of its execution even for the if-then-else statements.

(vii) Variations:

- Branch and statement coverage as basic testing criteria are well established as effective, reasonable, and easy to implement.
- There are two main classes of variations:
  1. Strategies between P_2 and total path testing.
  2. Strategies weaker than P_1 or P_2.
- The stronger strategies typically require more complicated path selection criteria, most of which are impractical for human test design.
- Typically, the strategy has been embedded in a tool that either selects a covering set of paths based on the strategy or helps the programmer to do so.
- While research can show that strategy A is stronger than B in the sense that all tests generated by B are included in those generated by A, it is much more difficult to ascertain cost–effectiveness.
- For example, if strategy A takes 100 times as many cases to satisfy as B, the effectiveness of A would depend on the probability that there are bugs of the type caught by A and not by B.
- We have almost no such statistics and therefore we know very little about the pragmatic effectiveness of this class of variations.
- As an example of how we can build a family of path–testing strategies, consider a family in which we construct paths out of segments that traverse one, two, or three nodes or more.
- If we build all paths out single–node segments P_1 (hardly to be called a "path," then we have achieved C1. If we use two–node segments (e.g., links = P_2) to construct paths, we achieve C2.

(2) Predicates, Path Predicates, and Achievable Paths:

(i) General

- Selecting a path does not mean that it is achievable.
- If all decisions are based on variables whose values are independent of the processing and of one another, then all combinations of decision outcomes are possible (2^n outcomes for n binary decisions) and all paths are achievable: in general, this is not so.
- Every selected path leads to an associated boolean expression, called the path predicate expression, which characterizes the input values (if any) that will cause that path to be traversed.

(ii) Predicates

(a) Definition and Examples

- The direction taken at a decision depends on the value of decision variables.
- For binary decisions, decision processing ultimately results in the evaluation of a logical (i.e., boolean) function whose outcome is either TRUE or FALSE.

- Although the function evaluated at the decision can be numeric or alphanumeric, when the decision is made it is based on a logical function's truth value.
- The logical function evaluated at a decision is called a predicate.
- That is Predicate is a function which is logically executed during the decision processing.
- The result of this function decides the direction of flow.

Example

- "A is greater than zero," "the fifth character has a numerical value of 31," "X is either negative or equal to 10," "$X + Y = 3Z^2 – 44$," "Flag 21 is set.".
- Every path corresponds to a succession of TRUE/FALSE values for the predicates traversed on that path.
- As an example:
  " 'X is greater than zero' is TRUE."
  AND
  " '$X + Y = 3Z^2 – 44$' is FALSE."
  AND
  " 'W is either negative or equal to 10' is TRUE."

- is a sequence of predicates whose truth values will cause the routine to take a specific path. A predicate associated with a path is called a path predicate**.**

(b) Multiway Branches

- The path taken through a multiway branch such as computed GOTO's (FORTRAN), case statements (Pascal), or jump tables (assembly language) cannot be directly expressed in TRUE/FALSE terms.
- Although it is possible to describe such alternatives by using multivalued logic, an easier expedient is to express multiway branches as an equivalent set of IF . . . THEN . . . ELSE statements.
- For example, a three–way case statement can be written as:
  IF case=1 DO A1 ELSE
  (IF case=2 DO A2 ELSE DO A3 ENDIF) ENDIF
- The translation is not unique because there are many ways to create a tree of IF . . . THEN . . . ELSE statements that simulates the multiway branch.
- We treat multiway branches this way as an analytical convenience in order to talk about testing.
- we don't replace multiway branches with nested IF's just to test them.

(c)Inputs

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- for example, inputs in a calling sequence, objects in a data structure, values left in a register.
- Although inputs may be numerical, set members, boolean, integers, strings, or virtually any combination of object types, we can talk about data as if they are numbers.

(iii) Predicate Expressions

(a) Predicate Interpretation

- Predicate interpretation refers to the process of expressing the predicate in terms of the given input vector by performing various symbolic replacement of operations.
- For example if X_1 and X_2 are inputs, the predicate might be "$X_1 + X_2 > 0$".
- Now let the value of X_2 be given using another predicate as X_2:=Y+5
- The substitution of X_2 value in the first predicate gives you another predicate which is X+Y+5 > 0. This process is known as predicate interpretation.

- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

(b) Independence and Correlation of Variables and Predicates

- The path predicates take on truth values (TRUE/FALSE) based on the values of input variables, either directly (interpretation is not required) or indirectly (interpretation is required).
- If a variable's value does not change as a result of processing, that variable is independent of the processing.
- Conversely, if the variable's value can change as a result of the processing the *variable* is process dependent.
- Similarly, a *predicate* whose truth value can change as a result of the processing is said to be process dependent and one whose truth value does not change as a result of the processing is process independent.
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.
- For example, the input variables are X and Y and the predicate is "X + Y = 10".
- The processing increments X and decrements Y.
- Although the numerical values of X and Y are process dependent, the predicate "X + Y = 10" is process independent.
- Variables, whether process dependent or independent, may be correlated to one another.
- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are uncorrelated.
- By analogy, a pair of predicates whose outcomes depend on one or more variables in common (whether or not those variables are correlated) are said to be correlated predicates.

(c) Path Predicate Expressions

- Path predicate expressions are the collection of expressions that must be fulfilled in order to achieve the desired path.
- This collection of expressions is satisfied based on input values provided.
- These input values must meet all the expressions. If all the expressions are met then the path is chosen else the path is rejected.
- This is shown by means of an example

  $X_1 = 18$

  $X_2 + 5 X_3 + 2 > 0$

  $X_4 - X_2 >= 10 X_3$

  Let the input values of X_2, X_3, X_4 be 2,1,12 respectively.

  Substituting the values in above predicates, we get

  $X_1 = 18$

  $X_2 + 5 X_3 + 2 = 2 + 5*1 + 2 = 9 > 0$

  $X_4 - X_2 >= 10 X_3$ i.e. 12-2 >=10(1) i.e. 10 >=10
- All the conditions appear to be correct as per the values so this path can be chosen.

(iv) Predicate Coverage

(a) Compound Predicates

- Most programming languages permit compound predicates at decisions—that is, predicates of the form A .OR. B or A .AND. B. and more complicated boolean expressions.
- The branch taken at such decisions is determined by the truth value of the entire boolean expression.
- Simply the compound predicate is the combination of two predicates.

- Even if a given decision's predicate is not compound, it may become compound after interpretation because interpretation may require us to carry forward a compound term.

(b) Predicate Coverage

- Predicate coverage is the process of testing all the truth values related to a specific path in all the possible ways.
- If all the values are tested in all possible directions then we can say that 100% predicate coverage is achieved which needs lots of efforts.
- Predicate coverage is slightly comparable to path coverage and is much powerful than the branch coverage.
- If we are using a compound predicate then predicate coverage involves testing of both the predicates in any order.

(v) Testing Blindness

(a) The Problem

- Blindness is a situation which results in the correct path via wrong route unintentionally.
- Testing blindness is a pathological situation in which the desired path is achieved for the wrong reason.
- It can occur because of the interaction of two or more statements that makes the buggy predicate "work" despite its bug and because of an unfortunate selection of input values that does not reveal the situation.
- There are three kinds of predicate blindness: assignment blindness, equality blindness, and self–blindness

(b) Assignment Blindness

- Assignment blindness comes into consideration when both the predicates irrespective of their correctness are satisfied by a value assigned to the assignment statement.
- Assignment blindness may also lead to wrong path selection.

Correct	***Buggy* (Incorrect)**
X := 7	X := 7
.....	.....
IF Y > 0 THEN	IF X + Y > 0 THEN

- If the test case sets Y := 1 the desired path is taken in either case, but there is still a bug.
- Some other path that leads to the same predicate could have a different assignment value for X, so the wrong path would be taken because of the error in the predicate.

(c)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.

Correct	***Buggy***
IF Y = 2 THEN. . .	IF Y = 2 THEN. . .
.....	.....
IF X + Y > 3 THEN. . .	IF X > 1 THEN. . .

- The first predicate (IF Y = 2) forces the rest of the path, so that for any positive value of X, the path taken at the second predicate will be the same for the correct and buggy versions.

(d) Self–Blindness

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

Correct	*Buggy*
X := A	X := A
.....	.....
IF X – 1 > 0 THEN...	IF X + A – 2 > 0 THEN

- The assignment (X := A) makes the predicates multiples of each other (for example, A – 1 > 0 and 2A – 2 > 0), so the direction taken is the same for the correct and buggy version.

(3) Path Sensitizing:

(i) Review :Achievable and Unachievable Paths.

- In order to accomplish test completeness (i.e. C_1 or C_2) for sufficient paths the procedure is as follows.
  1. Extract the programs control flowgraph and select a set of tentative covering paths.
  2. After path selection, determine the predicates for all paths that exist in the selected path set. This makes the basic nature of each predicate compound.
  3. In order to achieve a Boolean expression, the path is traced by multiplying the individual compound predicates. For instance, let the compound predicate be
     (A+BC)(D+E)(FGH)(IJ)(K)(L)
     where the terms in the parentheses are the compound predicates met at each decision along the path and each letter (A,B,…) stands for simple predicates.
  4. The Boolean expression is converted into SOP (Sum of Products) format by multiplying the terms in the given expression as follows
     ADFGHIJKL + AEFGHIJKL + BCDFGHIJKL + BCEFGHIJKL
- Path predicate expressions are the collection of expressions that must be fulfilled in order to achieve the desired path.
- If all the expressions are met then the path is achievable else the path is not achievable.
- The act of finding a set of solutions to the path predicate expression is called path sensitization.

(ii) Pragmatic Observations

- The purpose of the above discussion has been to explore the sensitization issues and to provide insight into tools that help us sensitize paths.
- If in practice you really had to do the above in the manner indicated then test design would be a difficult procedure suitable only to the mathematically inclined.
- It doesn't go that way in practice: it's much easier

(iii) Heuristic Procedures for Sensitizing Paths

- Heuristic procedures are the most optimistic ways for sensitizing paths.
- The first preference for selecting a path must be given to the paths which can be easily sensitized there by delaying the paths whose solution to the path predicate expression is difficult to obtain.
- This convention is followed just for the sake of coverage. Heuristic procedures for path sensitization involve discovery and problem solving using past experience and reasoning.

1. All the process dependent process independent and correlated input variables are first determined and classified accordingly. Show the type of relation that is (logical, arithmetic, functional) and dependency by means of equations for the correlated and dependent variables respectively.

2. After classifying the variables, determine and classify the predicates depending on the input variables into dependent, independent or correlated predicates and also show the type of relation that exists among them.

3. Consider the uncorrelated and independent predicates for selection or path. During the selection, if you have found any dependent predicate, then there may be a classification error or there might be a bug or complete path coverage is not yet achieved.

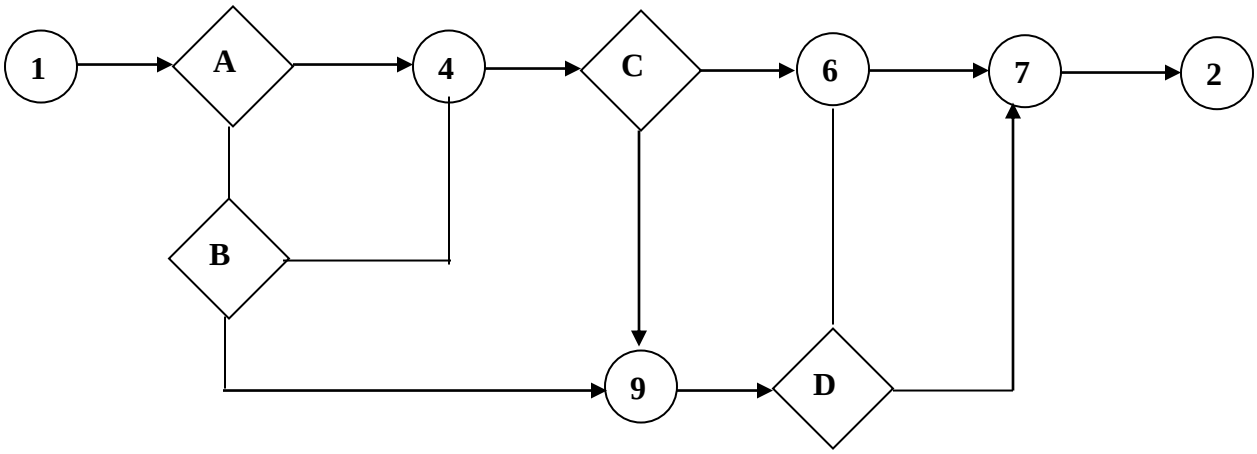
4. Now, consider the correlated and independent predicates if they are not covered then start considering the dependent and uncorrelated, predicates. If the complete coverage is not yet accomplished then move on to the last selection i.e. consider correlated, dependent variables.

5. Display all the input variables, its values, relationship among the variables, type of links for all independent, dependent and correlated variables respectively of every selected path.

6. Every path will produce some set of inequalities, which must be met in order to select that path.

(iv) Examples

(a) Simple, Independent, Uncorrelated Predicates



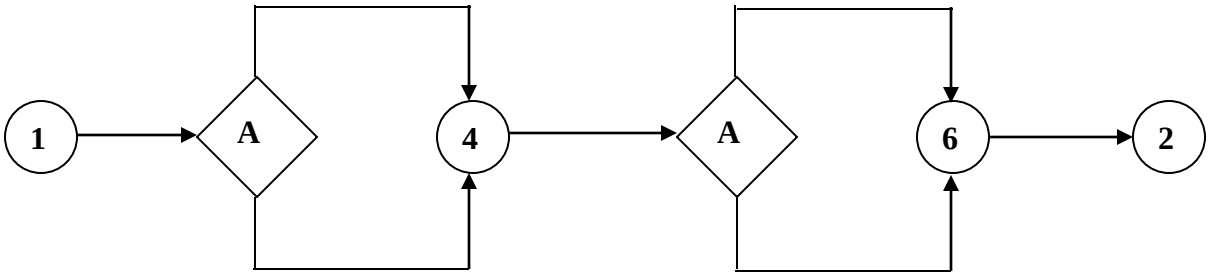
- Consider the independent, uncorrelated predicates.
- The uppercase letters in the decision boxes of the above figure represent the predicates.
- There are four decisions in this example and, consequently, four predicates.
- False predicates are denoted by a bar on the variable. True predicates are represented by the variables without any bar over them.
- From the above figure, we can retrieve the entire covering path and the predicate values which can be represented as follows.

Path	Predicate values
abcdef	AC
aghcimkf	ABCD
aglmjef	ABD

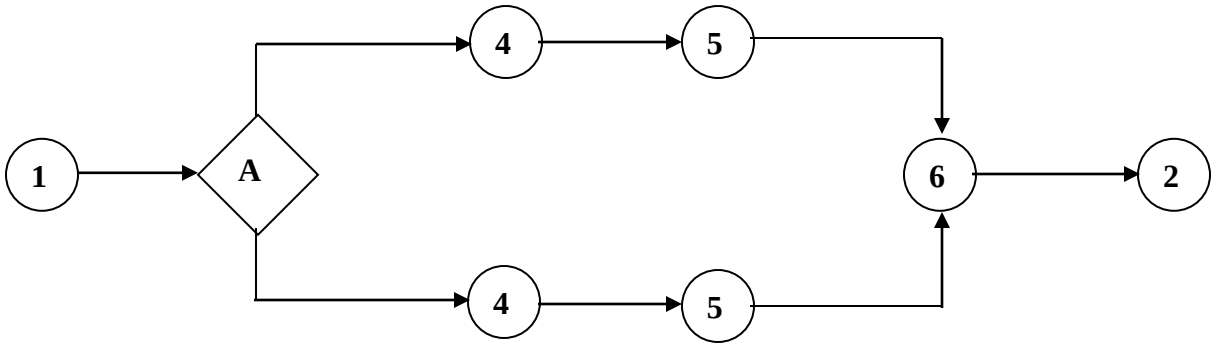
- Using a few more but simpler paths with fewer changes to cover the same flowgraph is

Path	Predicate values
abcdef	AC
abcimjef	ACD
abcimkf	ACD
aghcdef	ABC
aglmkf	ABD

(b) Correlated, Independent Predicates



- The two decisions in the above figure are correlated because they used the identical predicate (A).
- If you picked paths *abdeg* and *acdfg,* which seem to provide coverage, you would find that neither of these paths is achievable.
- If the A branch (*c*) is taken at the first decision, then the A branch (*e*) must also be taken at the second decision.
- There are two decisions and therefore a potential for four paths, but only two of them, *abdfg* and *acdeg,* are achievable.



- The flowgraph can be replaced with the above figure, in which we have reproduced the common code, or alternatively, we can embed the common link *d* code into a subroutine.

(c) Dependent Predicates

- Finding sensitizing values for dependent predicates may force you to "play computer."
- Usually, and thankfully, most of the routine's processing does not affect the control flow and consequently can be ignored.
- Simulate the computer only to the extent necessary to force paths.
- Loops are the most common kind of dependent predicates; the number of times a typical routine will iterate in the loop is usually determinable in a straightforward manner from the input variables' values.
- Consequently it is usually easy to work backward to determine the input value that will force the loop a specified number of times

(d) The General Case

- There is no simple procedure for the general case. It is easy to state the steps involved but much harder to accomplish them.

1. Select cases to provide coverage on the basis of functionally sensible paths. If the routine is well structured, you should be able to force most of the paths without deep analysis. Intractable paths should be examined for potential bugs before investing time solving equations or whatever you might have to do to find path–forcing input values.

2. Tackle the path with the fewest decisions first. Give preference to non looping paths over looping paths.

3. Start at the end of the path and not the beginning. Trace the path in reverse and list the predicates in the order in which they appear. The first predicate (the last on the path in the normal direction) imposes restrictions on subsequent predicates (previous when reckoned in the normal path direction). Determine the broadest possible range of values for the predicate that will satisfy the desired path direction.

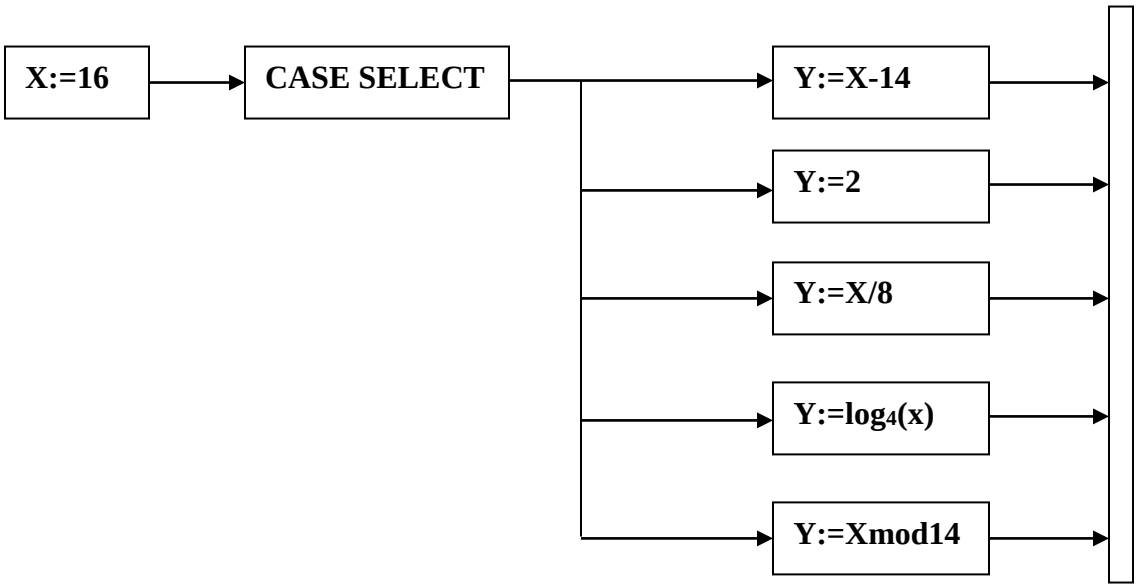
4. Continue working backward along the path to the next decision. The next decision may be restricted by the range of values you determined for the previous decision (in the backward direction). Pick a range of values for the affected variables as broad as possible for the desired direction and consistent with the set of values thus far determined.

5. Continue until you reach the entrance and therefore have established a set of input conditions for the entire path.

(4) Path Instrumentation:

(i) Coincidental Correctness:

- Coincidental Correctness is described as follows.



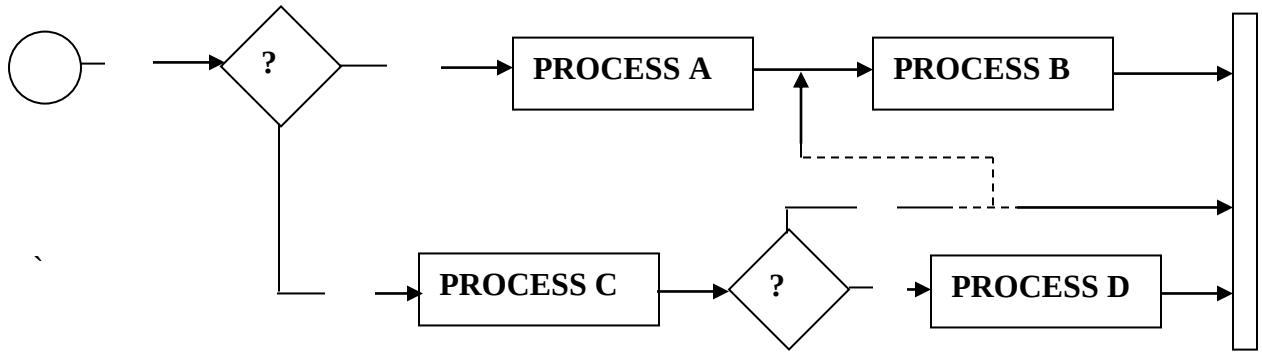
- Since the test outcome is considered as a part of design process, the test is made to run for comparing the actual outcome with the desired outcome.
- Even if the desired outcome is equal to the actual outcome, only some of the conditions are satisfied by the test which are not sufficient enough.
- This type of condition is named as coincidental correctness.
- Simply it can be defined as a condition in which we check whether the expected outcome of a test is generated truly.
- For instance, the coincidental correctness is represented as follows.
- Let us consider an input variable X with an initial value 16 (X=16) which produces a single outcome Y with a value 2 (Y=2) no matter which case we select.
- Therefore the tests chosen this way will not tell us whether we have achieved coverage.
- For example, the five cases could be totally jumbled and still the outcome would be the same.
- Path instrumentation is what we have to do to confirm that the outcome was achieved by the independent path.

(ii) Path Instrumentation.

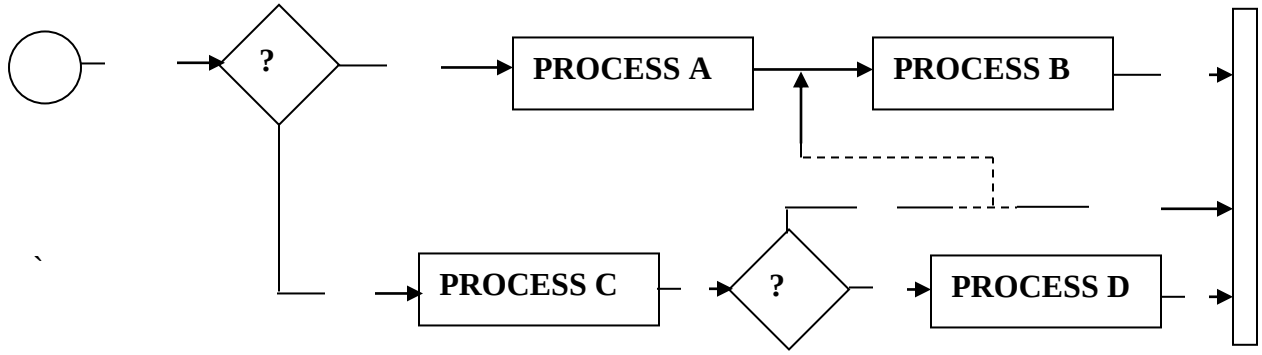
- Path instrumentation is a technique used for identifying whether the outcome of a test is achieved through the desired path or a wrong path.
- Path instrumentation technique is another form of interpretive trace program, which will run each and every statement sequentially there by storing all labels and values of the statements covered for.
- The trouble with traces is that they give us far more information than we need, which is of no use.
- To overcome this drawback many different instrumentation methods have evolved.

(iii) Link Markers

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lowercase letter. Whenever a link is passed, it's name is recorded in the marker.
- The concatenation of the names of all the links starting from an entry to an exit gives the path name.
- The single link marker may not serve the purpose, because there is every possibility of bug which may result in a new link in the middle of the link being traversed.



- We intended to traverse the ikm path, but because of a GOTO in the middle of the m link, we go to process B.
- If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.
- The solution is to implement two markers per link: one at the beginning of each link and one at the end.
- The two link markers now specify the path name and confirm both the beginning and end of the link.
- The double link markers are shown in the following figure.



(iv) Link Counters

- Link counter is one of the instrumentation techniques which usually based on the concept of counters.
- This method provides comparatively less information than interpretive trace method.
- Link counter method of instrumentation follows same procedure as that of link marker but make use of counters instead of using labels for each link which has executed.
- Counters in this method goes on increasing with respect to each link traversed.
- Single counter may not serve the purpose so, we move little deeper and introduce a separate counter for every link.
- With this in practice, we can cross check the total link count against the expected path length.
- This format is not reliable because there is every possibility of having a bug, which may result in a new link in the middle of the link being traveled.
- The same problem that led us to double link markers also leads us to double link counters.

(iv) Other Instrumentation Methods.

- The methods you can use to instrument paths are limited only by your imagination. Here's a sample:

1. Mark each link by a unique prime number and multiply the link name into a central register. The path name is a unique number and you can recapture the links traversed by factoring.

2. Use a bit map with a single bit per link and set that bit when the link is traversed.

3. Use a hash coding scheme over the link names, or calculate an error–detecting code over the link names, such as a check sum.

4. Use your symbolic debugger or trace to give you a trace only of subroutine calls and return.

5. Set a variable value at the beginning of the link to a unique number for that link and use an assertion statement at the end of the link to confirm that you're still on it.

- Every instrumentation probe (marker, counter) you insert gives you more information, but with each probe the information is further removed from reality.

(vi) Implementation

- For unit testing, path instrumentation and verification can be provided by a comprehensive test tool that supports your source language.
- It is easiest to install probes when programming in languages that support conditional assembly or conditional compilation.
- The probes are written in the source code and tagged into categories. Both counters and traversal markers can be implemented, and one need not be parsimonious with the number and placement of probes because only those that are activated for that test will be compiled or assembled.
- For any test or small set of tests, only some of the probes will be active. Rarely would you compile with all probes activated and then only when all else failed.

(5) Implementation and Application of path testing:

- Path testing is a process which involves all the available paths in a program from an entry to an exit in such a way that the entire path is thoroughly tested.
- Path testing implementation and application can be categorized as follows.

(i) Integration, Coverage, and Paths in Called Components

- Path–testing methods are mainly used in unit testing, especially for new software.
- Classical unit testing mainly involves the use of stubs for replacement of all called components and corequisite components thereby testing the new component individually.

- Path testing process which is carried out at this phase is to analyze the control flow errors rather than focusing on bugs in called or corequisite components.
- We then integrate the component with its called subroutines and corequisite components, one at a time, carefully probing the interface issues.
- Once the interfaces have been tested, we retest the integrated component, this time with the stubs replaced by the real subroutines and corequisite component.
- The component is now ready for the next level of integration. This bottom–up integration process continues until the entire system has been integrated.
- Coverage issue arises since, subroutines and corequisite components are considered to be a part of the component and hence, increasing the complexity as large code need to be processed which makes path sensitization much difficult.
- The main intention behind path testing is that, testing each level at any time increases the effectiveness of the test but the drawback associated with this approach is that it results in i.e. predicate coverage and blindness i.e. outcome of one level may not be compatible with the outcome of other consecutive levels.

(ii) New Code

- The new code (components) has to be given higher priority for testing than the old trusted components.
- Stubs are used where it is clear that the bug potential for the stub is significantly lower than that of the called component.
- That means that old, trusted components will not be replaced by stubs.
- Some consideration is given to paths within called components, but only to the extent that we have to do so to assure that the paths we select at the higher level is achievable.
- Paths within the low level components are also tested, so that there should not be any un-achievable path at higher level.
- Typically, we'll try to use the shortest entry/exit path that will do the job; avoid loops; avoid lower–level subroutine calls; avoid as much lower–level complexity as possible.
- Unit testing must be automated in such a way, that it must perform the testing at each level of integration.

(iii) Maintenance

- The maintenance situation is distinctly different.
- Path testing will be carried out on the modified components but called and corequisite components will be kept unchanged.
- If we have a configuration–controlled, automated, unit test suite, then path testing will be repeated entirely with such modifications as required to accommodate the changes.
- Otherwise, selected paths will be chosen in an attempt to achieve C2 over the changed code.
- As the maintenance methods are studied further a new methodology will be discovered, which will help us to achieve the desired coverage.

(iv) Rehosting

- Rehosting is a process of transforming the old software environment into a new more friendly environment in which rehosted software can run cost effectively.
- When used in conjunction with automatic or semiautomatic structural test generators, we get a very powerful, effective, rehosting process.
- The objective of rehosting is to change the operating environment and not the rehosted software.
- You cannot rehost the software, while performing changes in its environment i.e., the two things cannot be done simultaneously.
- Rehosting can be done in the following ways.

- First, a translator from the old to the new environment is created and tested as any piece of software would be. The bugs in the rehosting process, if any, will be in the translation algorithm and the translator, and the rehosting process is intended to catch those bugs .
- Second, a complete (C1 + C2) path test suite is created for the old software in the old environment.
- Components may be grouped to reduce total testing labor and to avoid a total buildup and reintegration, but C1 + C2 is not compromised.
- The suite is run on the old software in the old environment and all outcomes are recorded.
- These outcomes serve as a guideline for rehosted software. The outcomes and test cases are adapted by the new environment with the help of another interpreter.
- These adapted environment and software are integrated and retested.
- This approach might be even more costly than building the new software, but it provides us with an environment which suites the requirements of software there by providing stable and reliable software base without bothering about the issues pertaining to software security.

UNIT –II
TRANSACTION FLOW TESTING

(1) Transaction Flows:

(i) Definitions:

- A transaction is defined as a set of statements or a unit of work handled by a system user.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons, or devices that are outside of the system.
- Each transaction is usually associated with an entry point and an exit point.
- The execution of a transaction begins at the entry point and ends at an exit point there by producing some results.
- After getting executed, the transaction no longer exists in the system.
- All the results are finally stored in the form of records inside the system.

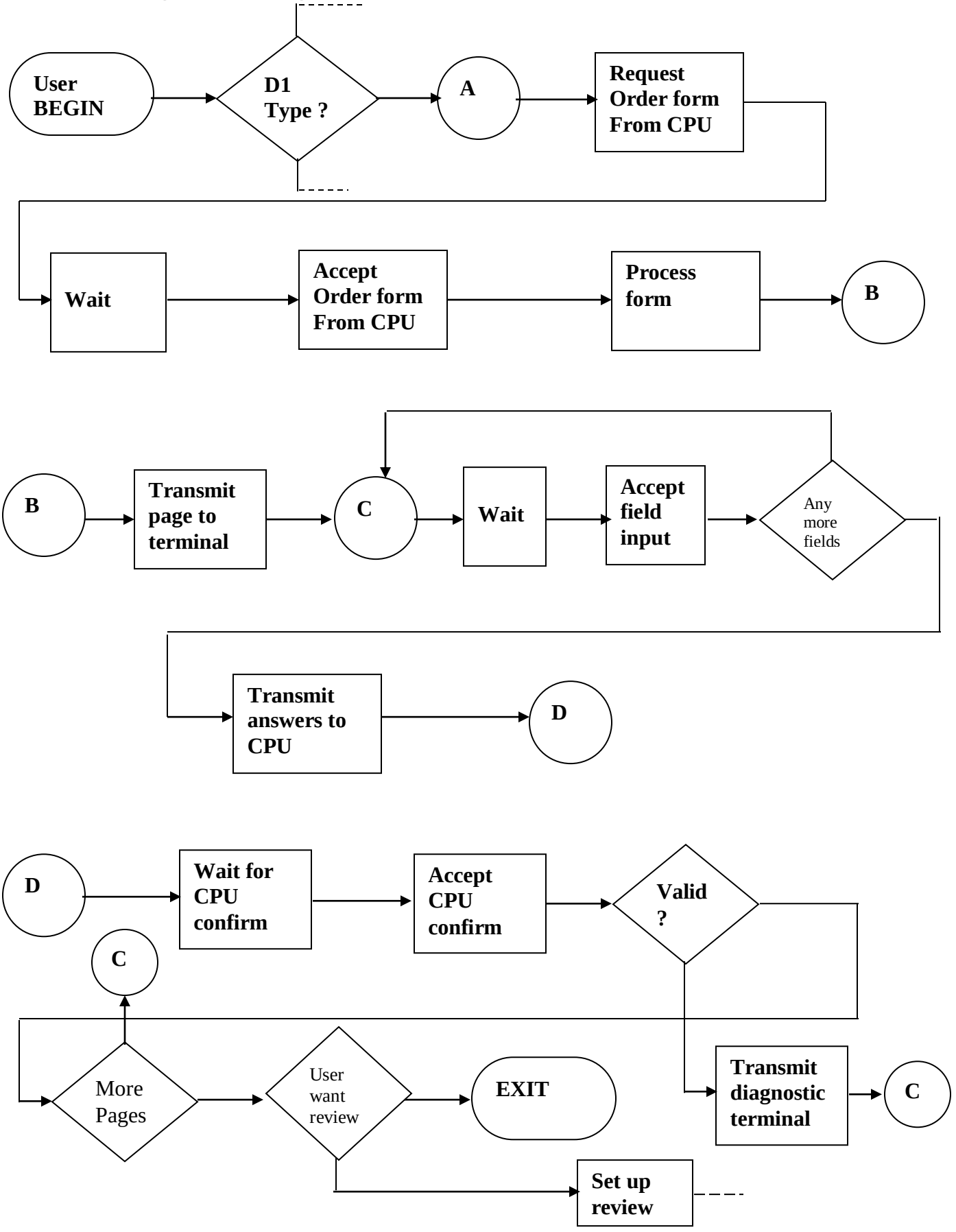
A transaction for an online information retrieval system might consist of the following steps:

1. Accept input (tentative birth).
2. Validate input (birth).
3. Transmit acknowledgment to requester.
4. Do input processing.
5. Search file.
6. Request directions from user.
7. Accept input.
8. Validate input.
9. Process request.
10. Update file.
11. Transmit output.
12. Record transaction in log and cleanup (death).

- The user processes these steps as a single transaction.
- From the system's point of view, the transaction consists of twelve steps and ten different kinds of subsidiary tasks.
- Most online systems process many kinds of transactions.
- For example, an automatic bank teller machine can be used for withdrawals, deposits, bill payments, and money transfers.
- Furthermore, these operations can be done for a checking account, savings account, vacation account, Christmas club, and so on.
- Although the sequence of operations may differ from transaction to transaction, most transactions have common operations.
- For example, the automatic teller machine begins every transaction by validating the user's card and password number.
- Tasks in a transaction flowgraph correspond to processing steps in a control flowgraph.
- As with control flows, there can be conditional and unconditional branches, and junctions.

(ii) Example:

- The following figure shows part of a transaction flow.
- A transaction flow is processed in Forms. Each form consists of several pages with records and fields in it.
- A system is taken as the terminal controller to process these form. Only those forms which are located on a central computer are requested for processing.


User BEGIN
D1 Type ?
A
Request Order form From CPU
Wait
Accept Order form From CPU
Process form
B
B
Transmit page to terminal
C
Wait
Accept field input
Any more fields
Transmit answers to CPU
D
D
Wait for CPU confirm
Accept CPU confirm
Valid ?
C
More Pages
User want review
EXIT
Transmit diagnostic terminal
C
Set up review

- Long forms are compressed and transmitted by the central computer to minimize the number of records in it.
- The output of each page is transmitted by the terminal controller to the central computer.
- If the output is invalid, the central computer transmits a code to the terminal controller.
- The terminal controller in tern transmits the code to the user to check the input. At the end the user reviews the filled out form.
- The above figure shows the processing of a transaction using forms.
  - When the transaction is to be initiated, the process p_1 requests forms from CPU.
  - The central computer accepts the form in the process p_3. p_4 process the form.
  - The characteristics of the transactions are shown by using a decision box D1 to determine whether to cancel or process further.
  - These decisions are handled by the terminal controller.
  - P_5 transmits the page to the terminal.
  - D_2 and D_4 are the decision boxes to know whether the form needs more pages or not.
  - D_3 is a decision for the structure of the form, to validate the input.
  - If necessary, the user reviews whole system in process p_{12}
  - The central computer then transmits a diagnostic code back to the terminal controller in p_{11}. After reviewing, the transaction flow is closed and exit operation is performed.

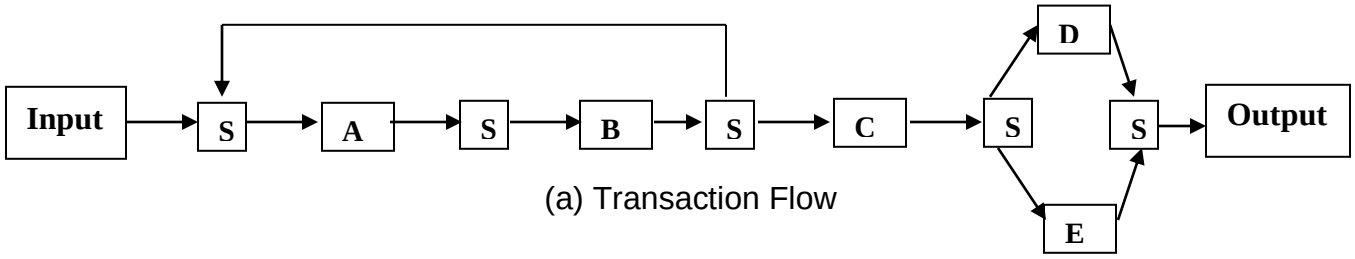
(iii) Usage:

- Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
- A big system such as an air traffic control or airline reservation system has not hundreds, but thousands of different transaction flows.
- The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.
- An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.

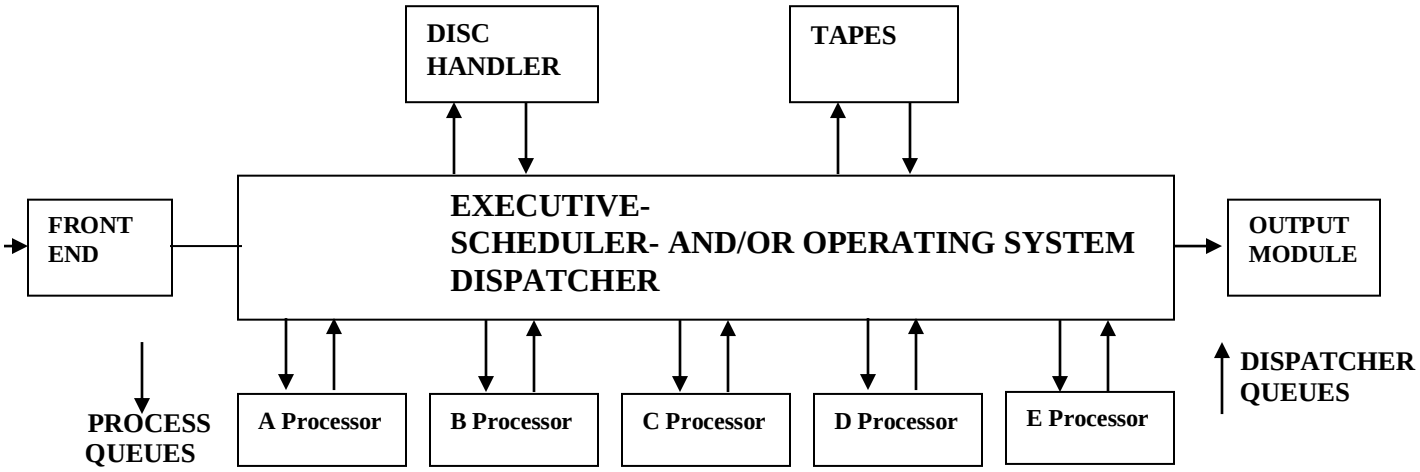
(iv) Implementation:

- Transaction flow has an implicit representation of system control structure.
- That is, there is no direct relation between the process and decisions.
- A transaction flow is represented by a path taken by a transaction through a succession of processing modules. These transactions are placed in a transaction-control block.
- The transactions present in that block are processed according to their flow.
- Each transaction is represented by a token and the transaction flowgraph shows a pictorial representation of these tokens.
- The transaction flowgraph is not the control structure of the program.
  - The below **figure a** shows transaction flow and corresponding implementation of a program that creates that flow.
  - This transaction goes through input processing, and then passes through process A, followed by B.
  - The result of process B may force the transaction to pass back to process A.
  - The transaction then goes to process C, then to either D or E, and finally to output processing.
  - **Figure b** is a diagrammatic representation of system control structure.
  - This system control structure is controlled either by an executive or scheduler or dispatcher operating system.
  - The links in the structure either represents a process queue or a dispatcher queue.
  - The transaction is created by placing a token on an input queue.

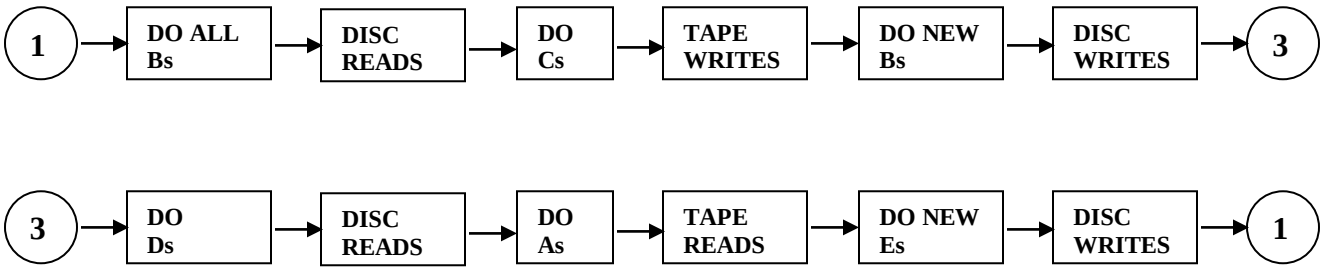
- The scheduler then examines the transaction and places it on the work queue for process A, but process A will not necessarily be activated immediately.
- When a process has finished working on the transaction, it places the transaction-control block back on a scheduler queue.
- The scheduler then examines the transaction control block and routes it to the next process based on information stored in the block.
- The scheduler contains tables or code that routes the transaction to its next process. In systems that handle hundreds of transaction types, this information is usually stored in tables rather than as explicit code.
- Alternatively, the dispatcher may contain no transaction control data or code; the information could be implemented as code in each transaction processing module.



(a) Transaction Flow



(b) System Control Structure



(c) Executive/Dispatcher Flowchart

- Figure c shows a simplified representation of transaction flow.
- Let's say that while there could be many different transaction flows in the system, they all used only processes A, B, C, D, E, and disc and tape reads and writes, in various combinations.
- Just because the transaction flow order is A,B,C,D,E is no reason to invoke the processes in that order.
- For other transactions, not shown, the processing order might be B,C,A,E,D. A fixed processing order based on one transaction flow might not be optimum for another.
- Furthermore, different transactions have different priorities that may require some to wait for higher-priority transactions to be processed.
- Similarly, one would not delay processing for all transactions while waiting for a specific transaction to complete a necessary disc read operation.

(v) Perspective:

- There were no restrictions on how a transaction's identity is maintained: implicit, explicit, in transaction control blocks, or in task tables.
- Transaction-flow testing is the ultimate black-box technique because all we ask is that there be something identifiable as a transaction and that the system will do predictable things to transactions.
- Transaction flowgraphs are a kind of data flowgraph.
- Data flowgraphs and control flowgraphs the most important difference is in control flowgraphs we defined a link or block as a set of instructions such that if any one of them was executed, all (barring bugs) would be executed.
- For data flowgraphs in general, and transaction flowgraphs in particular, we change the definition to identify all processes of interest.
- Another difference to which we must be sensitive is that the decision nodes of a transaction flowgraph can be complicated processes in their own rights.

(vi) Complications:

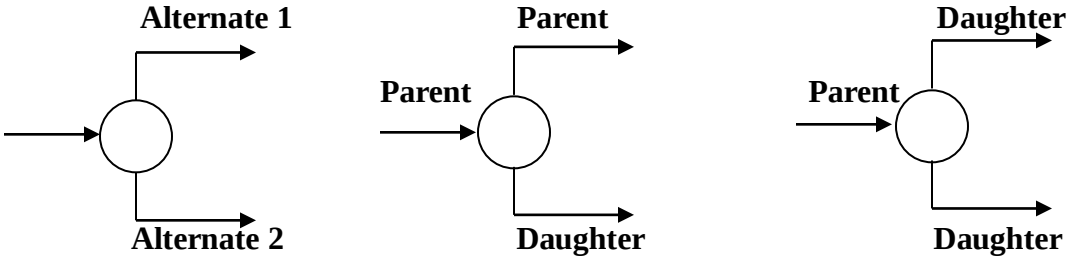
(a) General

- Transaction flows don't have a good structured design for code.
- The problems of transaction flows result in problems like error conditions, malfunctions, recovery actions etc.
- These errors are unstructured. As features are added into the transaction flows the complexity of the transaction flow increases.
- Transactions are interactions between modules. A good system design indicates that there is no implementation of new transaction or changing of an existing transaction.
- Hence transaction flow model results in consequences such as poor response times, security problems, inefficient processing, dangerous processing etc.
- The decision nodes of a transaction flowgraph can be complicated.
- These nodes have exists that go to central recovery processes.
- The effect of interrupts in a transaction flow model converts every process box into many, with exit links.
- Therefore the test design is no longer fit for transaction flow model.
- Examples for the transaction flow to be imperfect.

(b) Births

- A transaction can give birth to others and can also merge with others in many of the systems. From the time they are created to the time they are completed, transaction flows have a unique identity.

- The following figure shows three different possible interpretations of the decision nodes with two or more outlinks.

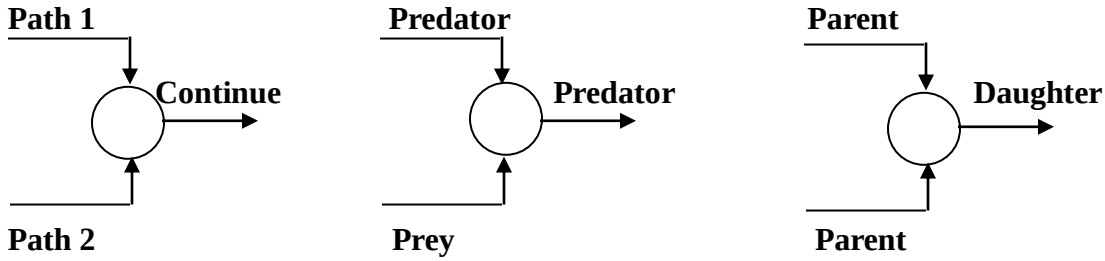


(a) Decision (b) Biosis (c) Mitosis

- In **figure a,** a transaction (Birth) has been created. The incoming transaction at decision node gives birth of two new transactions.
- The two transactions alternate 1 and alternate 2 has a different or same identity.
- The **figure b** shows a different situation compared to **figure a**.
- The parent transaction gives birth to two new transactions.
- One transaction has the same identity as Parent the other transaction results in a different identity Daughter. This situation is called Biosis.
- The figure c is similar to figure b, except that the parent transaction is destroyed and two new transactions (daughters) are created. This situation is called mitosis.

(c) Mergers

- Merging is as troublesome as transaction flow splitting. The two transactions are merged at decision node giving a new transaction with the same or different identity.



(a) Junction (b) Absorption (c) Conjunction

- In **figure a** path 1 and path 2 merge at a junction resulting in a single one Continue.
- The **figure b** is a predator transaction absorbs a prey. The prey is gone but the predator retains its identity.
- The **figure c** shows a slightly different situation in which two parent transactions merge to form a new daughter.

(d) Theoretical Status and Pragmatic Solutions (Solutions for the above examples)

- Transaction flow model doesn't meet the requirements of multiprocessor system. Therefore a generic model called Petri is taken.
- Petri nets use operations that include explicit representation of tokens in the stages of process.
- Petri net have been used to test the problems in protocol testing, network testing and so on. The application to software testing is still in its beginning stage to determine whether it is a productive model or not.

- As long as test results are good, the imperfect model doesn't not matter because the complexities that can invalidate the model have been ignored.
- The following are some of the possible cases:
  1. ***Biosis***
     - The parent flow is followed from beginning of a transaction flow to the end of a transaction flow.
     - A new birth is treated as a new flow, either to end or to absorb that birth.
  2. ***Mitosis***
     - It begins from the parent's flow to the mitosis point. From mitosis point, an additional flow starts and get destroyed at their respective ends.
  3. ***Absorption***
     - In this situation, the parent's flow is treated as the primary flow. The parent flow is modeled from its absorption point to the point at which it gets destroyed.
  4. ***Conjugation***
     - This situation is the opposite of mitosis situation. Each parent flow is modeled from its birth to the conjugation point.
     - And from the conjugation point, the resulting child flow starts and get destroyed.
- Births, Mitosis, Absorptions, and conjugations are as problematic for the software designers.
- Illegal births, wrongful deaths and lost children are some of the common problems.
- Although the transaction flow is modeled by simple flowgraphs, they recognize bugs where transactions are created, absorbed and conjugated.

(vii) Transaction flow structure:

- A sequential flow of operations is represented by a structure called a transaction flow structure.
- Even transaction flows are analogous to control flowgraphs, it is not necessary that good structure provided for code should also exist for transaction flows.
- Transactions flows are often considered as ill-structured due to the following reasons.
  1. It's a *model* of a process, not just code. While processing the transaction, humans can't be forced to follow the rules of a specific software structure, as they may incorporate decisions, loops, etc
  2. Behavior of other uncontrolled systems may be incorporated by some parts of the transactional flow.
  3. Permanent ill-structured nature of the transaction flow leads to loop jumps uncontrollable GOTO statements etc. Not even a small part of the transaction flow has the ability to handle error detection, failures, malfunctioning, recovery actions etc
  4. If any new features are added and enhancements are made in transactional flows, then the complexity of each and every transaction inherently increases. For instance one can't expect a good transaction flow from lawyers, politicians, salesman etc
  5. Basically systems are designed from specific modules and the transaction flows are designed or produced through the module of interaction..
  6. Modeling of interrupts, multitasking, synchronization, polling, queue disciplines are not related to structuring..

(2) Transaction Flow Testing Techniques:

(i) Get the Transaction Flows:

- Complicated systems that process a lot of different complicated transactions should have explicit representations of the transaction flows, or the equivalent documented.

- The transaction flows can be mapped into programs such that the flow of transaction will be created easily.
- The processing of the transactions is done in the design phase.
- The overview section in design phase contains the details of the transaction flows.
- Detailed transaction flows are necessary to design the system's functional test.
- Transaction flows are similar to control flow graphs where the act of getting information can be more effective.
- Therefore the bugs can be determined. The flow of transaction in design phase is done step by step such that the problems would not arise and a bad design can be avoided.

(ii) Transaction Flow testing:

- Transaction flow testing is a technique used in computerized applications.
- The transaction flow testing technique is used to control the documents that require the auditor to specify the following.
  - The business cycle in the flow.
  - The various types of transaction that flow through individual cycle.
  - The operations that are carried out within the cycle.
  - The objectives of internal control
  - The internal control methods used to attain each objective.
- The tester in the transaction flow testing is used to develop a flowchart. The tester tracks the transaction flow and performs various functions in the same order as that of the transaction.
- The internal control methods are recognized at each point of the transaction flow.

(iii) Inspections, Reviews, Walkthroughs:

- Transaction flows are a natural agenda for system reviews or inspections.
- Start transaction-flow walkthroughs at the preliminary design review and continue them in ever greater detail as the project progresses.

1. In conducting the walkthroughs, you should:

a. Discuss enough transaction types (i.e., paths through the transaction flows) to account for 98%–99% of the transactions the system is expected to process.

b. Discuss paths through flows in functional rather than technical terms.

c. Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.

2. Make transaction-flow testing the cornerstone of system functional testing just as path testing is the cornerstone of unit testing. For this you need enough tests to achieve C_1 and C_2 coverage of the complete set of transaction flowgraphs.

3. Select additional transaction-flow paths (beyond $C_1 + C_2$) for loops, extreme values, and domain boundaries.

4. Select additional paths for weird cases and very long, potentially troublesome transactions with high risks and potential consequential damage.

5. Design more test cases to validate all births and deaths and to search for lost daughters, illegitimate births, and wrongful deaths.

6. Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

7. Have the buyer concur that the selected set of test paths through the transaction flows constitute an adequate system functional test.

8. Tell the designers which paths will be used for testing but not (yet) the details of the test cases that force those paths.

(iii) Path Selection:

- Path selection for system testing based on transaction flows should have a distinctly different flavor from that of path selection done for unit tests based on control flowgraphs.
- Start with a covering set of tests ($C_1 + C_2$) using the analogous criteria you used for structural path testing, but don't expect to find too many bugs on such paths.
- Select a covering set of paths based on functionally sensible transactions as you would for control flowgraphs.
- Confirm these with the designers.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow. Create a catalog of these weird paths.
- This procedure is best done early in the game, while the system design is still in progress, before processing modules have been coded. The covering set of paths belongs in the system feature tests.
- It gives everybody more confidence in the system and its test.

(iv) Sensitization:

- The Good news is most of the normal paths are very easy to sensitize—80%–95% *transaction flow* coverage (C1 + C2) is usually easy to achieve.
- The bad news is that the remaining small percentage is often very difficult, if not impossible, to achieve by fair means.
- While the simple paths are easy to sensitize there are many of them, so that there's a lot of tedium in test design.
- Sensitization *is* the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.
- The reason these paths are often difficult to sensitize is that they correspond to error conditions, synchronization problems, overload responses, and other anomalous situations.

1. *Use Patches*

- The dirty system tester's best, but dangerous, friend.
- It's a lot easier to fake an error return from another system by a judicious patch than it is to negotiate a joint test session.

2. *Mistune*

- Test in a system sized with grossly inadequate resources.
- By "grossly" I mean about 5%–10% of what one might expect to need.
- This helps to force most of the resource-related exception conditions.

3. *Break the Rules*

- Transactions almost always require associated, correctly specified, data structures to support them.
- Often a system database generator is used to create such objects and to assure that all required objects have been correctly specified.
- Bypass the database generator and/or use patches to break any and all rules embodied in the database and system configuration that will help you to go down the desired path.

4.*Use Breakpoints*

- Put breakpoints at the branch points where the hard-to-sensitize path segment begins and then patch the transaction control block to force that path.

- You can use one or all of the above methods, and to sensitize the strange paths.
- These techniques are especially suitable for those long tortuous paths that avoid the exit.

(v) Instrumentation:

- Instrumentation plays a bigger role in transaction-flow testing than in unit path testing.

- Counters are not useful because the same module could appear in many different flows and the system could be simultaneously processing different transactions.
- The information of the path taken for a given transaction must be kept with that transaction.
- It can be recorded either by a central transaction dispatcher (if there is one) or by the individual processing modules.
- You need a trace of all the processing steps for the transaction, the queues on which it resided, and the entries and exits to and from the dispatcher.
- In some systems such traces are provided by the operating system.
- In other systems, such as communications systems or most secure systems, a running log that contains exactly this information is maintained as part of normal processing.

(vi) Test databases:

- About 30%–40% of the effort of transaction-flow test design is the design and maintenance of the test database(s).
- The first error is to be unaware that there's a test database to be designed.
- The result is that every programmer and tester designs his own, unique database, which is incompatible with all other programmers' and testers' needs.
- The consequence is that every tester (independent or programmer) needs exclusive use of the entire system. Furthermore, many of the tests are configuration-sensitive, so there's no way to port one set of tests over from another suite.

(vii) Execution:

- If you're going to do transaction-flow testing for a system of any size, be committed to test execution automation from the start.
- If more than a few hundred test cases are required to achieve C1 + C2 transaction-flow coverage, don't bother with transaction-flow testing if you don't have the time and resources to almost completely automate all test execution.
- You'll be running and rerunning those transactions not once, but hundreds of times over the project's life.
- Transaction-flow testing with the intention of achieving C1 + C2 usually leads to a big increase in the number of test cases.
- Without execution automation you can't expect to do it right.

DATA FLOW TESTING

(3) Basics of Data-Flow Testing:

(i) Motivation and assumptions:

(a) What is it?

- Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

(b) Motivation

- It is our belief that, just as one would not feet confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.
- To the extent that we achieve the widely sought goal of reusable code, we can expect the balance of source code statements to shift ever more toward data statement domination.

- In all known hardware technologies, memory components have been, are, and are expected to be cheaper than processing components.

(c) New Paradigms-Data-Flow Machines

- Data flow machines are programmable computers that use packet switching communication.
- The hardware in data flow machines is optimized for data-driven execution and for fine grain parallelism.
- Data flow machines support recursion. Recursion is a mechanism used to map virtual space to a physical space of realistic size. It is the fastest mechanism.
- The prototype in data flow machines is taken as a processing or working element.
- The overhead in data flow machines can be made acceptable by sophisticated hardware.
- There is a sufficient parallelism in many computer programs.
- The problem in data flow machine is in distribution of computation and storage of data structures.
- Another problem in data flow machines is to cease (stop) parallelism when resources tend to get overloaded.
- Some of the data flow machines are Von Neumann machines and MIMD (multi instruction, multi data) machines.

Von Neumann machines

- The Von Neumann architecture executes one instruction at a time in the following, typical, microinstruction sequence.
  1. Fetch instruction from memory.
  2. Interpret instruction.
  3. Fetch operand(s).
  4. Process (execute).
  5. Store result (perhaps in registers).
  6. Increment program counter (pointer to next instruction).
  7. GOTO 1.
- The pure Von Neumann machine has only one set of control circuitry to interpret the instruction, only one set of registers in which to process the data, and only one execution unit (e.g., arithmetic/logic unit).
- This design leads to a sequential, instruction-by-instruction execution, which in turn leads to control-flow dominance in our thinking.
- The Von Neumann machine forces sequence onto problems that may not inherently be sequential.

MIMD (multi-instruction, multi data) machines

- MIMD machines are massively parallel machines.
- They fetch several instructions in parallel.
- Therefore they have several mechanisms for executing the above steps 1-7.
- MIMD machines can also perform arithmetic or logical operation simultaneously.
- These operations are done on different data objects.
- In these machines parallel computation is left to the compiler for processing instructions.
- For a MIMD machine, the instructions are produced in parallel flow while for a conventional machine the instructions are produced in sequential flow.
- The Parallel machine is MIMD machine with multiple processors and sequential machine is Von Neumann machine with only one processor.

(d) The Bug Assumptions

- 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.

(ii) Data Flowgraphs:

(a) General:

- The data flowgraph is a graph consisting of nodes and directed links (i.e., links with arrows on them). The data flow is between the data objects in the data flowgraph.
- The data flowgraph not only shows the flow of data but also shows the deviation between the data objects to be implemented.

(b) Data Object State and Usage:

- Data objects can be three states i.e. created, killed and used states.
- They can be used in two distinct ways: in a calculation part and in the control flowgraph part. The following symbols denote these possibilities.

  d—defined, created, initialized, etc.
  k—killed, undefined, released.
  u—used for something.
  c—used in a calculation part.
  p—used in a predicate for operation purpose.

- Every symbol in data flowgraph has a meaning. Each symbol is described below.

1. *Defined*:

- An object is defined explicitly when it appears in a data declaration or implicitly when it appears on the left-hand side of an assignment statement.
- "Defined" can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, and so on.

2. *Killed or Undefined*

- When an object is released and is no longer in use, then it is known as a killed object. Killed object is similar to an undefined object.
- An object that is not available in the statement is known as Undefined object.
- For example, a loop in FORTRAN language gets terminated when an undefined variable exists.
- Another example for a killed variable is that, if an object A has been assigned a value such as A:=8 and another assignment is done for the same object A, such as A:=11 then the previous value of A (i.e. 8) is killed and redefined (i.e.11). Therefore the value of A is 11.
- Define and kill are complementary operations. That is, they generally come in pairs and one does the opposite of the other.

3. *Usage*

- A used variable is for computation (c) use and is on the right side of an assignment statement.
- It is also used in a predicate (P) such as if z > 0, to evaluate the flow of control.
- Hence usage variables are used both in predicate and computational purposes.

(c) Data-Flow Anomalies:

- An anomaly is a situation or condition where an object is defined but not used. For example

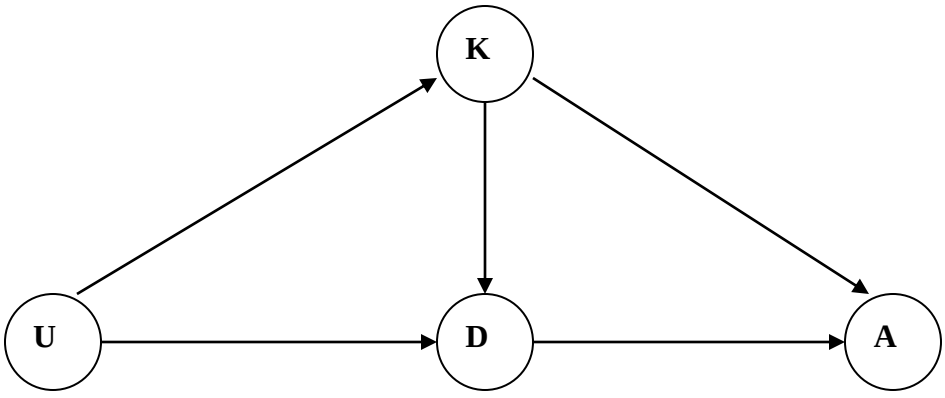
```
IF A>0 THEN X:=1 ELSE X:= -1
```

A:= 0
A:= 0
A:= 0
A:= B + C

- From the above example, we notice that object A is defined trice to zero. Hence an anomaly occurs.
- There are nine possible two-letter combinations for *d, k* and *u*. Some are bugs state, some are suspicious (dangerous) state, and some are normal state.
  dd—It results in a suspicious state where an object is defined twice.
  dk—results in a bug state.
  du—the normal case. The object is defined, then used.
  kd—normal situation. An object is killed, then redefined.
  kk—harmless but probably buggy.
  ku—A bug state.
  ud—suspicious state.
  uk—normal situation.
  uu—normal situation
- The three variables (d,k,u) show the representation of anomalous state.
- In addition to the above two-letter situations there are six single-letter situations
  -k: possibly anomalous.
  –d: okay. This is just the first definition along this path.
  –u: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
  k–: not anomalous. The last thing done on this path was to kill the variable.
  d–: possibly anomalous.
  u–: not anomalous.
- The single-letter situations do not lead to clear data-flow anomalies but only the possibility thereof.

(d) Data-Flow Anomaly State Graph :

- The data flow anomaly defines an object to be in one of the following four different states. The states are
  K—undefined, previously killed, does not exist.
  D—defined but not in use.
  U—has been used for computation or in predicate.
  A—anomalous



- Don't confuse these capital letters (K,D,U,A), which denote the state of the variable, with the program action, denoted by lowercase letters (*k,d,u*).
- The data flow anomaly starts in K state.
- An attempt is made to use an undefined variable. Hence it goes in an anomalous (A) state. The killed (K) state defines a variable d in defined (D) state.
- If a variable is killed from a defined (D) state then it becomes anomalous.
- The variable u is used in U state and is redefined d in D state.
- Variable k get killed in K state.

(e) Static versus Dynamic Anomaly Detection:

- Static analysis is an analysis done at compile time.
- The source code is checked and the quality is improved by removing the bugs in the program.
- Syntax errors are detected in static analysis.
- To improve the quality of a document, the document is analyzed and checked by a tool.
- If a problem, such as a data-flow anomaly, can be detected by static analysis methods, then it does not belong in testing—it belongs in the language processor.
- Static analysis tools are typically used by tools.
- Static analysis is done in design phases so that the whole model can be analyzed and the inconsistencies can be detected.
- Static analysis can be used in the detection of security problem.
- Dynamic analysis is done at run time. Dynamic analysis detects anomalous situations at run time with some of the data structures like Arrays, Pointers, Records etc..

1. *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.

2. *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, to know whether the values are within the boundary range or out of boundary range.

3. *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.
- In the case of records, files are created and the names of such files are dynamically known.
- Without execution there is no way to determine the state of such objects.

4. *Dynamic Subroutine or Function Names in a Call*

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

5. *False Anomalies*

- Anomalies don't occur when the path of objects is not completed.
- Such anomalies are false anomalies. The problem of identifying whether a path is completed or not is not solved.

6. *Recoverable Anomalies and Alternate State Graphs*

- What constitutes an anomaly depends on context, application, and semantics.
- Huang provided two anomaly state graphs

7. Concurrency, Interrupts, System Issues

- Anomalies become more sophisticated while moving from single processor surroundings to multi processors environment.
- The main purpose or task of interrupt is to develop correct anomalous which is even performed in true concurrency or pseudo concurrency.
- The objective of system integration testing is to detect data flow anomalies at run time that was not possible using context level testing.

- 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.

(f) Anomaly detection & types of data flow anomalies:

- An anomaly is a term that leads to inconsistency in the data flow analysis.
- The data flow is referred to as reading variables and data flow anomaly is referred to as reading variables without having an idea that the value of the variable is in use or not.
- During data flow analysis, every variable is referred to and inspected.
- There are different variables in data flow analysis.
- They are classified as

S.No	Variables	Definition
1	Defined (d)	Value assigned to a variable
2	Referenced (r)	Value read or used by a variable
3	Undefined (u)	Variable that has no defined value

- Depending on these variables, three different data flow anomalies are distinguished. They are
  1. ur-anomaly
  2. du-anomaly
  3. dd-anomaly

1. **ur-anomaly:**
   - During data flow analysis if the undefined value of a variable (u) is read (r) then it is known as a ur-anomaly.
2. **du-anomaly:**
   - A defined (d) variable becomes invalid or undefined (u) variable when a variable is not used within a particular time.
3. **dd-anomaly:**
   - This anomaly occurs when the variable accepts a value at the second assignment (d) and the first assignment value had not been used.
   - This situation occurs in dd-anomaly. For example if A:=7,A:=11 then it accepts A:=11.

- Depending on the usage of variables the anomalies can be detected.
- For example consider c++example The example shows an exchange of values of the variables A and B with the help of another variable get if the value of the variable A is greater than the value of the variable B.

```
void exchange(int &A,int &B)
{
```

```
        int get;
        if(A>B)
        {
            B=get;
            B=A;
            get=A;
        }
    }
```

- The detection of anomalies are
  1. **ur-anomaly:**
     - In the above example, the variable get is used on the right side of an assignment.
     - The variable get has an undefined value because it is not initialized where it is declared.
     - This undefined variable is being read or referred to and hence it results in ur-anomaly.
  2. **dd-anomaly:**
     - The variable B is used twice on the left side of an assignment.
     - The first assignment value becomes invalid or unused and the second assignment value is taken or used.
     - Therefore the unused variable B of the first assignment results in dd-anomaly
  3. **du-anomaly:**
     - The variable get has a defined value in the last assignment. The defined variable cannot be used anywhere in the function because only those variables are valid which are inside the function.
     - Therefore the unused variable results in du-anomaly.

(iii) The Data-Flow Model:

(a) General:

- Our data-flow model is based on the program's control flowgraph—don't confuse that with the program's data flowgraph.
- So Data-flow model is considered as the heart of programs control flowgraph.
- It consists of links which are denoted by symbols d,k,u,c,p or a sequence of the symbols like dd, du, ddd etc.
- This sequence specifies the sequential flow of data operations on the link with respect to the given variable.
- These symbols are called link weights as each link is assigned with weights (d,k,u,c,p).
- For all variables and array elements, different set of link weights exist.

  The symbols are defined as

  d= Defined object , k=Killed object, u=Used object

  c=Object for calculation purpose, p=predicate

(b) Components of the model:

- Here are the modeling rules.

  1. To every statement there is a node, whose name (number) is unique.

  Every node has at least one outlink and at least one inlink except exit nodes, which do not have outlinks, and entry nodes, which do not have inlinks.

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.

3.Another components is simple statements. These are the statements with only one outlink. The weight of simple statement is determined by sequential actions of data-flow with respect to the given statement.

For example, consider a simple statement A:= A + B in most languages is weighted by *cd* or possibly *ckd* for variable A.

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

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

6. 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.

7. If multiple data-flow actions are available on a link for a variable, then its corresponding weight is determined by the sequence of actions. Inversely a sequence of equivalent links are used to replace the link with more data flow actions.

(c) Putting it together:

- The following **figure a** shows the control flowgraph. The **figure b** shows this control flowgraph annotated for variables X and Y data flows.
- The **figure c** shows the same control flowgraph annotated for variable Z. Z is first defined by an assignment statement on the first link.
- Z is used in a predicate (Z >= 0?) at node 3, and therefore both outlinks of that node—(3,4) and (3,5)—are marked with a *p*. The data-flow annotation for variable V is shown in **figure d**.

(4) Strategies in Data-Flow Testing:

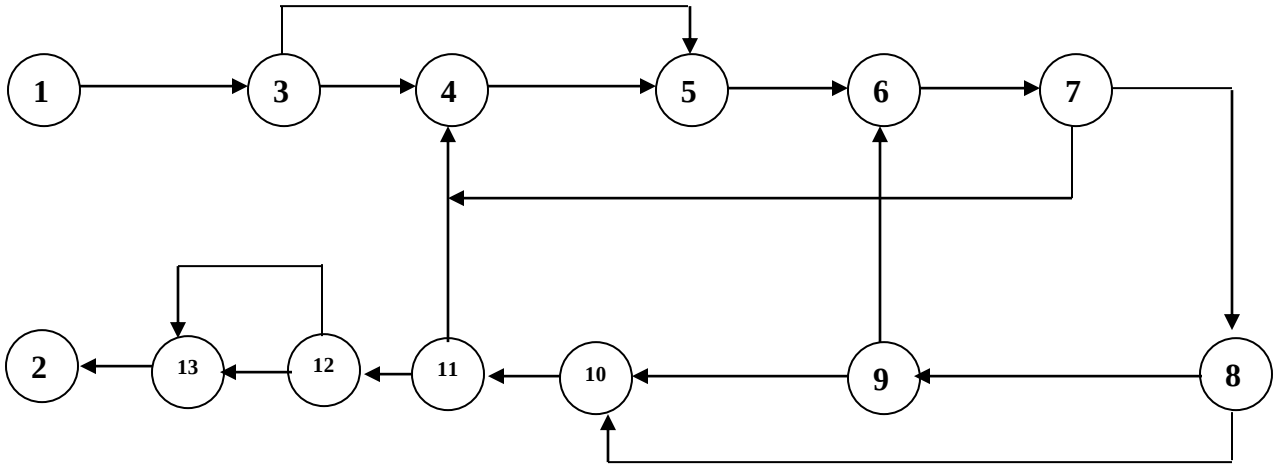
(i) General:

- Data-flow testing strategies are structural strategies.
- Data-flow testing strategies are based on the program's control flowgraph.
- Data-flow testing strategies are based on selecting test path segments (also called subpaths) that satisfy some characteristic of data flows for all data objects. For example, all subpaths that contain a *d* (or *u, k, du, dk*).
- These strategies differ in determining whether the paths of a given type are required or only one path of that type is required.
- The test set includes the predicate uses and computational uses of variables.
- This usage also differs in the test set that is either computational use or predicate use of variables.

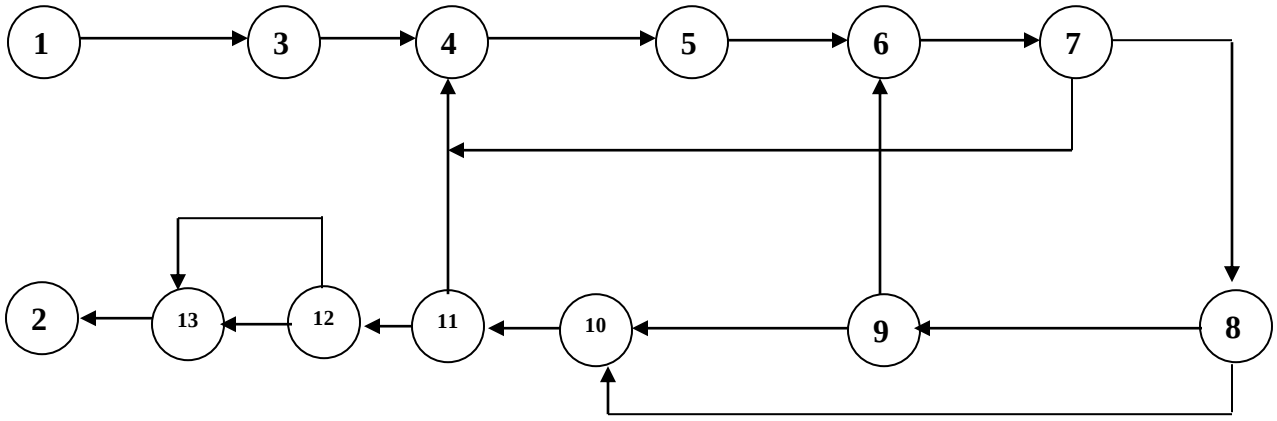
(ii) Terminology:

- We'll assume for the moment that all paths are achievable. Some terminology.
- A definition-clear path segment
  - A path segment is a sequence of connected links between nodes. This first link of the path is defined and the subsequent link of that path is killed.
  - A definition-clear path segment is a connected sequence of links such that X is (possibly) defined on the first link and not redined or killed on any subsequent link of that segment.
  - All paths in figure b are definition clear because variables X and Y are defined only on the first link (1,3) and thereafter. Similarly for variable V in figure d.

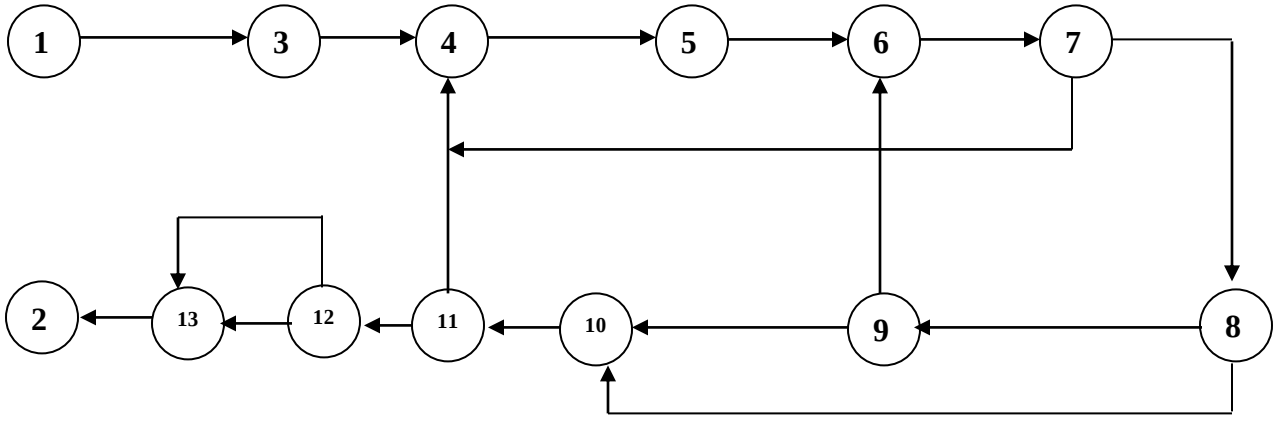
- In Figure c we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11).
- Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5).
- For practice, try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



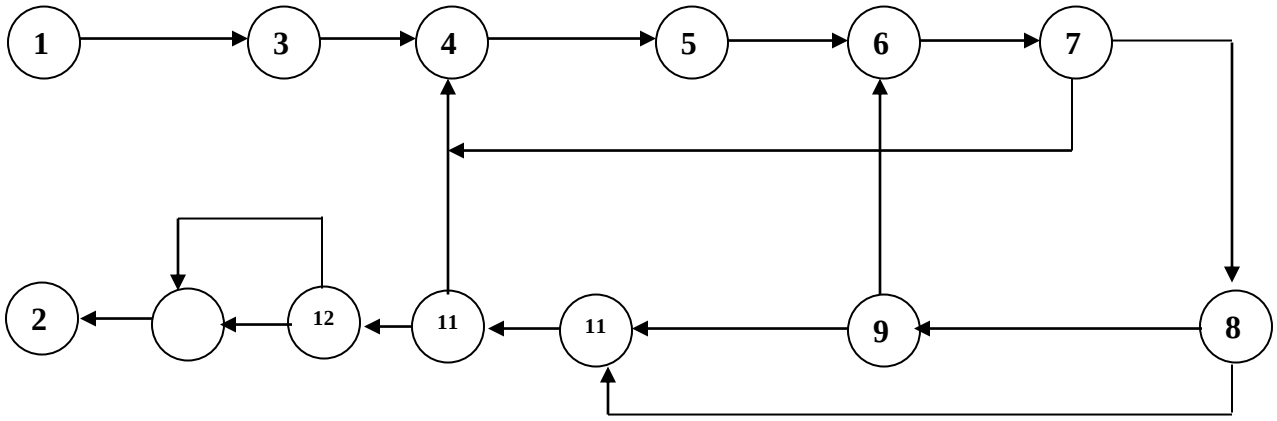
(a) Unannotated Control Flowgraph



(b) Control Flowgraph Annotated for X and Y Data Flows.



(c) Control Flowgraph Annotated for Z Data Flow



(d) Control Flowgraph Annotated for V Data Flow

- ❖ The fact that there is a definition-clear subpath between two nodes does not imply that all subpaths between those nodes are definition-clear; in general, there are many subpaths between nodes, and some could have definitions on them and some not.
- ❖ A definition clear sub path does not include loops. For example a loop consists of (i,j) and (j,i) links.
- ❖ These links have a definition on (i,j) and a computational use on (j,i). If we include loops in a path by definition-clear path segment then there is no need to go around such path.
- ❖ Because of this the testing strategies will have a finite number of test paths.
- ❖ The strategies must be weaker than the paths because a bug can be created whenever a loop has been traversed and iterated.

2. A loop-free path segment

- ❖ A loop-free path segment is a path segment for which every node is visited at most once.
- ❖ Path (4,5,6,7,8,10) in figure c is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

3.A simple path segment

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

4.A du path

- ❖ 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 path.
- ❖ if the penultimate 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.

(iii) The Strategies:

(a) Overview:

- ❖ The structural test strategies are based on the program's control flowgraph.
- ❖ These strategies differ in determining whether the paths of a given type are required or only one path of that type is required.
- ❖ The test set includes the predicate uses and computational uses of variables.

- This usage also differs in the test set that is either computational use or predicate use of variables.
- The different data flow testing strategies are given below.

(b) All-du Paths (ADUP) strategy:

- 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.
- In the above figure b variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).
- The situation for variable Z in figure c is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...).
- The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).
- Variable V in figure d is defined only once on link (1,3).
- Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-*du*-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4).
- Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions.
- They must be included because they provide alternate *du* paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.
- The all-*du*-paths strategy *is* a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

(c) All-uses Strategy:

- Just as we reduced our ambitions by stepping down from all paths (P_∞) to branch coverage (P_2), say, we can reduce the number of test cases by asking that the test set include *at least one* path segment from every definition to every use that can be reached by that definition—this is called the all-uses (AU) strategy.
- The 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.
- In figure d, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both.
- Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath.

(d) All-p-Uses/Some-c-Uses and All-c-Uses/Some-p-Uses Strategies:

- Weaker criteria require fewer test cases to satisfy. We would like a criterion that is stronger than P_2 but weaker than AU.
- Therefore, select cases as for All (Section 3.3.3) except that if we have a predicate use, then (presumably) there's no need to select an additional computational use (if any). More formally, the all-*p*-uses/some-*c*-uses (APU+C) strategy is defined as follows: 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 variable that

are not covered by the above prescription, then add computational-use test cases as required to cover every definition.

- The all-*c*-uses/some-*p*-uses (ACU+P) strategy reverses the bias: 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.
- In figure b for variables X and Y, any test case satisfies both criteria because definition and uses occur on link (1,3). In figure c, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the *c*-use of Z: but that's okay according to the strategy's definition because every definition is covered.
- Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z.
- Find a covering set of test cases under APU+C for all variables in this example—it only takes two tests. In figure d, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the *c*-use at (9,10) need not be included under the APU+C criterion.
- The figure d shows a single definition for variable V. C-use coverage is achieved by (1,3,4,5,6,7,8,9,10,11,12,13,2). In figure c, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) *p*-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) *p*-uses.
- The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

(e) All definitions Strategy:

- The all-definitions (AD) strategy asks only that every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.
- Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V. From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

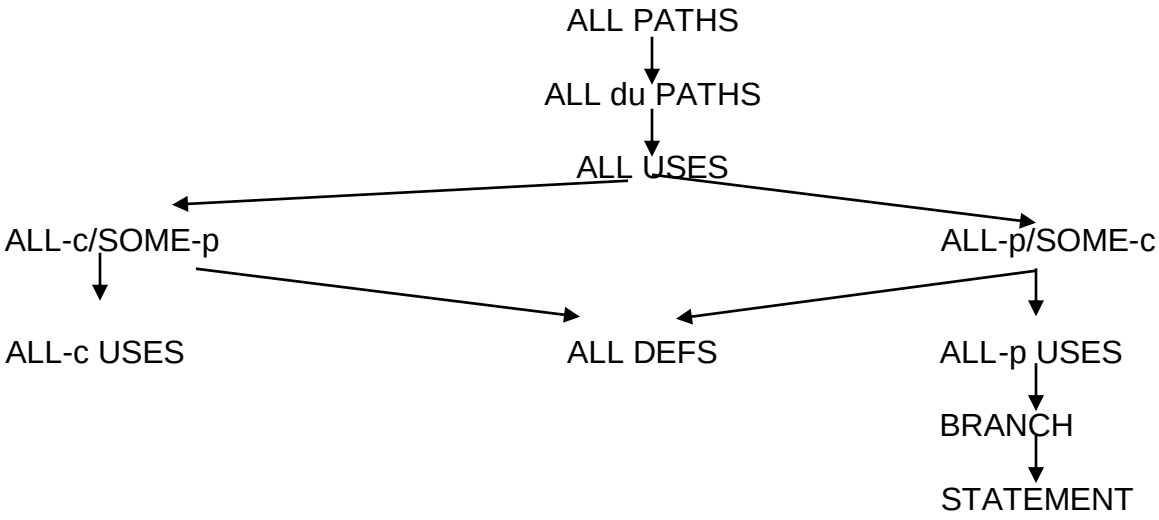
(f) All-Predicate Uses, All-Computational Uses Strategies:

- The all-predicate-uses (APU) strategy is derived from the 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 following each definition.
- Similarly, the all-computational-uses (ACU) strategy is derived from ACU+P by dropping the requirement that we include a *p*-use if there are no *c*-use instances following a definition.
- It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

(g) Ordering the Strategies:

- The below figure compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head.
- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.
- Variations of data-flow strategies exist, including different ways of characterizing the paths to be included and whether or not the selected paths are achievable.

- The strength relation graph of the above figure can be substantially expanded to fit almost all such strategies into it. Indeed, one objective of testing research has been to place newly proposed strategies into the hierarchy.



(iv) Slicing, Dicing, Data Flow and Debugging:

(a) General:

- Slicing is a program originally developed for conventional languages.
- It helps in understanding data flow and debugging techniques. The Slicing is done based on variable sharing.
- Dicing and debugging are the concepts related to removal of unwanted bugs.

(b) Slices and Dices:

- There are two types of slicing technique. i.e. Static slicing & dynamic slicing.
- Static slicing is a part of a program defined with respect to a given variable X and a statement *i*:
- It consists of all statements that could affect the value of X at statement *i*.
- The result of a false statement effect in an improper computational use or predicate use of some other variable.
- If the variable X is correct then the bug is detected in the program itself.
- A program dice is a part of a slice in which the statements which are correct has been removed.
- The idea behind slicing and dicing is based on Weiser's observation that these constructs are at the heart of the procedure followed by good debuggers.
- Dynamic slicing is a refinement of static slicing. Dynamic slicing compares the data flow relationship with respect to static data flows.
- Dicing is defined as the process of refining slice by removing all the unwanted bug statements in a program.
- Basically a dice is generated from a slice which posses the information about testing or debugging the function of a dice is to improve or refine a slice by removing the unwanted statements from a program.
- The process of dicing is often employed by debuggers. The current methods of dicing encompass assumptions related to bugs and programs.
- Due to the existence of bugs the usage of real program is declined.

(c) Data-flow:

- Data flow is defined as the process of reading variables. The central concept of data-flow is to bridge the gap between debugging and testing.
- The idea of slices was extended to arrays and data vectors and the data-flow relations (such as dc and dp) in dynamic slices are analogous compared to the data-flow relations in static slices (dc and dp).
- Where dc and dp are the data objects. Here
  d=Object definition,
  c=Computation
  p=Symbol used in a predicate for operation purpose.

(d) Debugging:

- Debugging is defined as an iterative method in which refinement of slices is carried out through dices so as to obtain the dicing information.
- Basically debugging is carried out after a test case is successfully executed.
- The process of debugging terminates when all the bugs that exists in the program statements are corrected.
- Methods of slicing leads to commercial testing or development of different debugging tools.
- The test cases involved in integration and testing are modeled for efficient error detection, where as the cases involved in debugging are modeled for efficient error isolation.

(5) Application of Data-Flow Testing:

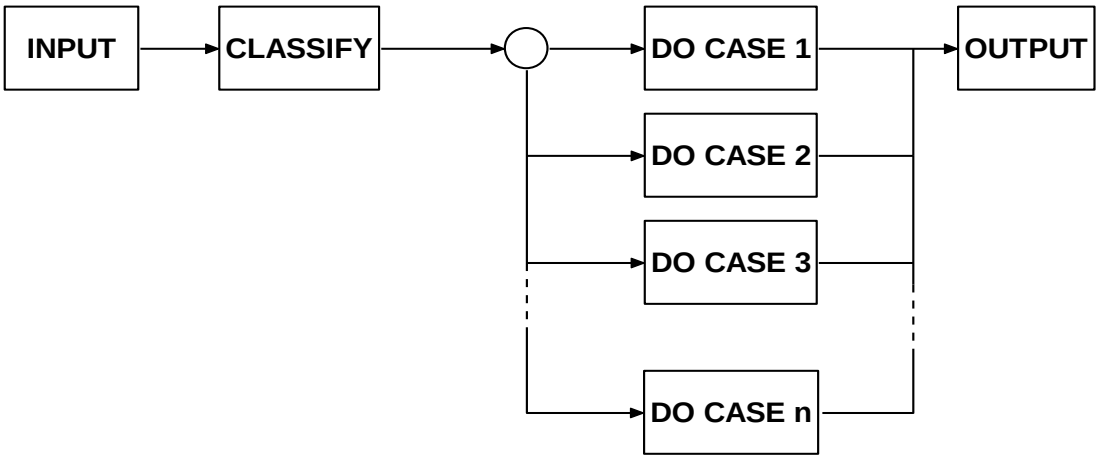
- Data flow testing is used to detect the different abnormalities that may arise due to data flow anomalies.
- Data flow testing shows the relationship between the data objects that represents data.
- Data flow testing strategies help in determining the usage of variables that are included in the test set.
- Data flow testing is cost effective.
- Data flow testing solves the problems that are encountered while performing.
- Data flow testing uses practical applications rather than mathematical applications.
- Data flow testing is used in developing web applications with Java technology.

UNIT –III
DOMAIN TESTING

(1) Domains and paths:

(i) The Model:

- Domain testing can be based on specifications and/or equivalent implementation information.
- If domain testing is based on specifications, it is a functional test technique; if based on implementations, it is a structural technique.
- Domain testing is applied to one input variable or to simple combinations of two variables, based on specifications.
- The schematic representation of Domain testing is given below.



- First the different input variables are provided to a program.
- The classifier receives all input variables and divides them into different cases.
- Every case there should be at least one path to process that specified case.
- Finally output is received from this do cases..

(ii) A domain is a set:

- An input domain is a set. If the source language supports set definitions less testing is needed because the compiler (compile-time and run-time) do much of it for us.

(iii) Domains, paths and predicates:

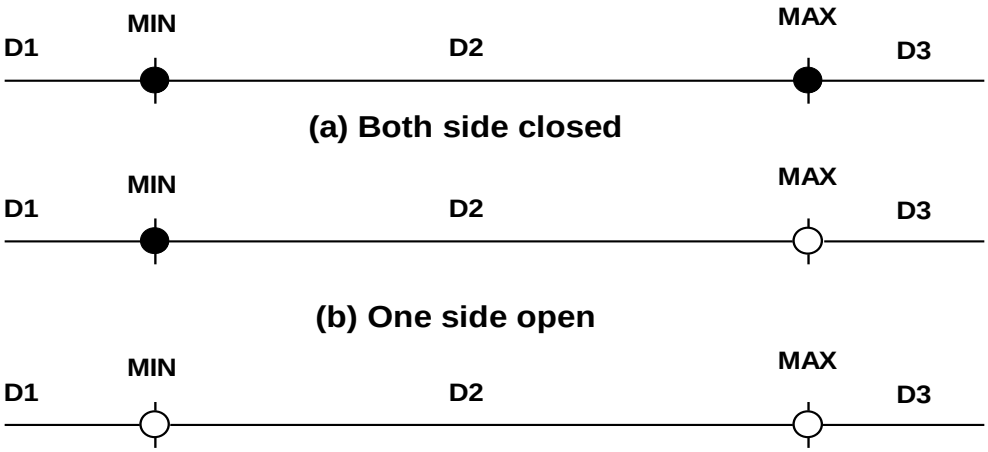
- In domain testing, predicates are assumed to be interpreted in terms of input vector variables.
- If domain testing is applied to structure (implementation), then predicate interpretation must be based on control flowgraph.
- If domain testing is applied to specifications, then predicate interpretation is based on data flowgraph.
- For every domain there is at least one path through the routine.
- There may be more than one path if the domain consists of disconnected parts.
- Unless stated otherwise, we'll assume that domains consist of a single, connected part.
- We'll also assume that the routine has no loops.
- Domains are defined by their boundaries. For every boundary there is at least one predicate.
- For example in the statement, IF X > 0 THEN ALPHA ELSE BETA we know that number greater than zero, belong to ALPHA, number smaller to zero, belong to BETA.

Review:

1. A domain is a loop free program.
2. For every domain there is at least one path through the routine.
3. The set of interpreted predicates defines the domain boundaries.

(iv) Domain Closure:

- To understand the domain closure, consider the following figure.



- If the domain boundary point belongs to the same domain then the boundary is said to close. If the domain boundary point belongs to some other domain then the boundary is said to open.
- In the above figure there are three domains D1, D2, D3.
- In **figure a** D2's boundaries are closed both at the minimum and maximum values. If D2 is closed, then the adjacent domains D1 and D3 must be open.
- In **figure b** D2 is closed on the minimum side and open on the maximum side, meaning that D1 is open and D3 is closed. In **figure c** D2 is open on both sides, which mean that the adjacent domains D1 and D3 must be closed.

(v) Domain Dimensionality:

- Depending on the input variables, the domains can be classified as number line domains, planer domains or solid domains.
- That is for one input variable the value of the domain is on the number line, for two variables the resultant is planer and for three variables the domain is solid.
- One important thing here is to note that we need not worry about the domains dimensionality with the number of predicates. Because there might be one or more boundary predicates.

(vi) The Bug Assumptions:

- The bug assumption for domain testing is that processing is okay but the domain definition is wrong.
- An incorrectly implemented domain means that boundaries are wrong, which mean that control-flow predicates are wrong.
- The following are some of the bugs that give to domain errors.

(a) Double-Zero Representation:

- Boundary errors for negative zero occur frequently in computers or programming languages where positive and negative zeros are treated differently.

(b) Floating-Point Zero Check:

- A floating-point number can equal to zero only if the previous definition of that number is set it to zero or if it is subtracted from itself, multiplied by zero.
- Floating-point zero checks should always be done about a small interval.

(c) Contradictory Domains:

- Here at least two assumed distinct domains overlap.

(d) Ambiguous Domains:

- These are missing domain, incomplete domain.

(e) Over specified Domains:

- The domain can be overloaded with so many conditions.

(f) Boundary Errors:

- This error occurs when the boundary is shifted or when the boundary is tilted or missed.

(g) Closure Reversal

- This bug occurs when we have selected the wrong predicate such as x>=0 is written as x<=0.

(h) Faulty Logic:

- This bug occurs when there are incorrect manipulations, calculations or simplifications in a domain.

(vii) Restrictions:

(a) General

- Domain testing has restrictions. i.e. we cannot use domain testing if they are violated.
- In testing there is no invalid test, only unproductive test.

(b) Coincidental Correctness

- Coincidental correctness is assumed not to occur.
- Domain testing is not good for which outcome is correct for the wrong reason.
- One important point to be noted here is that, domain testing does not support Boolean outcomes (TRUE/FALSE).
- If suppose the outputs are some discrete values, then there are some chances of coincidental correctness.

(c) Representative Outcome

- Domain testing is an example of partition testing.
- Partition testing divide the program's input space into domains.
- If the selected input is shown to be correct by a test, then processing is correct, and inputs within that domain are expected to be correct.
- Most test techniques, functional or structural fall under partition testing and therefore make this representative outcome assumption.

(d) Simple Domain Boundaries and Compound Predicates

- Each boundary is defined by a simple predicate rather than by a compound predicate.
- Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, x >= 0 .AND. x < 17, just specifies two domain boundaries by one compound predicate.

(e) Functional Homogeneity of Bugs

- Whatever the bug is, it will not change the functional form of the boundary predicate.

(f) Linear Vector Space

- A linear predicate is defined by a linear inequality using only the simple relational operators >, >=, =, <=, <>, and <.
- Example $x^2 + y^2 > a^2$.

(g) Loop-free Software

- Loops (indefinite loops) are problematic for domain testing.

- If a loop is an overall control loop on transactions, say, there's no problem.
- If the loop is definite, then domain testing may be useful for the processing within the loop, and loop testing can be applied to the looping values.

(2) Nice Domains:

(i) Where Do Domains Come From?

- Domains are often created by salesmen or politicians.
- The first step in applying domain testing is to get consistent and complete domain specifications.

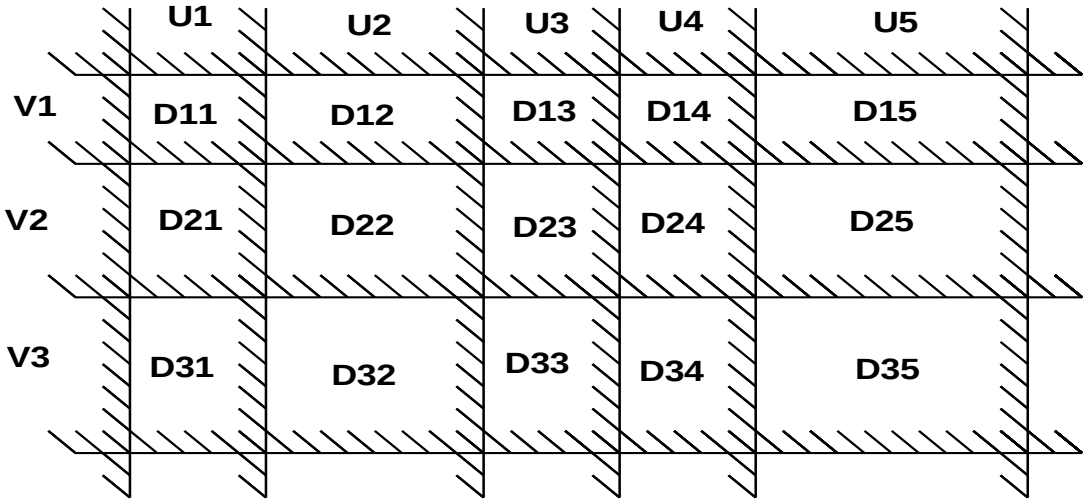
(ii) Specified versus Implemented Domains:

- Implemented domains can't be incomplete or inconsistent but specified domains can be incomplete or inconsistent.
- Incomplete means that there are input vectors for which no path is specified and inconsistent means that there are at least two contradictory specifications.

(iii) Nice Domains:

(1) General

- The representation of Nice two-dimensional domains is as follows. .



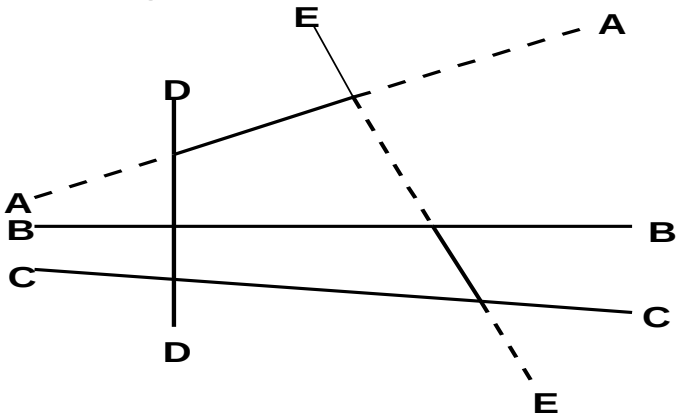
- The U and V represent boundary sets and D represents domains.
- The boundaries have several important properties. They are linear, complete, systematic, orthogonal, consistently closed, simply connected and convex.
- If domains have these properties, domain testing is very easy otherwise domain testing is tough.

(2) Linear and Nonlinear Boundaries

- Nice domain boundaries are defined by linear inequalities or equations.
- The effect on testing comes from only two points then it represents a straight line.
- If it considers three points then it represents a plane and in general it considers n + 1 points then it represents an n-dimensional hyperplane.
- Linear boundaries are more frequently used than the non-linear boundaries.
- We can linearize the non-linear boundaries by using simple transformations.

(3) Complete Boundaries

- Complete boundaries are those boundaries which do not have any gap between them.
- Nice domain boundaries are complete boundaries because they cover from plus infinity to minus infinity in all dimensions.
- Incomplete boundaries are those boundaries which consist of some gaps between them and are not covered in all dimensions.
- The following figure represents some incomplete boundaries.



- The Boundaries A and E have gaps so they are incomplete & the boundaries B, C, D are complete.
- The main advantage of a complete boundary is that it requires only one set of tests to verify the boundary

(4) Systematic Boundaries

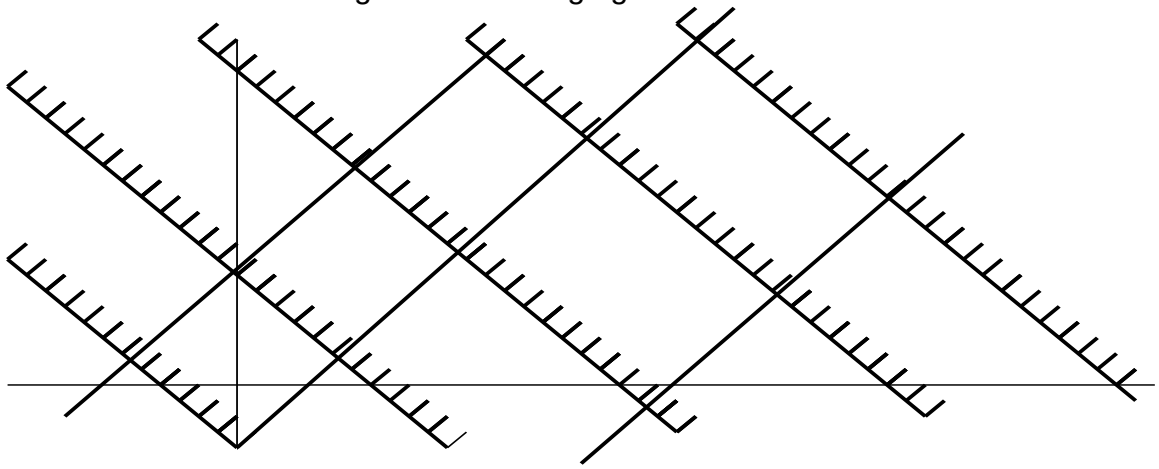
- Systematic boundaries refer to boundary inequalities with simple mathematical functions such as a constant.
- Consider the following relations,

  $f_1(X) >= k_1$ or $f_1(X) >= g(1,c)$
  $f_2(X) >= k_2$ $f_2(X) >= g(2,c)$
  ....
  $f_i(X) >= k_i$ $f_i(X) >= g(i,c)$

  - Where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function that yields a constant, such as $k + ic$.

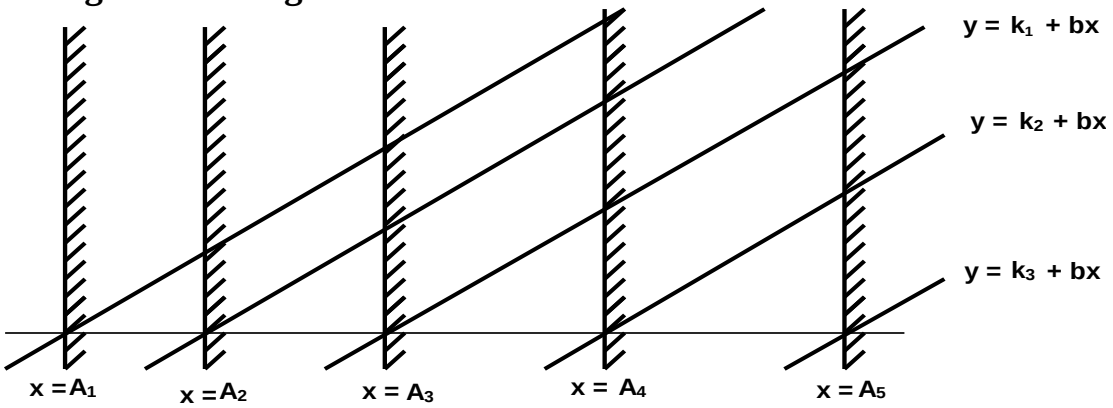
(5) Orthogonal Boundaries

- The *U* and *V* boundary sets in Nice two-dimensional domains figure are orthogonal; that is, the every boundary *V* is perpendicular to every other boundary *U*.
- If two boundary sets are orthogonal, then they can be tested independently.
- If we want to tilt the above orthogonal boundary we can do it by testing its intersection points but this can change the linear growth, O(n) into the quadratic growth $O(n^2)$.
- If we tilt the boundaries to get the following figure then we must test the intersections.



(6) Closure Consistency

- Consistent closures are the most simple and fundamental closure.
- It gives consistent and systematic results.
- The following figure shows the boundary closures are consistent.



- In the above figure, the shading lines show one boundary and thick lines show other boundary.
- It shows Non orthogonal domain boundaries, which mean that every inequality in domain x is not perpendicular to every inequality in domain y.

(7) Convex

- A figure is said to be convex when for any two boundaries, with two points placed on them are combined by using a single line then all the points on that line are within the range of the same figure.
- Nice domains support convex property, where as dirty domains don't.

(8) Simply Connected

- Nice domains are usually simply connected because they are available at one place as a whole but not dispersed in other domains..
- Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

(iv) Ugly Domains:

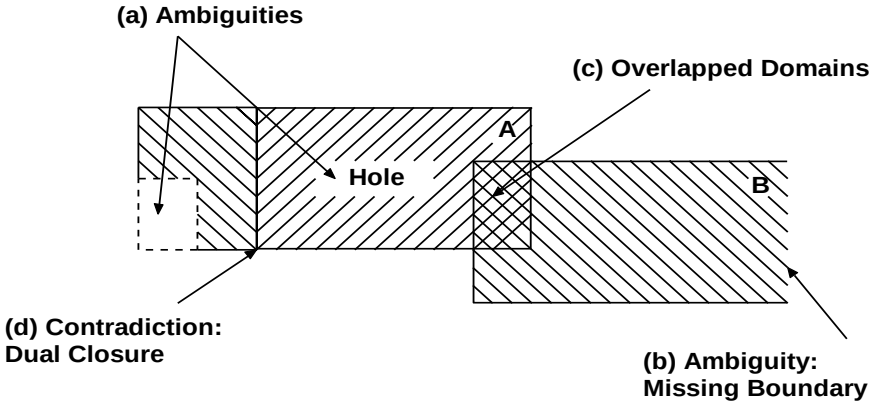
(a) General

- Some domains are born ugly. Some domains are bad specifications.
- So every simplification of ugly domains by programmers can be either good or bad.
- If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.
- But if the domain's complexity is essential such simplifications gives bugs.

(b) Nonlinear Boundaries

- Non linear boundaries are rare in ordinary programming, because there is no information on how programmers correct such boundaries.
- So if a domain boundary is non linear, then programmers make it linear.

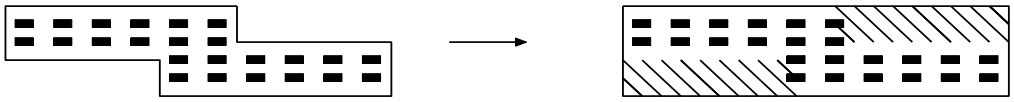
(c) Ambiguities and Contradictions:.



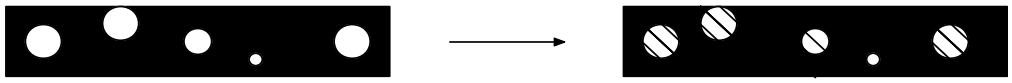
- Domain ambiguity is missing or incomplete domain boundary.
- In the above figure Domain ambiguities are holes in the A domain and missing boundary in the B domain.
- An ambiguity for one variable can be see easy.
- An ambiguity for two variables can be difficult to spot.
- An ambiguity for three or more variables impossible to spot. Hence tools are required.
- Overlapping domains and overlapping domain closure is called contradiction.
- There are two types of contradictions are possible here.
  (1) Overlapped domain specifications
  (2) Overlapped closure specifications.
- In the above figure there is overlapped domain and there is dual closure contradiction. This is actually a special kind of overlap.

(d) Simplifying the Topology

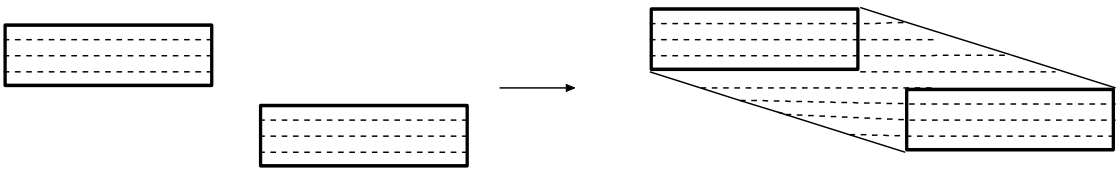
- Connecting disconnected boundary segments and extending boundaries is called simplifying the topology
- There are three generic cases of simplifying the topology.



(a) Making it convex



(b) Filling in the Holes

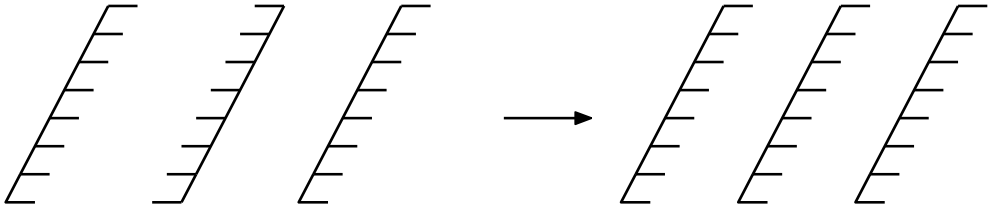


(c) Joining the Pieces

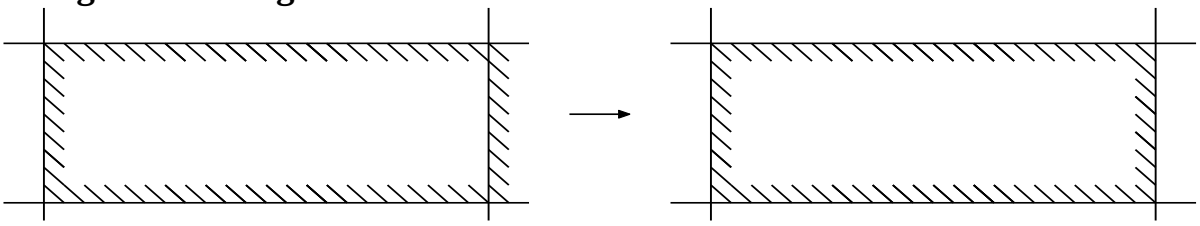
- Programmers introduce bugs and testers misdesign test cases by, smoothing out concavities, filling in holes, joining disconnected pieces.

(e) Rectifying Boundary Closures

- Different boundaries in different directions can obtain in consistent direction is called rectifying boundary closures.
- That is domain boundaries which are different directions can obtain in one direction.



(a) Consistent Direction



(b) Inclusion/Exclusion Consistency

- In the above figure the hyper plane boundary is outside that can obtain inside. This is called inclusion / exclusion consistency.

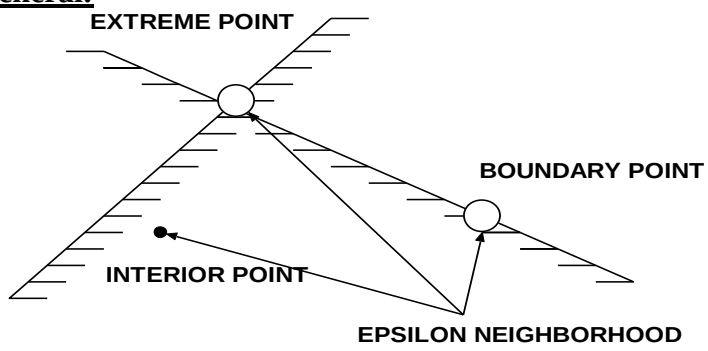
(3) Domain Testing:

(i) Overview:

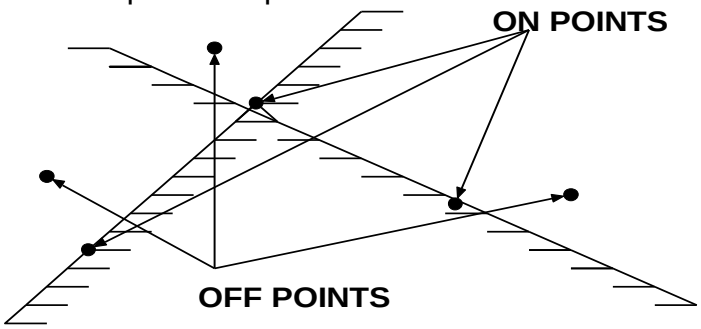
- Domains are defined by their boundaries. So domain testing concentrates test points on boundaries or near boundaries.
- Find what wrong with boundaries, and then define a test strategy.
- Because every boundary uses at least two different domains, test points used to check one domain can also be used to check adjacent domains.
- Run the tests, and determine if any boundaries are faulty.
- Run enough tests to verify every boundary of every domain.

(ii) Domain Bugs and How to Test for Them:

(a) General:

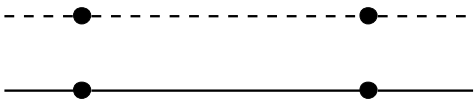


- An interior point is a point in a domain. It can be defined as a point which specifies certain distance covered by some other points in the same domain.
- This distance is known as epsilon neighborhood.
- A boundary point is on the boundary that is a point with in a specific epsilon neighborhood.
- An extreme point is a point that does not lie between any other two points.

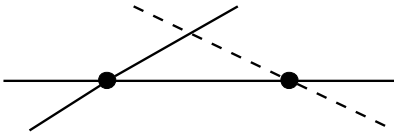


- An on point is a point on the boundary. An off point is outside the boundary.
- If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.

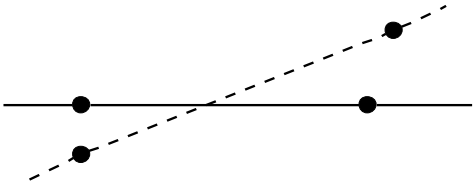
- If the domain boundary is open, an off point is a point near the boundary but in the same domain.
- Here we have to remember CLOSED OFF OUTSIDE, OPEN OFF INSIDE
- i.e. COOOOI
- The following figure shows a generic domain ways.



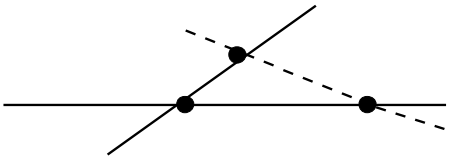
SHIFTED BOUNDARIES



EXTRA BOUNDARY



TILTED BOUNDARIES



MISSING BOUNDARY

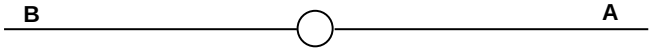
OPEN / CLOSE ERROR

CORRECT ——————

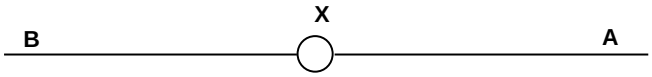
INCORRECT - - - - - - -

(b) Testing One-Dimensional Domains:

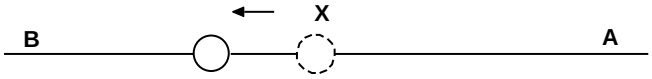
- The following figure shows one dimensional domain bugs for open boundaries.



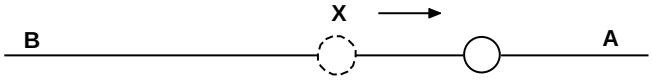
a) An Open Domain (A)



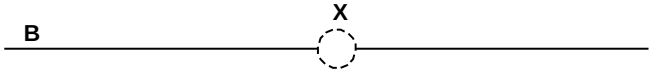
b) Closure bug



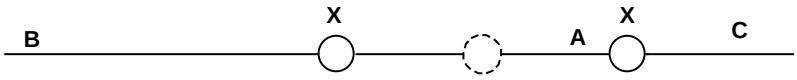
c) Boundary shifted left



c) Boundary shifted right

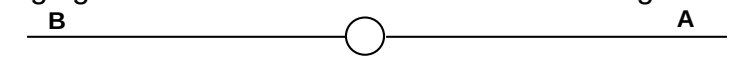


e) Missing Boundary

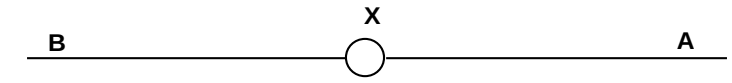


f) Extra Boundary

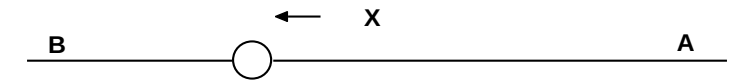
- In the above figure a) we assume that the boundary was to open for A.
- In figure b) one test point (marked X) on the boundary detects the bug.
- In figure c) a boundary shifts to left.
- In figure d) a boundary shifts to right.
- In figure e) there is a missing boundary. In figure f) there is an extra boundary.
- The following figure shows one dimensional domain bugs for closed boundaries.



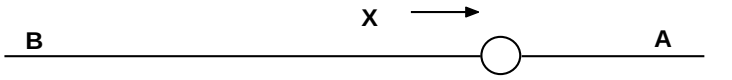
a) A closed Domain (A)



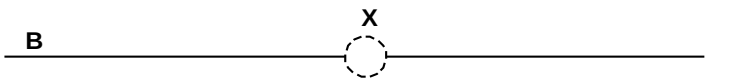
b) Closure bug



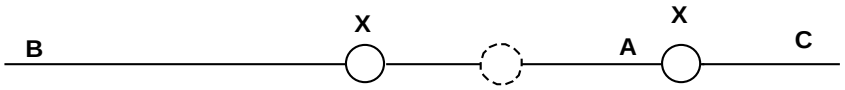
c) Boundary shifted left



c) Boundary shifted right



e) Missing Boundary



f) Extra Boundary

- In the above figure a) we assume that the boundary was to close for A.
- In figure b) one test point (marked X) on the boundary detects the bug.
- In figure c) a boundary shifts to left. In figure d) a boundary shifts to right.
- In figure e) there is a missing boundary. In figure f) there is an extra boundary.
- Only one difference from this diagram to previous diagram is here we have closed boundaries.

(c) Testing Two-Dimensional Domains:

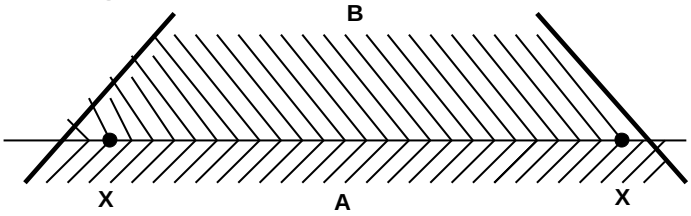
- The following figure shows domain boundary bugs for two dimensional domains.
- A and B are adjacent domains, and the boundary is closed with respect to A and the boundary is opened with respect to B.

(i) Closure Bug:

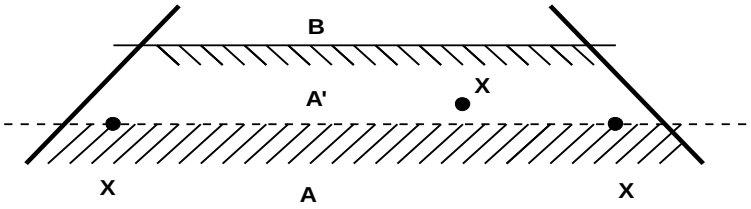
- The figure (a) shows a wrong closure, that is caused by using a wrong operator for example, x>=k was used when x > k was intended.
- The two on points detect this bug.

(ii) Shifted Boundary:

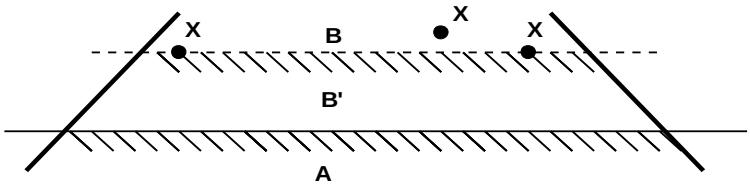
- In figure (b) the bug is shifted up, which converts part of domain B into A'.
- This is caused by incorrect constant in a predicate for example x + y >= 17 was used when x + y > = 7 was intended. Similarly figure (c) shows a shift down.



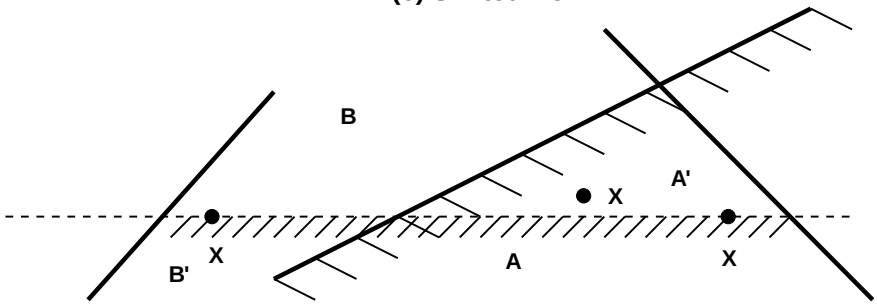
(a) Closure Bug



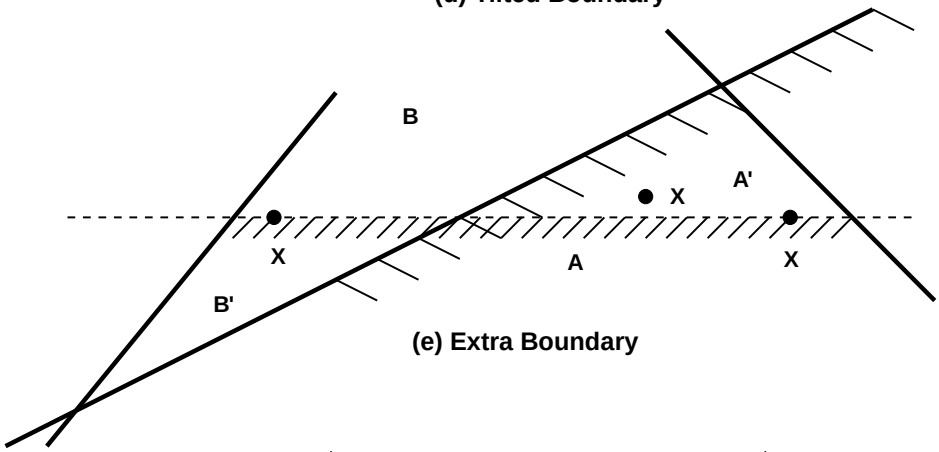
(b) Shifted Up



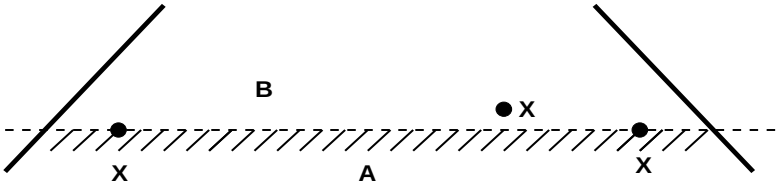
(c) Shifted Down



(d) Tilted Boundary



(e) Extra Boundary



(f) Missing Boundary

(iii) Tilted boundary:

- A tilted boundary occurs, when coefficients in the boundary inequality are wrong.
- For example we used 3x + 7y > 17 when 7x + 3y > 17 is needed.
- Figure (d) shows a tilted boundary which creates domain segments A' and B'.

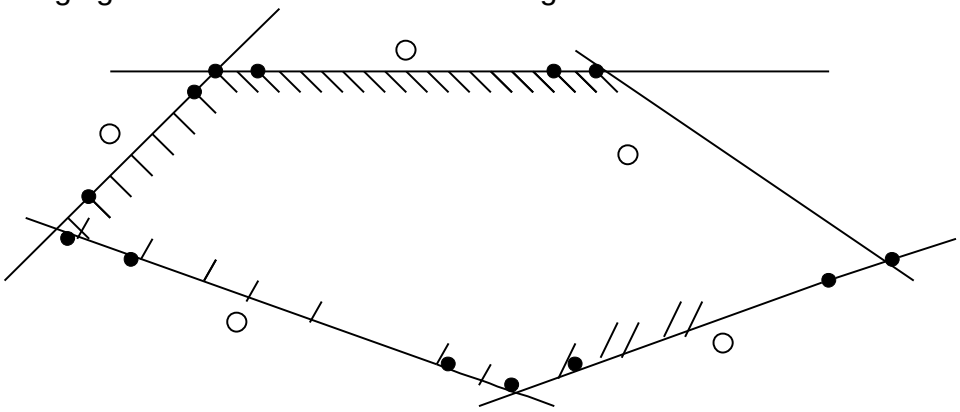
(iv) Extra Boundary:

- An extra boundary is created by an extra predicate.
- Figure (e) shows an extra boundary. The extra boundary is caught by two on points.

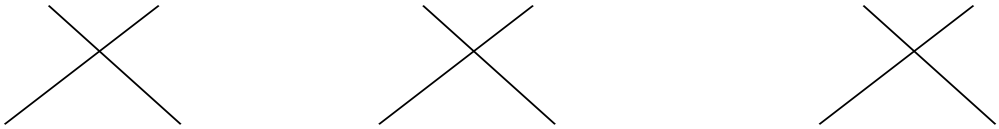
(v) Missing Boundary:

- A missing boundary is created by leaving out the predicate.
- A missing boundary shown in figure (f) is caught by two on points.

- The following figure summarizes domain testing for two dimensional domains.

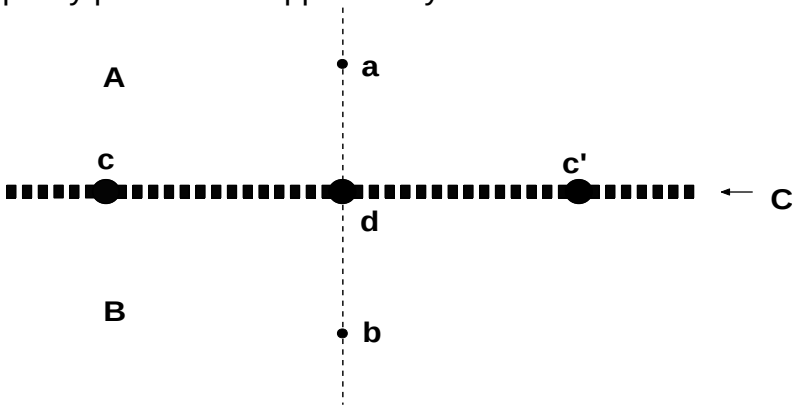


- There are two on points (closed circles) for each segment and one off point (open circle)
- Note that the selected test points are shared with adjacent domains.
- The on points for two adjacent boundary segments can also be shared.
- The shared on points is given below.



(d) Equality and Inequality Predicates:

- Equality predicates are defined by equality equation such as x + y =12.
- Equality predicates supports only few domain boundaries



- Inequality predicates are defined by inequality equation such as x + y > 12 or x + y <12
- Inequality predicates supports most of the domain boundaries.
- In domain testing, equality predicate of one dimension is a line.
- Similarly equality of two dimensions is a two dimensional domain and equality of three dimensions is a planer domain.

- ❖ Inequality predicates test points are obtained by taking adjacent domains into consideration.
- ❖ In the above figure the three domains A, B, C are planer. The domain C is a line.
- ❖ Here domain testing is done by two on points & two off points.
- ❖ That is test point b for B, and test point a for A and test points c and c' for C.

(e) Random Testing:

- ❖ Random testing is a form of functional testing that is useful when the time needed to write and run directed tests are too long.
- ❖ One of the big issues of random testing is to know when a test fails.
- ❖ When doing random testing we must ensure that they cover the specification.
- ❖ The random testing is less efficient than direct testing. But we need random test generators.

(f) Testing n-Dimensional Domains:

- ❖ If domains defined over n-dimensional input space with p-boundary segments then the domain testing gives testing n-dimensional domains.

(iii) Procedure:

- ➢ Generally domain testing can be done by hand for two dimensions.
- ➢ Without tools the strategy is practically impossible for more than two variables.

1. Identify the input variables.
2. Identify variables which appear in domain predicates.
3. Interpret all domain predicates in terms of input variables.
4. For p binary predicates there are 2^p domains.
5. Solve the inequalities to find all the extreme points of each domain.
6. Use the extreme points to solve for near by on points.

(iv) Variations, Tools, Effectiveness:

- ➢ Variations can vary the number of on and off points or the extreme points.
- ➢ The basic domain testing strategy discussed here is called the N X 1 strategy, because it uses N on points and one off point.
- ➢ In cost effectiveness of domain testing they use partition analysis, which includes domain testing, computation verification and both structural and functional information.
- ➢ Some specification tools are used in domain testing.

(4) Domains and Interface Testing:

(i) General:

- ➢ The domain testing plays a very important role in integration testing. In integration testing we can find the interfaces of different components.
- ➢ We can determine whether the components are accurate or not.

(ii) Domains and Range:

- ➢ Domains are the input values used. Range is just opposite of domains.
- ➢ i.e. Range is output obtained.
- ➢ In most testing techniques, more forces on the input values.
- ➢ This is because with the help of input values it will be easy to identify the output.
- ➢ But interface testing gives more forces on the output values.
- ➢ An interface test consists of exploring the correctness of the following mappings.

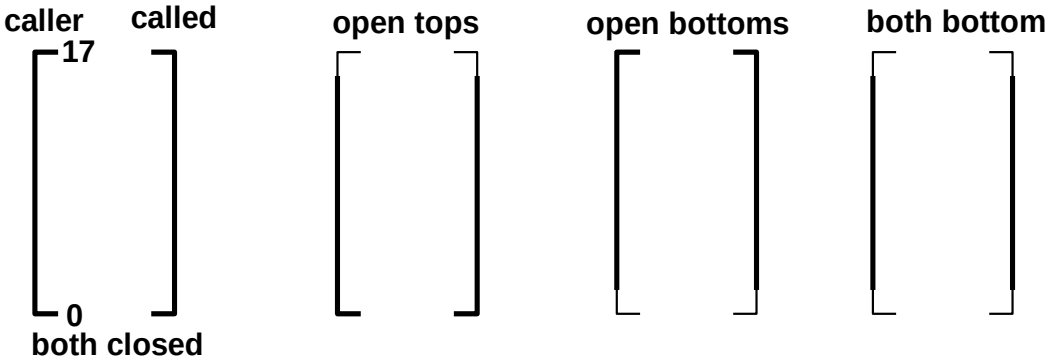
Caller domain ⟶ Caller range

Caller range ⟶ Called domain

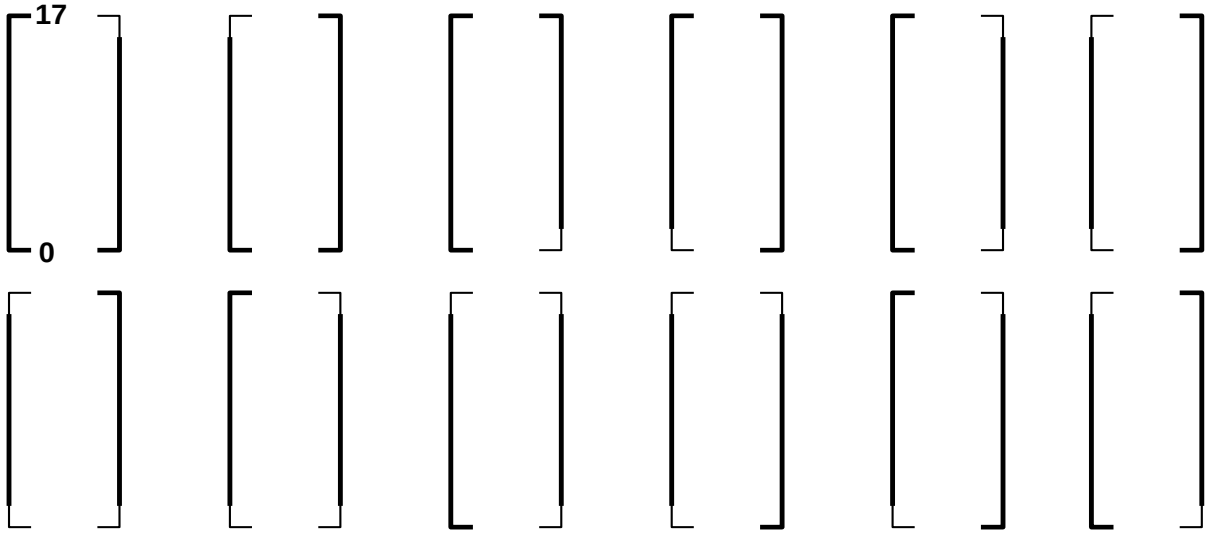
Called domain ⟶ Called range

(iii) Closure Compatibility:

- Assume that the caller's range and the called domain spans the same numbers say 0 to 17
- The closure compatibility shows the four cases in which the caller's range closure and the called's domain closure can agree.
- The four cases consists of domains that are closed on top (17) & bottom (0), open top & closed bottom, closed top & open bottom and open top & bottom.
- Here the thick line represents closed and thin line represents open.



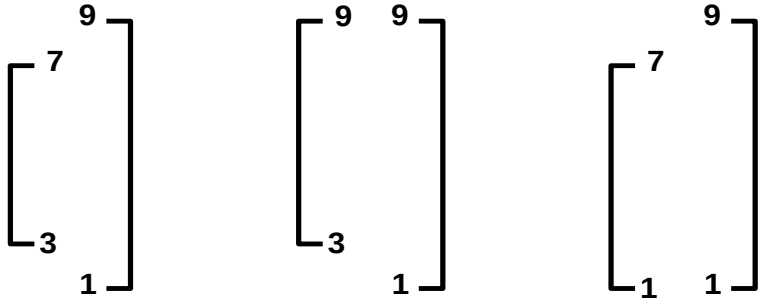
- The following figure shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs.



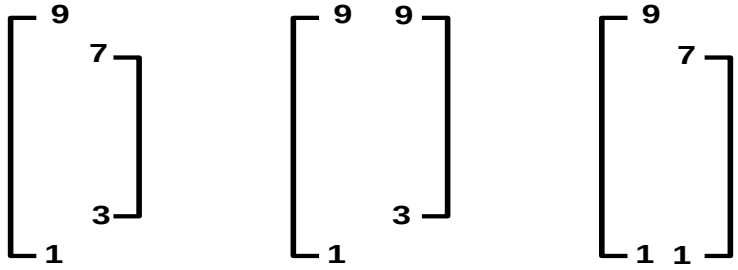
- Here the four cases in which a caller boundary is open and the called is closed are not buggy.

(iv) Span Compatibility:

- The following figure shows three possibly harmless of span incompatibilities.
- In this figure Caller span is smaller than Called.



- The range of a caller is a sub set of the called domain. That is not necessarily a bug.
- The following figure shows Called is Smaller than Caller.



(v) Interface Range/ Domain Compatibility Testing:

- The application of domain testing is also very important for interface testing because it tests the range and domain compatibilities among caller and called routines.
- It is the responsibility of the caller to provide the valid inputs to the called routine.
- After getting the valid input, the test will be done on every input variable.

(vi) Finding the values:

- Start with the called routine's domains and generate test points.
- A good component test should have included all the interesting domain-testing cases.
- Those test cases are the values for which we must find the input values of the caller.

(5) Domains and Testability:

(i) General:

- Domain testing gives orthogonal domain boundaries, consistent closure, independent boundaries, linear boundaries, and other characteristics. We know that which makes domain testing difficult. That is it consists of applying algebra to the problem.

(ii) Linearizing Transformations:

- This is used to transfer non linear boundaries to equivalent linear boundaries.
- The different methods used here are

(i)Polynomials:

- A boundary is specified by a polynomial or multinomial in several variables.
- For a polynomial each term can be replaced by a new variable.
- i.e. x, x^2, x^3, …can be replaced by $y_1 = x$, $y_2 = x^2$, $y_3 = x^3$, …
- For multinomials you add more new variables for terms such as xy, x^2y, xy^2, …
- So polynomial plays an important role in linear transformations.

(ii)Logarithmic Transforms:

- Products such as xyz can be linearized by substituting $u = \log(x)$, $v = \log(y)$, $\log(z)$.
- The original predicate $xyz > 17$ now becomes $u + v + w > 2.83$.

(iii)More general forms:

- Apart from logarithmic transform & polynomials there are general linearizable forms such as $x / (a + b)$ and ax^b. We can also linearize by using Taylor series.

(iii) Coordinate Transformations:

- The main purpose of coordinate transformation technique is to convert Parallel boundary inequalities into non parallel boundary inequalities and Non-parallel boundary inequalities into orthogonal boundary inequalities.

(iv) A Canonical Program Form:

- Testing is clearly divided into testing the predicate and coordinate transformations.
- i.e. testing the individual case selections, testing the control flow and then testing the case processing..

(v) Great Insights:

- Sometimes programmers have great insights into programming problems that result in much simpler programs than one might have expected.

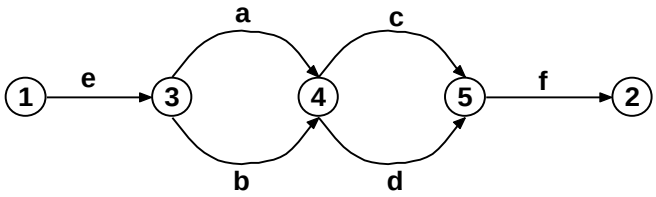
PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

(1) Path products & path expression:

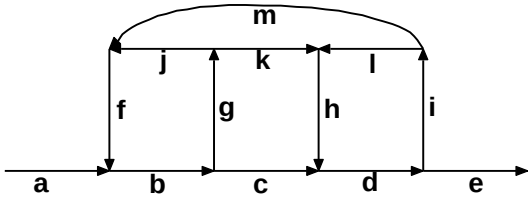
(1) Explain Paths, Path products, Path expressions, path sums and loops?

(a) Paths:

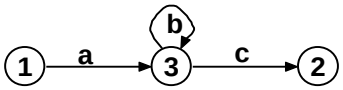
- A sequence of statements which starts at an entry and ends at an exit and passes all the decisions, junctions & processes is known as path.
- A path represents different links and we can give a simplest weight to a link is a name.
- Using link names, we can convert the graphical flowgraph into an equivalent algebraic expression.
- The link name will be denoted by lower case italic letters.
- In traversing a path, we traverse link names that give the name of the path.
- If you traverse links a, b, c, d then the name for that path is abcd.
- This path name is also called a path product. The following are some examples of paths.



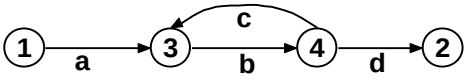
The different paths are: eacf, eadf, ebcf, ebdf



The different paths are: abcde, abgjfbcde, abcdimfbcde



The different paths are: ac, abc, abbc, abbbc, abbbbc



The different paths are: abd, abcbd, abcbcbd, abcbcbcbd

(b) Path Products:

- The concatenation of names of two consecutive path segments is called a path product.
- For example if X and Y are defined as X = abcde and Y = fghij then

  XY = abcdefghij YX = fghijabcde

  aX = aabcde Xa = abcdea XaX = abcdeaabcde .
- Another example is if X = abc + def + ghi and Y = uvw + z then

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

  If X = abcde then X^1 = abcde

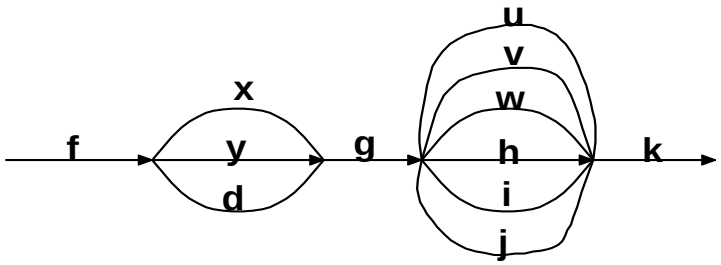
  $X^2 = (abcde)^2$ = abcdeabcde

  ...
- The path product is not commutative that is XY does not necessarily equal to YX.

- The path product is associative that is (XY)Z = X(YZ).

(c) Path expression:

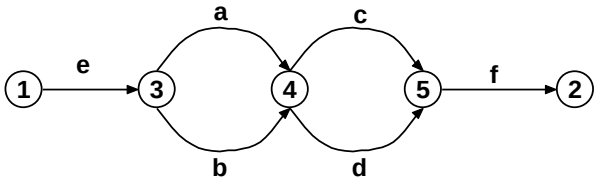
- Path expression is defined as an expression which represents set of all possible paths between an entry and exit nodes. For example:



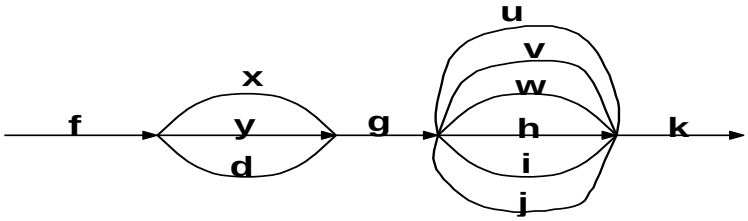
- The path expression to the above figure is: f (x + y + d) g (u + v + w + h + i + j) k

(d) Path sums:

- The path sum is the sum of all the parallel links between two nodes or sum of all parallel paths between two nodes. Path sum is denoted by '+'.
- Ex (i)



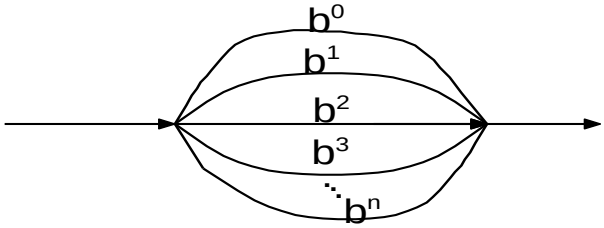
- In the above figure, links a & b are parallel, so these parallel paths are denoted by a + b.
- Similarly c and d are parallel & these parallel paths are denoted by c + d.
- The set of parallel paths between 1 and 2 nodes are eacf + eadf + ebcf + ebdf.
- Ex (ii)



- The first set of parallel path is denoted by X + Y + d and second by u + v + w + h + i + j.
- The set of all paths in this flowgraph is f (X + Y + d) g(u + v + w + h + i + j) k
- Path sum is commutative and associative. Commutative is X + Y = Y + X
  Associative is (X+Y)+Z=X+(Y+Z)

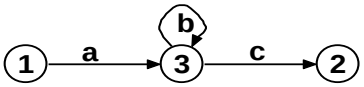
(e) Loops:

- If a single link or path expression is traversed indefinite no of times leading to infinite no of parallel paths then it is called a loop. For example the loop consists of a single link b, then the set of all paths through that loop is $b^0 + b^1 + b^2 + \ldots b^n$



- This infinite sum is denoted by b*. So $b^* = b^0 + b^1 + b^2 + \ldots b^n$.
- If the loop is taken at least once then it is denoted by b^+.

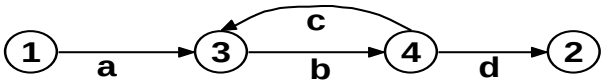
- Ex (i)



The path expression is: $ab^*c = a(b^0)c + a(b^1)c + a(b^2)c + a(b^3)c + \ldots\ldots$

$= ac + abc + a\,bbc + a\,bbbc + \ldots\ldots$

Ex (ii)



The path expression is: $a(bc)^*bd = a(bc)bd + a(bc)bd + a(bc)bd + \ldots\ldots$

$= abd + abcbd + abcbcbd + \ldots$

(2) Discuss all the rules in path representation of graphs?

Rule 1:

$A(BC)=(AB)C=ABC$

Rule 2:

$X + Y = Y + X$

Rule 3:

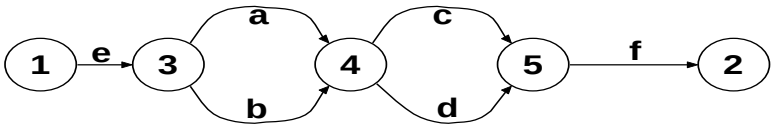
$(X + Y) + Z = X + (Y + Z) = X + Y + Z$

$A(BC)=(AB)C=ABC$

Rule 4:

- Distributive laws are $A(B+C) = AB + AC$

  $(B + C)\,D = BD + CD.$
- For example:



$e(a+b)(c+d)f = e(ac+ad+bc+bd)f = eacf + eadf + ebcf + ebdf$

Rule 5:

- The absorption rule is, if X and Y denote the same set paths, then the union of these sets is not changed. Ex: $X + X = X$.
- Another example is: if $X = a + aa + abc + abcd + def$ then

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

Rule 6:

$X^n + X^m = X^n$ if n is bigger than m

$= X^m$ if *m* is bigger than *n*

Rule 7:

$X^nX^m = X^{n+m}$

Rule 8:

$X^nX^* = X^*X^n = X^*$

Rule 9:

$X^nX^+ = X^+X^n = X^+$

Rule 10:

$X^*X^+ = X^+X^* = X^+$

Identity elements:(Rule 11 to Rule 17)

- a^0, X^0 denote the path whose length is zero. The rules are

Rule 11:

$1 + 1 = 1$

Rule 12:

$1X = X1 = X$

Rule 13:

$1^n = 1^{\underline{n}} = 1^* = 1^+ = 1$

Rule 14:

$1^+ + 1 = 1^* = 1$

Rule 15:

$X + 0 = 0 + X = X$

Rule 16:

$X0 = 0X = 0$

Rule 17:

$0^* = 1 + 0^1 + 0^2 + \ldots = 1$

(2) A Reduction Procedure:

(1) Write the steps involved in Node Reduction Procedure. Illustrate all the steps with the help of neat labeled diagrams?

Node Reduction Procedure:

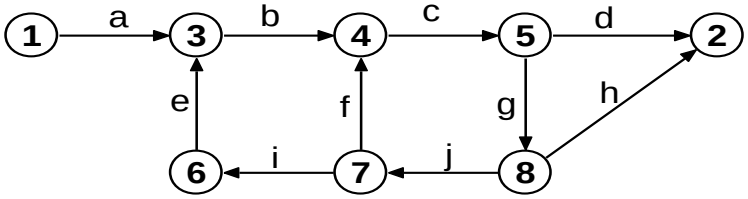
- The main aim of Node Reduction Procedure is to remove all the intermediate nodes between entry and exit nodes. This procedure is helpful in debugging process. i.e. Instead of gathering information about path expression of all the intermediate nodes for debugging; it is easy to debug only the path expression between entry and exit nodes.

Procedure:

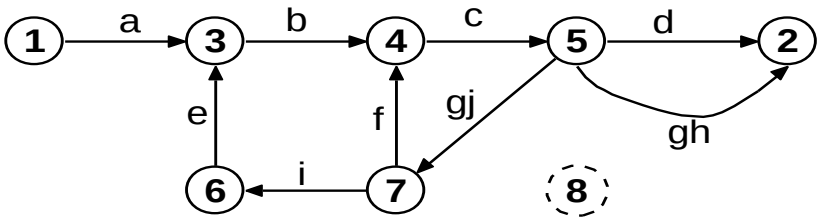
1. Combine all serial links by multiplying their path expressions.
2. Combine all parallel links by adding their path expressions.
3. Remove all self loops by replacing them with a link of the form x^*, where x is the path expression of the link in that loop.
4. Choose the node which is to be removed other than initial and final node. The path expression of the inlink and outlink of this node is multiplied and a direct link is applied with the product of path expression. This step-4 is called Cross-Term Step.
5. Combine any remaining serial links by multiplying their path expressions.
6. Combine all parallel links by adding their path expressions. This Step-6 is called Parallel Term Step.
7. Remove all self-loops as in step 3. This Step-7 is called Loop Term Step.
8. If the graph consists of a single link between the entry and the exit node, then the path expression for that link is a required path expression. Otherwise return to step 4.

Example:

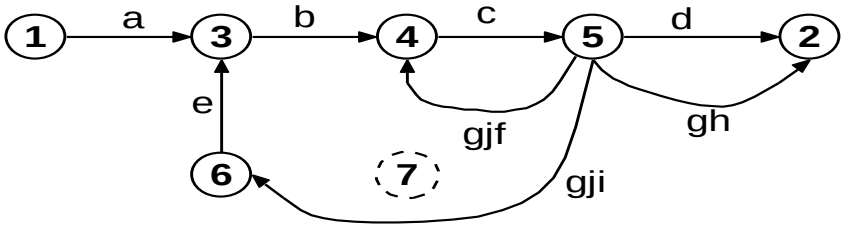
- Consider the following graph.



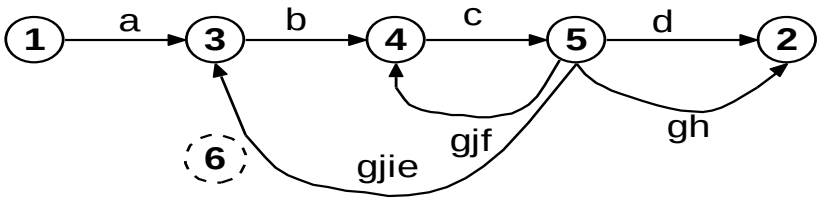
- First remove node 8 by applying step 4 (cross-term step) and combine by step 5.



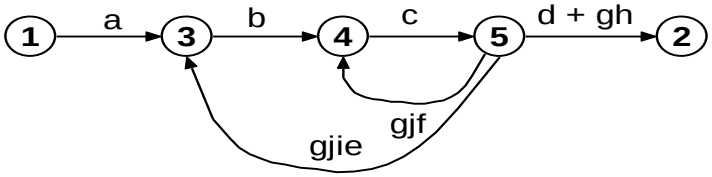
- Remove node 7 by applying step 4 (cross-term step) and combine by step 5.



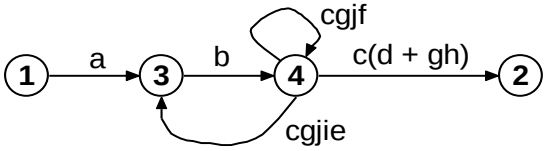
- Remove node 6 by applying step 4 (cross-term step) and combine by step 5.



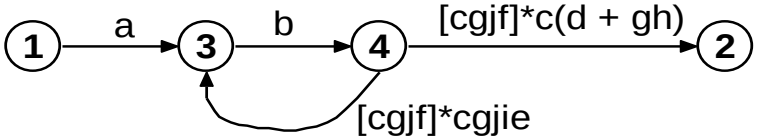
- Add parallel links between node 5 and node 2 by applying parallel term step.



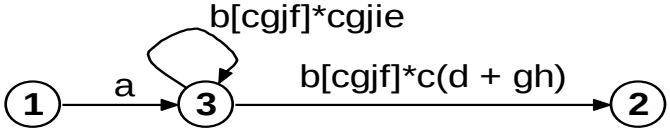
- Remove node 5 by applying step 4 (cross-term step) and combine by step 5.



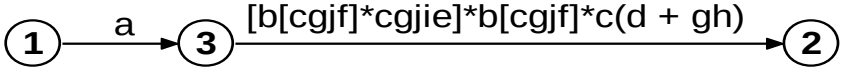
- Remove self loop at node 4 by applying loop term step.



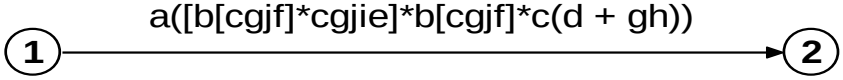
- Remove node 4by applying step 4 (cross-term step) and combine by step 5.



- Remove self loop at node 3 by applying loop term step.



- Remove node 3 by applying step 4.



(3) Applications:

(1) How many paths in a Flowgraph:

Q. Explain maximum path count arithmetic of a flowgraph with an example?

Maximum Path Count Arithmetic:

- Here each link is represented by a link weight. There are three arithmetic cases that are considered here.
- They are

- **(i) Parallel rule:**
  - Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

(ii) Series rule:

- Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there are W_A paths in A and W_B paths in B then there are $W_A W_B$ paths in its combination.

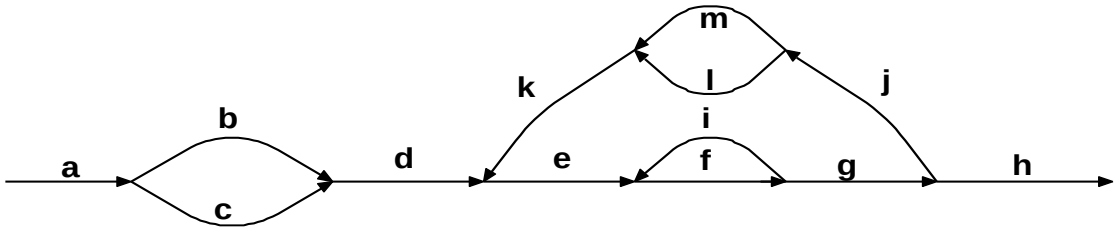
(iii) Loop rule:

- Loop rule is evaluated by considering number of times that the path is iterated.

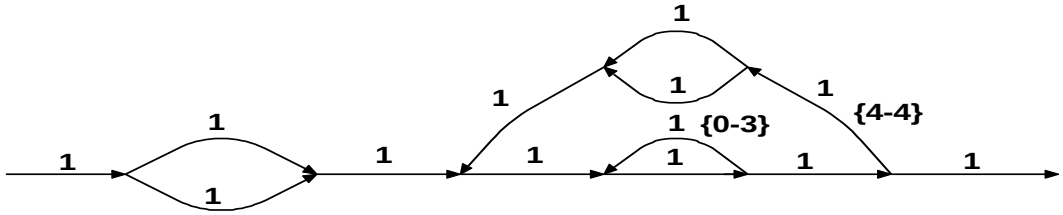
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	A^n	$\sum_{i=0}^{n} W_A{}^i$

Example:

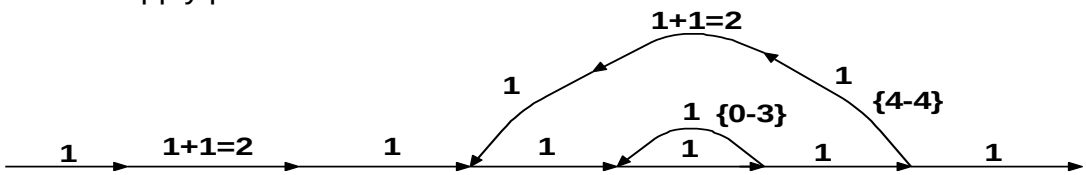
- Determine the path expression to the following figure.



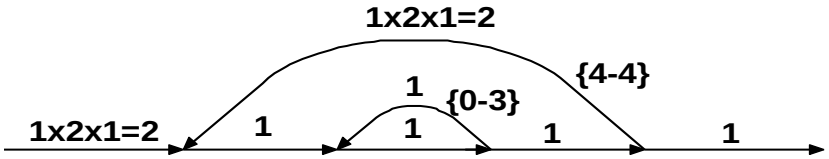
- The path expression is given by
  a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- Let each link represents a single link and is given by a link weight 1.
- Assume that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times.
- The reduction is as follows.



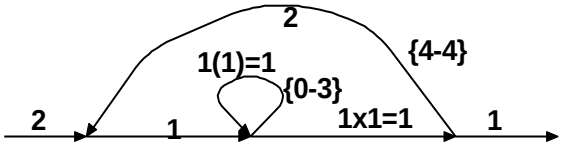
- Now apply parallel rule.

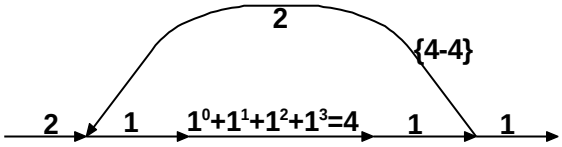


- Now apply series rule.

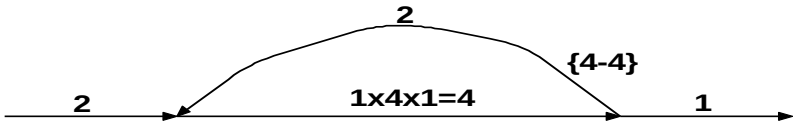


- Now create inner self loop & Apply loop rule for removing inner self loop.

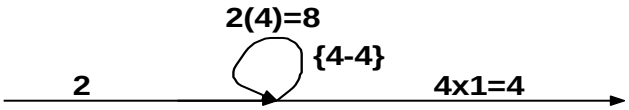




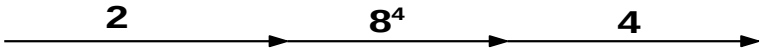
- Now apply series rule.



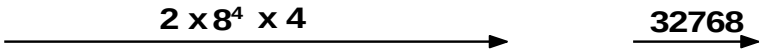
- Now create outer self loop.



- Apply loop rule to remove the self loop.



- Apply series rule.



- Alternatively we can calculate the maximum number of paths as follows.
- The path expression is given by
  a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- In the above expression each link is substituted by 1.
  $1(1+1)1[1(1x1)^3\ 1x1x1\ (1+1)1]^4\ 1(1x1)^3\ 1x1x1$
  $=1(2)[1^3 \times 2]^4\ 1x1^3$
  $=2[4x2]^4 \times 4$ [since $1^3 = 1^0 + 1^1 + 1^2 + 1^3 = 4$]
  $=2 \times [8]^4 \times 4 = 32{,}768$..

(2) Approximate Minimum number of paths:

Q. Define structured code. Explain about lower path count arithmetic?

Structured code:

- A structured flowgraph is one that can be reduced to a single link by successive application of transformations.
- Based on the path expression obtained by node-by-node reduction procedure we can determine whether the given flow graph is a structured or unstructured.
- That is if the resultant expression is large and ugly then the graph is unstructured one otherwise the graph is structured one.

Lower path count arithmetic:

- The lowest number of paths in a structured flowgraph can be approximately known; it may or may not be accurate because there is every possibility of a path being unachievable which further lowers the number count.
- Here each link is represented by a link weight. Loops are always problematic.

- So it must be traversed only one time or zero times to achieve the coverage. There are three arithmetic cases here. They are.
- **(i) Parallel rule:**
  - Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

(ii) Series rule:

- Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB.
- If there are W_A paths in A and W_B paths in B then there are MAX (W_A, W_B) paths in its combination.

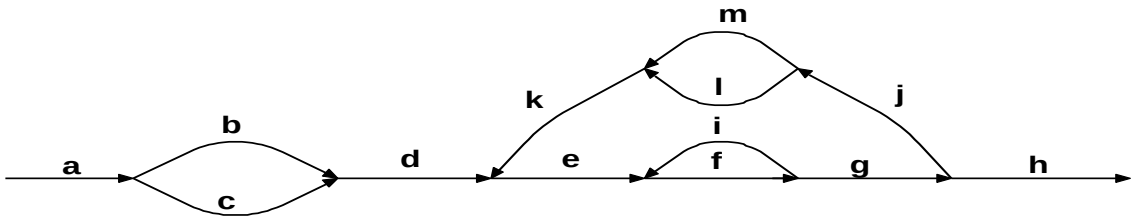
(iii) Loop rule:

- Loop rule is taken either by considering only one time that the path is iterated or zero times the path is iterated. So it gives the value 1 or its link weight.

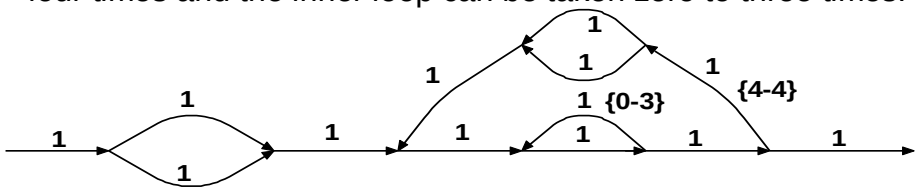
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	MAX(W_A,W_B)
LOOP	A^n	1,W_1

Example:

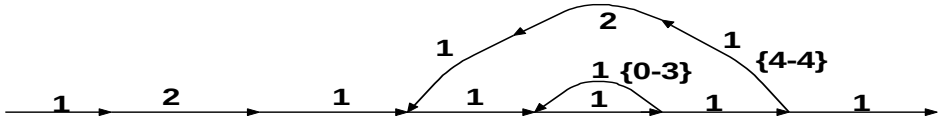
- Determine the path expression to the following figure.



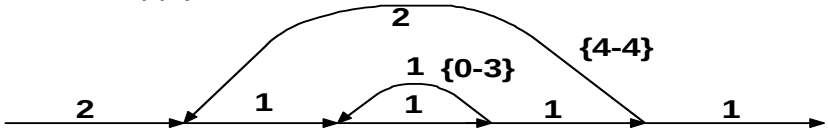
- The path expression is given by a(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- Let each link represents by a link weight 1. Assume that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times. The reduction is as follows.



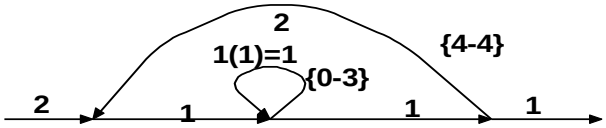
- Now apply parallel rule.

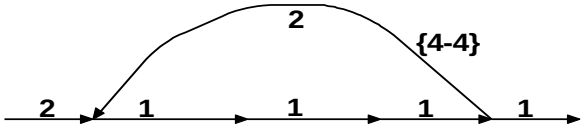


- Now apply series rule.

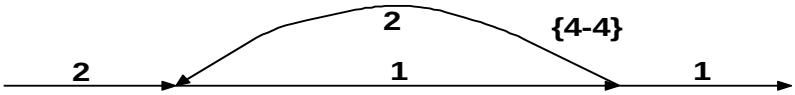


- Now create inner self loop & apply loop rule for removing inner self loop.

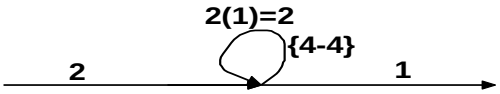




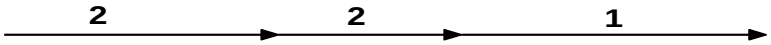
- Now apply series rule.



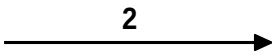
- Now create outer self loop.



- Apply loop rule to remove self loop.



- Apply series rule.



- Alternatively we can calculate the minimum number of paths as follows.
- The path expression is given bya(b +c) d [e(fi)*fgj(m + l)k]*e(fi)*fgh
- In the above expression each link is substituted by 1.

  $1(1+1)1[1(1x1)^0\ 1x1x1\ (1+1)1]^0\ 1(1x1)^0\ 1x1x1$

  $=1(2)[1^0 \times 2]^0\ 1x1^0 = 2x1 = 2$

(3) The probability of getting there:

Q. What is the probability of path expressions? Write arithmetic rules. Explain with an example.

Probability of path expressions:

- Specify each out link of a node equal to the probability of that link. The sum of the out link probabilities is equal to 1. For a simple loop, if the loop is taken N times then the looping probability is N/(N+1) and non looping probability is 1/(N+1).
- There are three arithmetic cases here. They are

Parallel rule:

- Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B.
- If there is a path expression A with Probability P_A and path expression B with Probability P_B then the resultant probability is $P_A + P_B$.

Series rule:

- Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there is a path expression A with Probability P_A and path expression B with Probability P_B then the resultant probability is $P_A P_B$

Loop rule:

- If the probability of looping node is P_L and the probability of link leaving the loop node is P_A then $P_A + P_L=1$. So $P_A = 1- P_L$

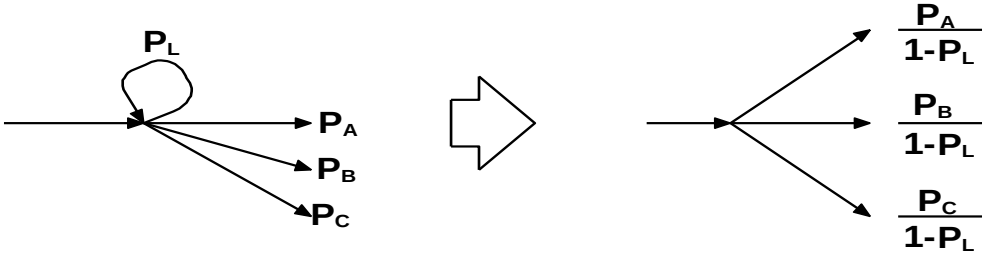
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$P_A + P_B$
SERIES	AB	$P_A P_B$
LOOP	A^n	$P_A/(1-P_L)$

Example (i)

$P_A = 1 - P_L$

New Probability $P_{NEW} = P_A / (1-P_L) = (1-P_L) / (1-P_L) = 1$

Example (ii)



Here $P_L + P_A + P_B + P_C = 1$

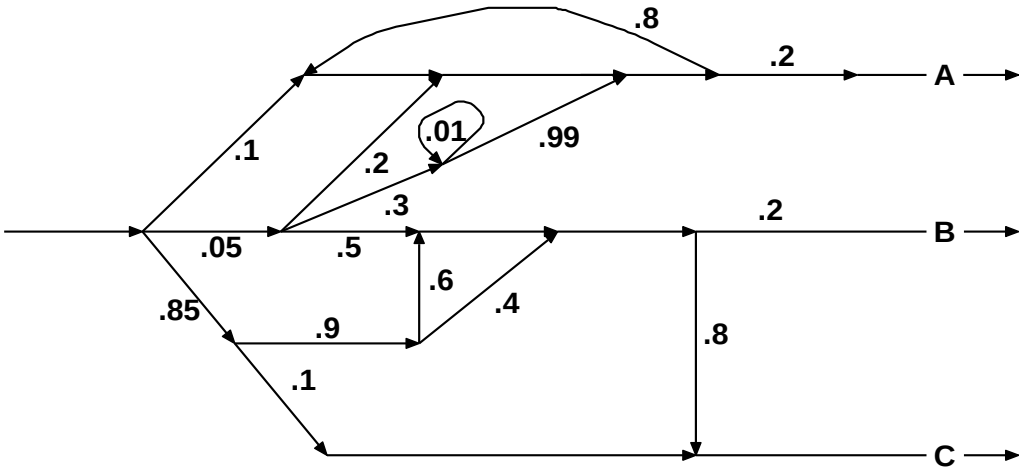
$1 - P_L = P_A + P_B + P_C$

$P_A / (1 - P_L) + P_B / (1 - P_L) + P_C / (1 - P_L) = (P_A + P_B + P_C) / (1 - P_L)$

$= (P_A + P_B + P_C) / (P_A + P_B + P_C) = 1$

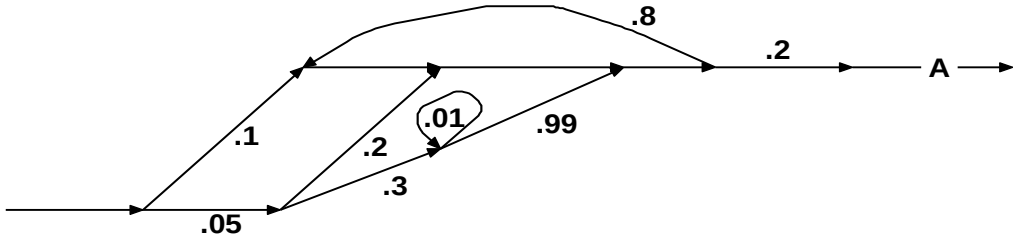
Example:

- Consider the following flowgraph.

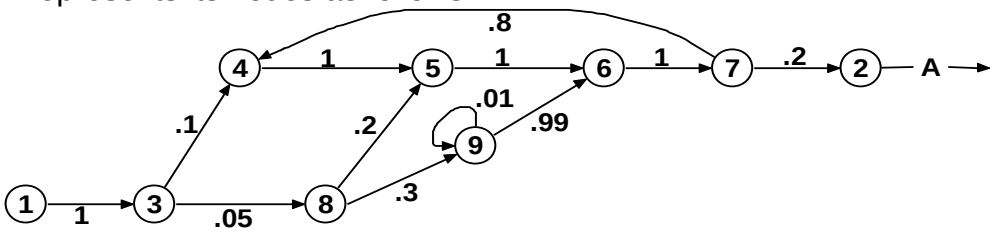


- Calculate the probabilities of cases A, B, C.

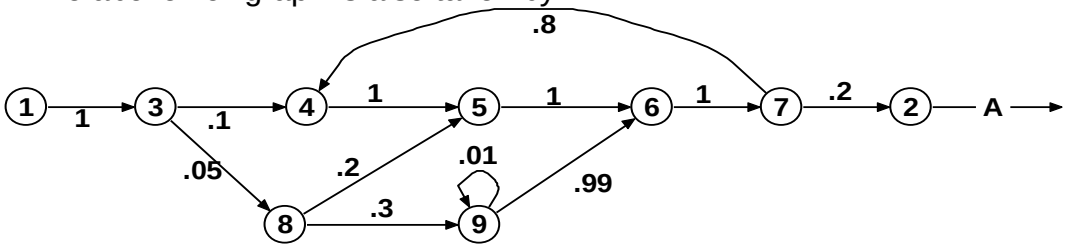
First consider case A:



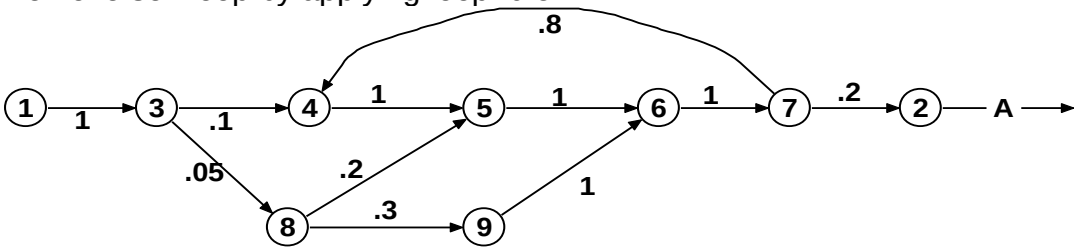
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



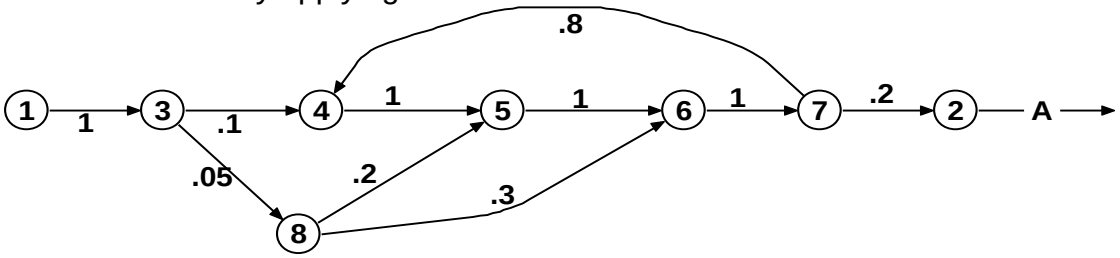
- The above flowgraph is also taken by



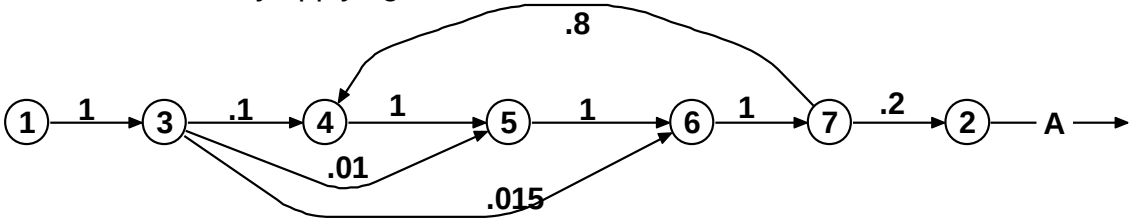
- Remove self loop by applying loop rule



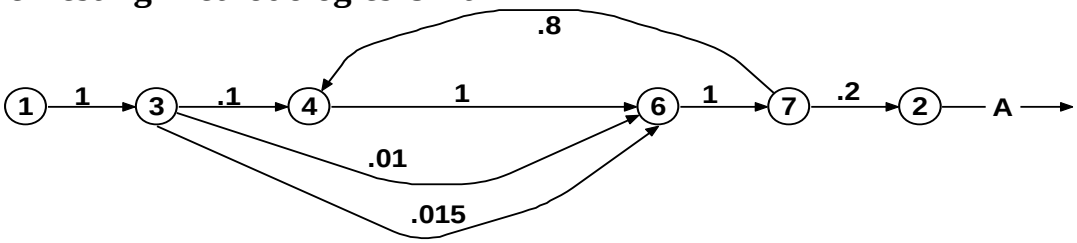
- Remove node 9 by applying series rule



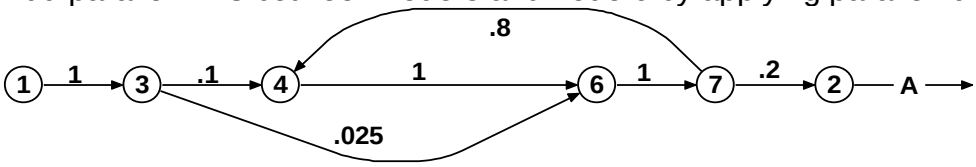
- Remove node 8 by applying series rule



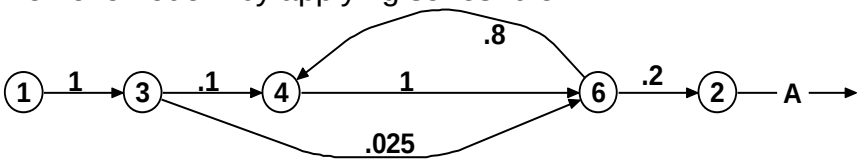
- Remove node 5 by applying series rule



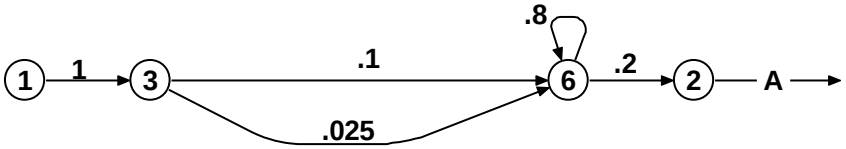
- Add parallel links between node 3 and node 6 by applying parallel rule



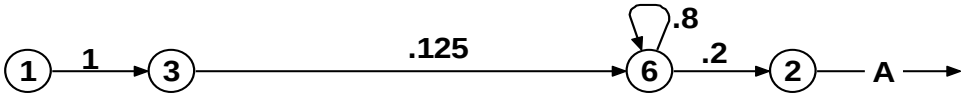
- Remove node 7 by applying series rule



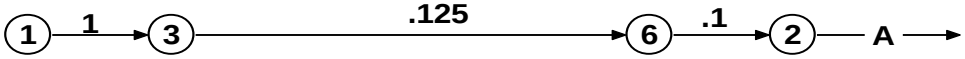
- Remove node 4 by applying series rule



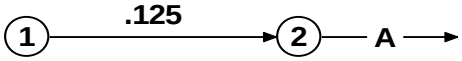
- Add parallel links between node 3 and node 6 by applying parallel rule



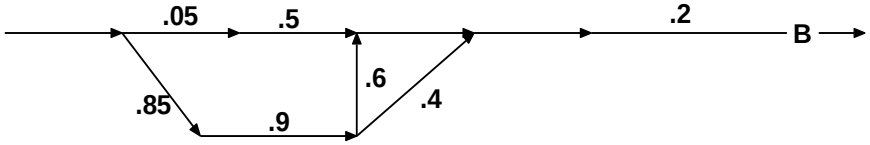
- Remove self loop at node 6 by applying loop rule



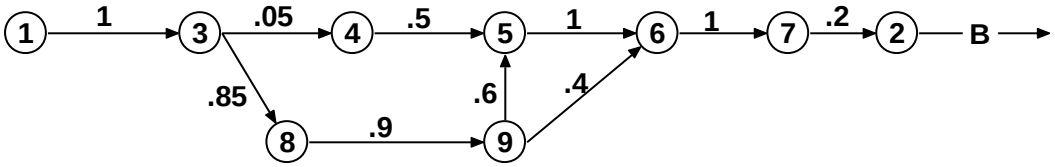
- Remove node 3 and node 6 by applying loop rule



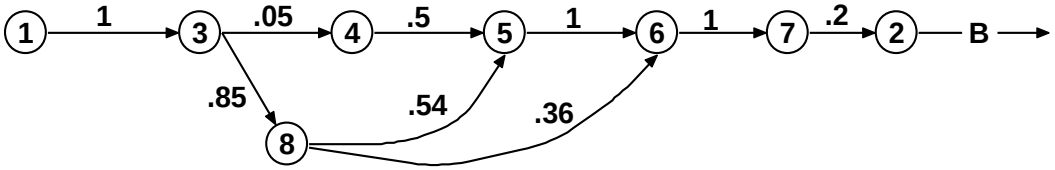
Consider case B:



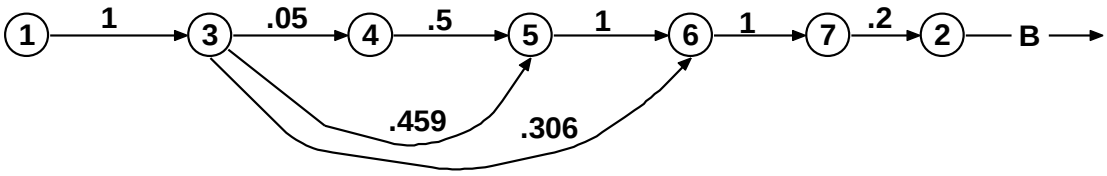
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



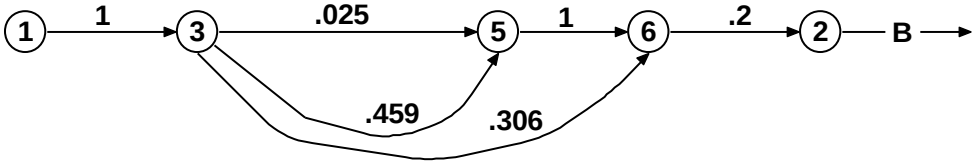
- Remove node 9 by applying series rule.



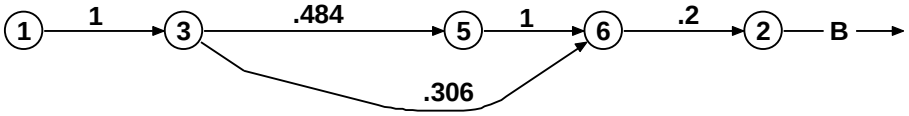
- Remove node 8 by applying series rule.



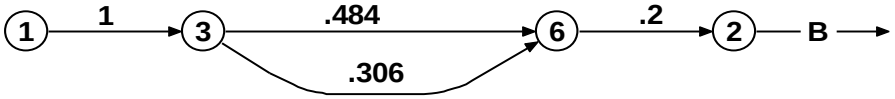
- Remove node 4 and node 7 by applying series rule.



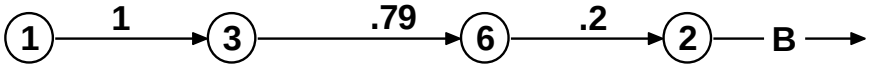
- Add parallel links between node 3 and node 5 by applying parallel rule



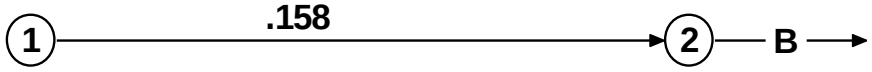
- Remove node 5 by applying series rule.



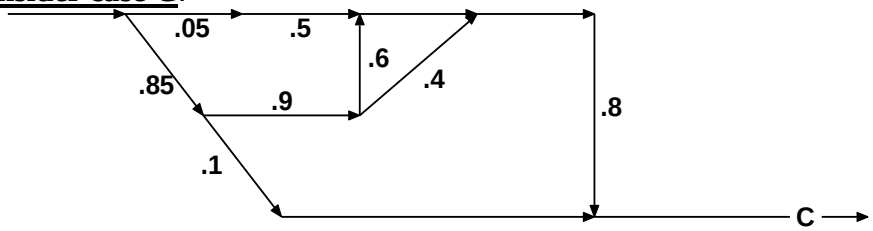
- Add parallel links between node 3 and node 5 by applying parallel rule



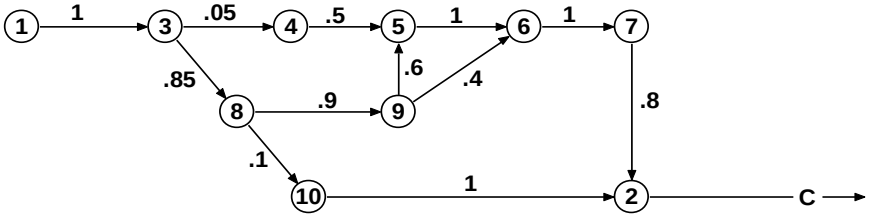
- Remove node 5 by applying series rule.



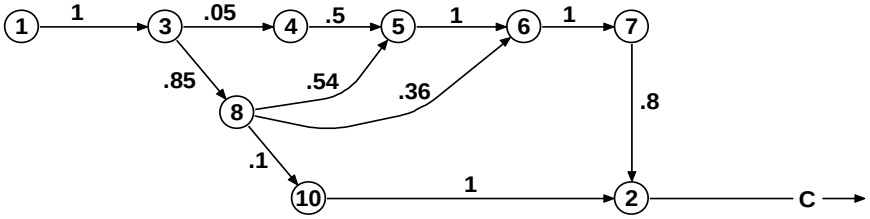
Consider case C.



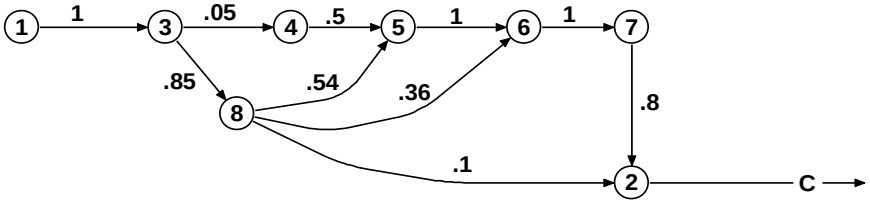
- In the above flowgraph if the link weight is not specified then it is specified by 1 and also represents its nodes as follows.



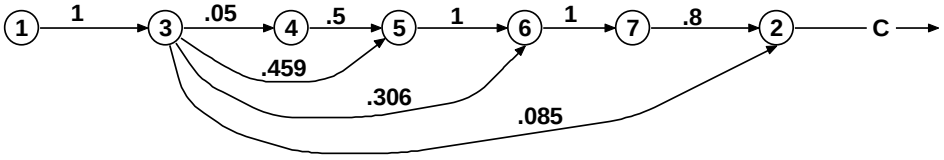
- Remove node 9 by applying series rule.



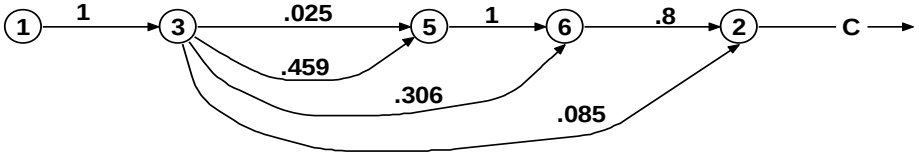
- Remove node 10 by applying series rule.



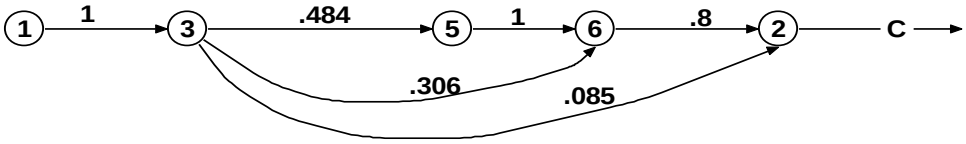
- Remove node 8 by applying series rule.



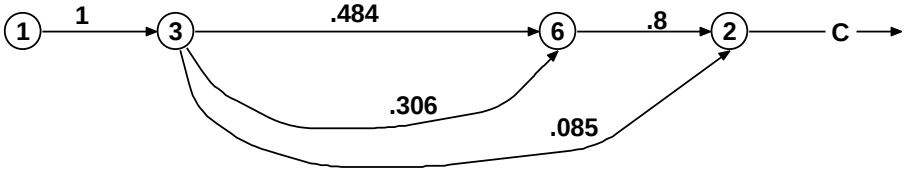
- Remove node 7 & node 4 by applying series rule.



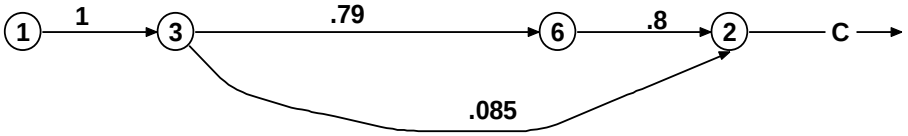
- Add parallel links between node 3 and node 5 by applying parallel rule



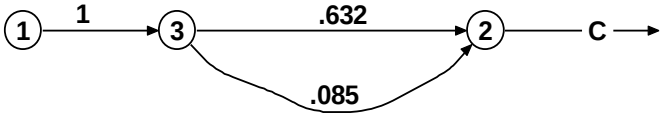
- Remove node 5 by applying series rule



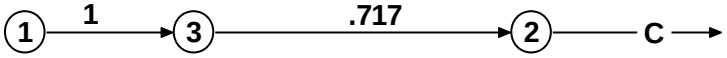
- Add parallel links between node 3 and node 6 by applying parallel rule



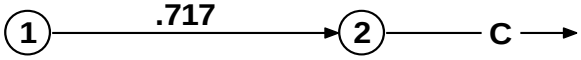
- Remove node 6 by applying series rule



- Add parallel links between node 3 and node 2 by applying parallel rule



- Remove node 3 by applying series rule



Cross check:

- Sum of case A + case B + case C = .125 + .158 + .717 =1.

(4) The mean processing time of a routine

Q. What is the mean processing time of a routine? Write arithmetic rules. Explain with an example.

Mean processing time of a routine:

- Here every link has two weights.

- One is the processing time for that link denoted by T, & other one is the probability of that link denoted by P.
- There are three arithmetic cases here.
- They are

Parallel rule:

- It is the arithmetic mean of all processing time over all parallel links.

Series rule:

- It is the sum of two processing times.

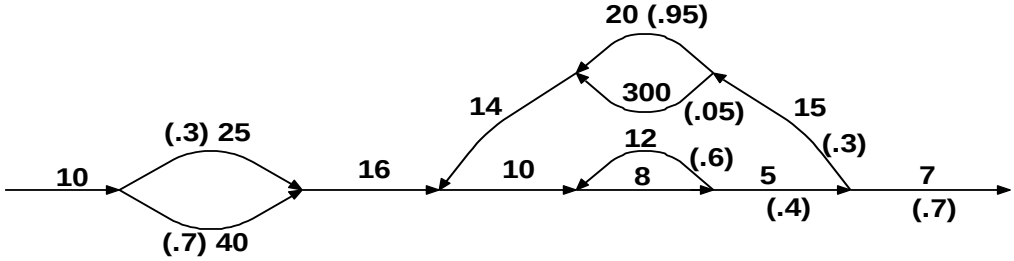
Loop rule:

- It is evaluated by considering number of times the path is iterated

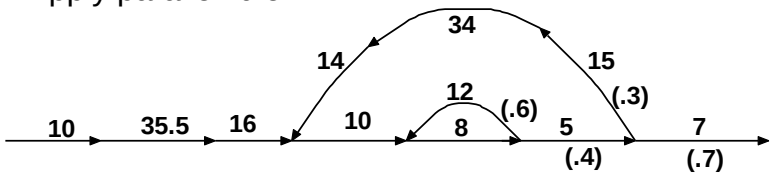
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$T_{A+B} = (P_A T_A + P_B T_B)/(P_A + P_B)$ $P_{A+B} = P_A + P_B$
SERIES	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
LOOP	A^*	$T_A = (T_L P_L)/(1-P_L) + T_A$ $P_A = P_A/(1-P_L)$

Example:

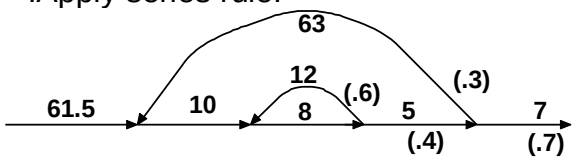
- The following figure is represented by, loop probabilities, and processing time for each link. The probabilities are given in parentheses.



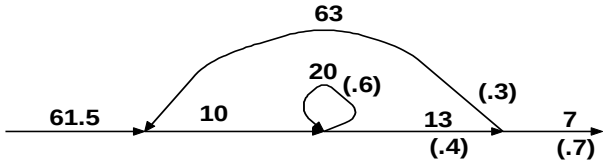
- Apply parallel rule.



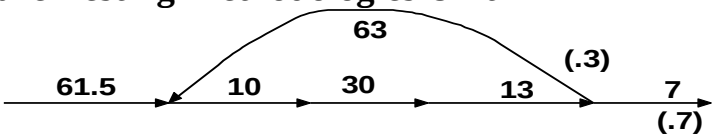
- .Apply series rule.



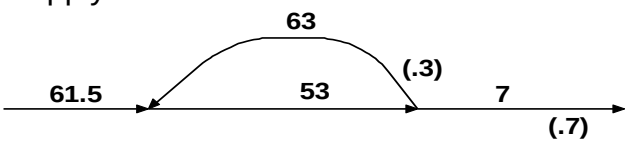
- Now create inner self loop.



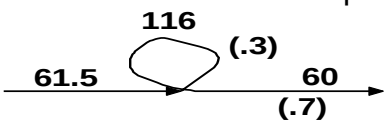
- Remove the inner self loop by applying loop rule.



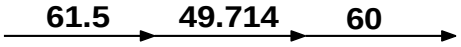
- Apply series rule.

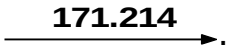


- Create the outer self loop.



- Remove the outer self loop by applying loop rule.



- Apply series rule
- 

(5) Push/Pop, Get/Return

Q. What is Push/Pop, Get/Return? Write arithmetic rules. Explain with an example.

Push/Pop:

- Here PUSH operation is used to insert elements into the stack. POP operation is used to remove elements from the stack.
- Apart from PUSH/POP other operations are GET/RETURN, OPEN/CLOSE and START/STOP.
- There are three arithmetic cases here.
- They are

Parallel rule:

- Each term of the path expression A is added with each term of the path expression B if there are two path expressions A and B. So it is A+B. If there are W_A paths in A and W_B paths in B then there are $W_A + W_B$ paths in its combination.

Series rule:

- Each term of the path expression A is multiplied with each term of the path expression B if there are two path expressions A and B. So it is AB. If there are W_A paths in A and W_B paths in B then there are $W_A W_B$ paths in its combination.

Loop rule:

- It is evaluated by considering number of times the path is iterated.

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	A + B	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	A^*	W^*_A

- PUSH/POP operations satisfy commutative, associative, and distributive law of addition and multiplication.
- The arithmetic tables for PUSH/POP are given by

PUSH/POP MULTIPLICATION TABLE

X	H	P	1
H	H^2	1	H
P	1	P^2	P
1	H	P	1

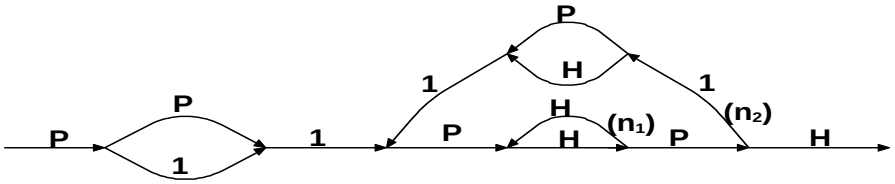
PUSH/POP ADDITION TABLE

+	H	P	1
H	H	P+H	H+1
P	P+H	P	P+1
1	H+1	P+1	1

- These tables are used to determine the weight of addition and multiplication operation.
- Here H represents the PUSH operation, P represents the POP operation and 1 represents NO operation.

Example:

- Consider the following flowgraph.



- Path expression for the above flowgraph is.

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

- Simplifying by using the arithmetic tables

  $PUSH/POP = (P^2 + P)[P(HH)^{n1}(P+H)]^{n2}(HH)^{n1}$

  $= (P^2+P)[H^{2n1}(P^2+1)]^{n2}H^{2n1}$

- Let us consider M_1,M_2 represents the two looping terms. i.e. M_1 represents the number of times the inner loop is considered, M_2 represents the number of times the outer loop is considered.

CASE (i)

Consider $M_1=0$, $M_2=0$ (i.e. $n_1=0$, $n_2=0$)

$PUSH/POP= (P+P^2)[H^0(P^2+1)]^0H^0 = P + P^2$

CASE (ii)

Consider $M_1=0$, $M_2=1$ (i.e. $n_1=0$, $n_2=1$)

$PUSH/POP= (P+P^2)[H^0(P^2+1)]^1H^0$

$= (P + P^2)[1+P^2] = P + P^2 + P^3 + P^4$

- For different combination of M_1, M_2 values the following table is obtained.

M_1	0	0	0	0	1	1	1	1	2	2	2	2
M_2	0	1	2	3	0	1	2	3	0	1	2	3
PUSH /POP	$P + P^2$	$P + P^2 + P^3 + P^4$	6 $\sum P^i$ 1	8 $\sum P^i$ 1	1+H	3 $\sum H^i$ 0	5 $\sum H^i$ 0	7 $\sum H^i$ 0	H^2+H^3	7 $\sum H^i$ 4	11 $\sum H^i$ 6	16 $\sum H^i$ 8

Get/Return:

- The arithmetic tables for GET/RETURN are.

GET/RETURN MULTIPLICATION TABLE

×	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

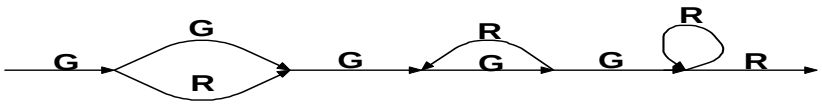
GET/RETURN ADDITION TABLE

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

- The arithmetic table for GET/RETURN is same as that of PUSH/POP.

Example:

- Consider the following flowgraph.



- Path expression for the above flowgraph is. G(G+R) G(GR)* GGR* R
- Simplifying by using the arithmetic tables

  GET/RETURN = $G(G+R)G^3 R^*R$

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

(6) Limitations and Solutions

Q. What are the limitations and solutions of the applications?

- 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 achievable, but may provide misleading results when some paths are unachievable.
- The solution to handling unachievable paths is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable. But the resulting sub graphs may overlap, because one path may be common to several different subgraphs.
- Each predicate's truth value splits the graph into two subgraphs.
- For n predicates there may be 2^n sub graphs. Here there is an algorithm for one predicate.

  1. Set the value of the predicate to TRUE and strike out all FALSE links for that predicate.

  2. Discard any node, other than an entry or exit node, that has no incoming links. Discard all links that leave such nodes. If there is no exit node, the routine has a bug because there is a predicate value that forces an endless loop or the equivalent.

  3. Repeat step 2 until there are no more links or nodes to discard. The resulting graph is the subgraph corresponding to a TRUE predicate value.

  4. Change "TRUE" to "FALSE" in the above steps and repeat. The resulting graph is the subgraph that corresponds to a FALSE predicate value.
- Only correlated predicates should be included in this analysis not all predicates that may control the program flow.

(4) Regular expressions and flow anomaly detection:

Q. Explain about Regular expression and Flow-Anomaly detection?

(i) The Problem:

- The generic flow-anomaly detection problem is used to search for a specific sequence of operations considering all possible paths through a routine.
- Let's say the operations are SET and RESET, denoted by *s* and *r* respectively, and we want to know if there is a SET followed immediately by a SET or a RESET followed immediately by a RESET (i.e, an *ss* or an *rr* sequence).

- Flow anomaly detection is used to know if particular sequence occurred, but not to know the total impact of the procedure.
- It is used to detect the bug sequence in the following situations.

  1. A file can be opened (*o*), closed (*c*), read (*r*), or written (*w*). If the file is read or written to after it is closed, then it is anomalous. i.e. *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.

  2. The operations performed by tape transport device are read(r), write(w), rewind (*d*), forward (*f*), skip (*k*) and stop (p). In a tape-transport device rewind and forward operations cannot be performed one after the other without performing stop operation. So the following sequences are anomalous: *df, dr, dw, fd*, and *fr*.

  3. With the help of generic flow anomaly detection, it is possible to detect the data flow bugs sequence such as *dd, dk, kk*, and *ku.*

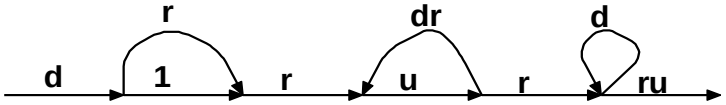
  4. A bug that occur only if two operations a and b occurred in the order aba or bab.

(ii) Huang Theorem:

- Annotate each link in the graph with the appropriate operator or the null operator 1.
- Simplify things using $a + a = a$ and $1^2 = 1$.
- The regular expression obtained should be simplified carefully, as null operations cannot be combined with other operations.
- For example, 1*a* may not be the same thing as a alone. Huang theorem is used to simplify the regular expression and to examine the specific operation sequence.
  - Let A, B, C, be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters.
  - Then if T is a substring of AB^nC, then T will appear in AB^2C.
  - As an example, let A = *pp* B = *srr* C = *rp* T = *ss*
  - The theorem states that if *ss* is a substring of $pp(srr)^nrp$ then ss will appear in $pp(srr)^2rp$.
  - Similarly let A = *p* + *pp* + *ps* B = *psr* + *ps*(*r* + *ps*) C = *rp* T = P^4
  - If p^4 is a substring of AB^nC then p^4 will appear in AB^2C $(p + pp + ps)[psr + ps(r + ps)]^2rp$
- Huang theorem is also useful in test design.
- Further Huang shows that if you substitute $1 + X^2$ for every expression of the form X*, the paths that result from this substitution are sufficient to determine whether a given two-character sequence exists or not.
- Two character string sequences are used to represent data flow anomaly. Then using Huang's theorem these anomalous can be detected if these loop is iterated twice.

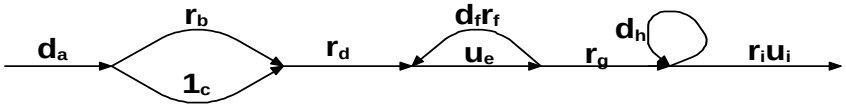
Data Flow Testing Example:

- By assigning appropriate operators on each link the following flowgraph can be used to detect different anomalies bugs.



- Huang's theorem states that the following expression is sufficient to detect any two character sequence. $d(r + 1)r[1 + (udr)^2]ur(1 + d^2)ru$
- This makes the dd bug obvious. A *kk* bug cannot occur and also a *dk* bug cannot occur.

  $(drr + dr)(1 + udrudr)(urru + urd^2ru)$
- A better way to the above is subscript the operator with the link name.



- The regular expression is $d_a(r_b + 1_c)r_d(u_e d_f r_f)^* u_e r_g d_h^* r_i u_i$

- Applying Huang's theorem:

  $d_a(r_b + 1_c)r_d(1 + (u_e d_f r_f)^2)u_e r_g(1 + d^2_h)r_i u_i$

  $(d_a r_b r_d + d_{ac} r_d)(u_e r_g + u_e d_f r_f u_e d_f r_f u_e r_g)(r_i u_i d^2_h r_i u_i)$

(iii) Generalizations, Limitations and comments:

- Huang's theorem can be easily generalized to cover sequences of greater length than two characters. If A, B, and C are nonempty sets of strings of one or more characters, and if T is a string of *k* characters, and if T is a substring of AB^nC, where *n* is greater than or equal to *k*, then T is a substring of AB^kC.
- A sufficient test for strings of length *k* can be obtained by substituting $P^{\underline{k}}$ for every appearance of P*

  $P^{\underline{k}} = 1 + P + P^2 + P^3 + \ldots + P^k$
- In order to find the starting and ending sequence of strings in a path expression, the mathematical approaches such as application of derivations to algebraic expression makes it easier and time consuming than the path tracing process on a flowgraph.
- Static flow analysis methods can't determine whether a path is achievable or is not achievable.
- If unachievable paths exist, then the exactness and applicability of all flow analysis methods reduces gradually. Hence achievable paths are preferred in order to overcome the problems of unachievable paths.

UNIT –IV
SYNTAX TESTING

1. WHY, WHAT, AND HOW:

1.1. Garbage

- "Garbage-in equals garbage-out" is one of the worst cop-outs ever invented by the computer industry.
- We know when to use that one! When our program screws up and people are hurt.
- An investigation is launched and it's discovered that an operator made a mistake, the wrong tape was mounted, or the source data were inconsistent, or something like that.
- That's the time to put on the guru's mantle, shake your head, disclaim guilt, and mutter, "What do you expect? Garbage-in equals garbage-out."
- Can we say that to the families of the airliner crash victims? Will you offer that excuse for the failure of the intensive care unit's monitoring system?
- How about a nuclear reactor meltdown, a supertanker run aground, or a war? GIGO is no explanation for anything except our failure to install good data-validation checks, or worse, our failure to test the system's tolerance for bad data.
- Garbage shouldn't get in—not in the first place or in the last place.
- Every system must contend with a bewildering array of internal and external garbage, and if you don't think the world is hostile, how do you plan to cope with alpha particles?

1.2. Casual and Malicious Users

- Systems that interface with the public must be especially robust and consequently must have prolific input-validation checks.
- It's not that the users of automatic teller machines, say, are willfully hostile, but that there are so many of them—so many of them and so few of us.
- It's the million-monkey phenomenon: a million monkeys sit at a million typewriters for a million years and eventually one of them will type *Hamlet*.
- The more users, the less they know, the likelier that eventually, on pure chance, someone will hit every spot at which the system's vulnerable to bad inputs.
- There are malicious users in every population—infuriating people who delight in doing strange things to our systems.
- Years ago they'd pound the sides of vending machines for free sodas. Their sons and daughters invented the "blue box" for getting free telephone calls.
- Now they're tired of probing the nuances of their video games and they're out to attack computers. They're out to get *you*.
- Some of them are programmers. They're persistent and systematic. A few hours of attack by one of *them* is worse than years of ordinary use and bugs found by chance. And there are so many of them; so many of them and so few of us.
- Then there's crime. It's estimated that computer criminals (using mostly hokey inputs) are raking in hundreds of millions of dollars annually.
- A criminal can do it with a laptop computer from a telephone booth in Arkansas.
- Every piece of bad data accepted by a system—every crash-causing input sequence—is a chink in the system's armor that smart criminals can use to penetrate, corrupt, and eventually suborn the system for their own purposes.
- And don't think the system's too complicated for them.
- They have your listings, and your documentation, and the data dictionary, and whatever else they need.*
- There aren't many of them, but they're smart, motivated, and possibly organized.

1.3. Operators

- Roger and I were talking about operators and the nasty things they can do, and the scenarios were getting farfetched.
- Who'd think of mounting a tape with a write-ring installed, writing a few blocks, stopping, opening the transport's door, dismounting the tape reel without unloading the buffers, removing the ring, remounting the tape without telling the system, and then attempting to write a new block? The malice we ascribed to the operators was embarrassing.
- I said to Roger, the designer most concerned with the impact of operator shenanigans, "What the operators have done to these systems in the past is bad enough—just imagine how they'd act if they knew how we talked about them."
- To which he snapped, "If they knew how we talked about them, they'd probably act the way we expect them to!"
- I'm not against operators and I don't intend to put them down. They're our final defense against our latent bugs.
- Too often they manage, by intuition, common sense, and brilliance, to snatch a mere catastrophe from the jaws of annihilation.
- Operators make mistakes—and when they do, it can be serious. It's right that they probe the system's defenses, catalog its weaknesses and prepare themselves for the eventualities we didn't think of.

1.4. The Internal World

- Big systems have to contend not only with a hostile external environment but also a hostile internal environment.
- Malice doesn't play a role here, but oversight, miscommunication, and chance can be just as deadly.
- Any big system is subdivided into loosely coupled subsystems and consequently, there are many internal interfaces.
- Each interface presents another opportunity for data corruption and may require explicit internal-data validation.
- Furthermore, hardware can fail in bizarre ways that will cause it to pump streams of bad data into memory, across channels, and so on.
- Another piece of software may fail and do the same. And then there're always alpha particles.

1.5. What to Do

- Input validation is the first line of defense against a hostile world.
- Good designers design their system so that it just doesn't accept garbage—good testers subject systems to the most creative garbage possible.
- Input-tolerance testing is usually done as part of system testing, such as in a formal feature test or in a final acceptance test, so it's usually done by independent testers.
- This kind of testing and test design is more fun than any other kind I know of: it's great therapy and they pay you for it.
- My family and pets loved it when I was doing these tests; after I was through kicking and stomping the programmers around, there wasn't a mean bone left in my body.
- But to be really diabolical takes organization, structure, discipline, and method.
- Taking random potshots and waiting for inspirations with which to victimize the programmer won't do the job.
- Syntax testing is a primary toot of dirty testing, and method beats sadism every time.

1.6. Applications and Hidden Languages

- Opportunities for applying syntax testing abound in most systems because most systems have **hidden languages**.
- A **hidden language** is a programming language that hasn't been recognized as such.

- Remember the Third Law (Chapter 2, Section 3.4.1.): *Code Migrates to Data*.
- One of the ways this happens is by taking control information and storing it as data (in lists or tables, say) and then interpreting that data as statements in an unrecognized and undeclared, internal, high-level language.
- Syntax testing is used to validate and break the explicit or (usually) implicit parser of that language.
- The troubles with these hidden languages are: there's no formal definition of syntax; the syntax itself is often buggy; and parsing is inexorably intertwined with processing.
- The key to exploiting syntax testing is to learn how to recognize hidden languages. Here are some examples:

  1. User and operator commands are obvious examples of languages. Don't think that it doesn't pay to use syntax-testing methods because you only have a few commands. I've found it useful for only one or two dozen commands. For mainframe systems there are system operator commands and also a big set of user OS commands. For PC or any other application that can be used interactively, there are application-specific command sets.

  2. The counterpart to operator and user command languages for batch processing are job control languages: either at the operating system level (e.g., JCL) or application-specific.

  3. In Chapter 4, Section 5.2, I wrote about transaction-control languages. Reread that section for this application.

  4. A system-wide interprocess-communication convention has been established. Isn't that a minilanguage? A precompilation preprocessor has been implemented to verify that the convention is followed: use syntax-testing thinking to test it.

  5. An offline database generator package is used to create the database. It has a lot of fields to look at and many rules to follow—and more syntax to check.

  6. Any internal format used for interprocess communications that does not consist of simple, fixed fields should be treated to a syntax test—for example, project or application calling sequence conventions, or a macro library.

  7. Almost any communication protocol has, in part, a command language and/or formats that deserve syntax testing. Even something as simple as using a telephone can be tested by syntax testing methods.

  8. A complicated application may have several hidden languages: an external language for user commands and an internal language, not apparent to the user, out of which the applications are built. The internal languages could be subtle and difficult to recognize. For example, a language could consist of a pattern of calls to worker subroutines. A deep call tree with a big common subroutine library can be viewed as a **syntax graph** (see below). A tip-off for syntax testing is that there are a few high-level routines with subroutine or function names in the call and a big common subroutine library. When you see that kind of a call tree, think of syntax testing.
- I wouldn't use syntax-testing methods against a modern compiler.
- The way modern compiler construction is automated almost guarantees that syntax testing won't be effective.
- By extension, if the hidden language is out in the open and implemented as a real language, then syntax testing will probably fail—not because the tests won't be valid but because they won't reveal enough bugs to warrant the effort.
- Syntax testing is a shotgun method that depends on creating many test cases.
- Although any one case is unlikely to reveal a bug, the fact that many cases are used and that they are very easy to design makes the method effective.
- It's almost impossible for the kinds of bugs that syntax testing exposes to remain by the time (typically, in system testing) that syntax testing is used if it is used against a modern compiler.

- Also, for most programming languages there's no need to design these tests because you can buy a test oracle for the compiler at a far lower cost than you can design the tests.

1.7. The Graph We Cover

- Look at the back of almost any Pascal reference manual, and you'll see several pages of graphs such as the one shown in Figure 9. 1.
- It is a graph because it has nodes joined by links. The nodes shown here are either circles or boxes.
- The links are arrows as usual. The circles enclose actual characters.
- The boxes refer to other parts of the syntax graph, which you can think of as subroutine calls.
- The meanings attached to this graph are slightly different than those we've used before:
- **What do you do when you see a graph? COVER IT!** Do you know how to do syntax testing? Of course you do! Look at Figure 9. 1.
- You can test the normal cases of this syntax by using a covering set of paths through the graph.
- For each path, generate a *fields* that corresponds to that path.
- There are two explicit loops in the graph, one at the top that loops around *identifier* and another at the bottom following the "OF."
- There is an implicit loop that you must also test: note that *fields* calls itself recursively, I would start with a set of covering paths that didn't hit the recursion loop and only after that was satisfactory, hit the recursive cases.
- In syntax testing, we must test the syntax graph with (at least) a covering set of test cases, but we usually go much further and also test with a set of cases that cannot be on the graph—the dirty cases.
- We generate these by methodically screwing up one circle, box, or link at a time.
- Look at the comma in the top loop: we can remove it or put something else in its place for two dirty test cases.
- We can also test a link that doesn't exist, such as following the comma with a *type* as indicated by the dotted arrow.
- You also know the strategy for loops.
- The obvious cases are: not looping, looping once, looping twice, one less than the maximum, the maximum, and one more than the maximum.
- The not-looping case is often productive, especially if it is a syntactically valid but semantically obscure case.
- The cases near the maximum are especially productive.
- You know that there must be a maximum value to any loop and if you can't find what that value is, then you're likely to draw blood by attempting big values.

1.8. Overview

- Syntax testing consists of the following steps:

  1. Identify the target language or format (hidden or explicit).

  2. Define the syntax (format) of the language, formally, in a convenient notation such as **Backus-Naur form (BNF).**

  3. Test and debug the syntax to assure that it is complete and consistent and that it satisfies the intended semantics.

  4. Normal condition testing consists of a covering set of input strings including critical loop values. The difficult part about normal case testing is predicting the outcome and verifying that the processing was correct. That's ordinary functional testing—i.e., semantics. Covering the syntax graph assures that all options have been tested. This is a minimum mandatory requirement with the analogous strengths and weaknesses of branch testing for control flowgraphs. It isn't "complete" syntax testing by any measure.

5. Syntax testing methods pay off best for dirty testing. Test design is a top-down process that consists of methodically identifying which component is to be cruddied-up and how.

6. Much of syntax test design can and should be automated by relatively simple means.

7. Test execution automation is essential for syntax testing because this method produces so many tests.

2.A GRAMMAR FOR FORMATS:

2.1. Objectives

- Every input has a syntax.
- That syntax may be formally specified or undocumented and "just understood," but it does exist.
- Data validation consists (in part) of checking the input for correct syntax. It's best when the syntax is defined in a formal language—best for the designer and the tester.
- Whether the designer creates the data-validation software from a formal specification or not is not important to the tester, but the tester needs a formal specification to create useful garbage.
- That specification is conveniently expressed in Backus-Naur form, which is very similar to regular expressions.
- Regular expressions were introduced in Chapter 8 as an algebraic representation of all the paths in a graph.
- It's usually more convenient to deal with the algebraic version of graphs than with the pictorial version.
- Get comfortable with going back and forth between algebraic forms and pictorial forms for graphs and with talk about "covering a graph" even if there's no pictorial graph around.
- This isn't new to you because you worked with paths through an algebraic representation of a graph long before you heard about flowgraphs: what did you mean by "paths through code"?

2.2. BNF Notation (BACK59)

2.2.1. The Elements

- Every input can be considered as if it were a string of characters.
- The software accepts valid strings and rejects invalid ones.
- If the software fails on a string, we've really got it. If it accepts an invalid string, then it's guilty of GIGO.
- There's nothing we can do about syntactically valid strings whose values are valid but wrong—that kind of garbage we have to accept.
- The syntax definition must be formal, starting with the most elementary parts, the characters themselves.
- Here's a sample definition:

  alpha_characters ::= A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/
  R/S/T/U/V/W/X/Y/Z
  numerals ::= 1/2/3/4/5/6/7/8/9
  zero ::= 0
  signs ::= !/#/$/%/&/*/(/)/-/+/=/;/:/"/'/,/./?
  space ::= *sp*

- The left-hand side of the definitions is the name given to the collection of objects on the right-hand side.
- The string "::=" is interpreted as a single symbol that means "is defined as." The slash "/" means "or."
- We could have used the plus sign for that purpose as with regular expressions but that wouldn't be in keeping with the established conventions for BNF. We are using BNF to define a miniature language.
- The "::=" is part of the language in which we talk about the minitanguage, called the **metalanguage**.

- Spaces are always confusing because we can't display them on paper. We use *sp* to mean a space.
- The actual spaces on this page have no meaning.
- Similarly, an italicized (or underlined) symbol is used for any other single character that can't conveniently be printed, such as *null (nl), end-of-text (eot), clear-screen, carriage-return (cr), line-feed (lf), tab, shift-up (su), shift-down (sd), index, backspace (bs),* and so on.
- The underlined space, as in *alpha_characters*, is used as usual in programming languages to connect words that comprise a single object.

2.2.2. BNF Operators

- The operators are the same as those used in path expressions and regular expressions: "or," concatenate, (which doesn't need a special symbol), "×", and "+".
- Exponents, such as A^n, have the same meaning as before—*n* repetitions of the strings denoted by the letter A.
- Syntax is defined in BNF as a set of definitions.
- Each definition may in turn refer to other definitions or to itself in such a way that eventually it gets down to the characters that form the input string.
- Here's an example:
  word ::= *alpha_character alpha_character / numeral sp numeral*
- I've defined an input string called *word* as a pair of *alpha_characters* or a pair of *numerals* separated by a space.
- Here are examples of *words* and non*words*, by this definition:
  words : AB, DE, XY, 3 *sp* 4, 6 *sp* 7, 9 *sp* 9, 1 *sp* 2
  nonwords : AAA, A *sp* A1, A), 11, 111, WORD, NOT *sp* WORD, +
- There are 722 possible *words* in this format and an infinite number of non*words*. If the strings are restricted to four characters, there are more than a million non*words*.
- The designer wants to detect and accept words and reject non*words*; the tester wants to generate non*words* and force the program to deal with them.

2.2.3. Repetitions

- As before, $object^{1\text{-}3}$ means one to three *objects*, *object** means zero or more repetitions of *object* without limit, and *object*+ means one or more repetitions of object.
- Neither the star (*) nor the plus (+) can legitimately appear in any syntax because both symbols mean a possibly infinite number of repetitions.
- That can't be done in finite memory or in the real world. The software must have some means to limit repetitions.
- It can be done by an explicit test associated with every + or * operator, in which case you should replace the operator with a number.
- Another way to limit the repetitions is by placing a global limit on the length of any string.
- The limit then applies to all commands and it may be difficult to predict what the actual limit is for any specific command.
- You test this kind of limit by maximum-length strings. Yet another way to implement limits is to limit a common resource such as stack or array size.
- Again, the limits for a specific command may be unpredictable because it is a global limit rather than a format-specific limit.
- The way to test this situation is with many repetitions of short strings.
- For example, using as many as possible minimum-length *identifiers* and not including *expression* in the first loop of Figure 9.1: "ID1, ID2, ID3, ID4,.... ID999: *type*" will do the job.
- One of the signs of weak software is the ease with which you can destroy it by overloading its repetition-limiting mechanisms.

- If the mechanism doesn't exist, you can probably scribble all over the stack or code—crash-crash, tinkle-tinkle, goody-goody.

2.2.4. Examples

- This is an example and not a real definition of a telephone number:

special_digit ::= 1/2/5
zero ::= 0
other_digit ::= 3/4/6/7/8/9
ordinary_digit ::= *special digit / zero / other_digit*
exchange_part ::= *other_digit2 ordinary_digit*
number_part ::= *ordinary_digit4*
phone-number ::= *exchange_part number-part*

According to this definition, the following are *phone-numbers*,
3469900, 9904567, 3300000
and these are not:
5551212, 5510000, 123, 8, ABCDEFG, 572-5580, 886-0144.

- Another example:

operator_command ::= *mnemonic field_unit*$^{1-8}$ +
An *operator_command* consists of a *mnemonic* followed by one to eight *field_units* and a plus sign.
field_unit ::= *field delimiter*
mnemonic ::= *first_part second_part*
delimiter ::= *sp* / , / . / $ / ×sp$^{1-42}$
field ::= *numeral / alpha / mixed / control*
first-part ::= *a_vowel a_consonant*
second_part ::= *b_consonant alpha*
a_vowel ::= A/E/I/O/U
a_consonant ::= B/D/F/G/H/J/K/L/M/N/P/Q/R/S/T/V/X/Z
b_consonant ::= *B/G/X/Y/Z/W/M/R/C*
alpha ::= *a-vowel / a consonant b_consonant*
numeral ::= 1/2/3/4/5/6/7/8/9/0
control ::= $/×/%/*sp*/@
mixed ::= *control alpha control /*
control numeral control /
control control control

Here are some valid *operator_commands*:
ABXW A. B. C. 7. +
UTMA W *sp sp sp sp* +
While the following are not *operator-commands*:
ABC *sp* +
A *sp* BCDEFGHIJKLMNOPQR *sp*^{47} +

- The telephone number example and the operator command example are different.
- The telephone number started with recognizable symbols and constructed the more complicated components from them—a bottom-up definition.
- The command example started at the top and worked down to the real characters—a top-down definition.
- These two ways of defining things are equivalent—it's only a matter of the order in which the definition lines are printed.
- The top-down order is generally more useful and it's the usual form for language design.

- Looking at the definition from the top down leads you to some tests and looking from the bottom up can lead to different tests.
- As a final notational convenience, it's sometimes useful to enclose an expression in parentheses to reduce the number of steps in the definition.
- For example, the definition step for *field_unit* could have been simplified as follows:

  operator_command ::= *mnemonic (field delimiter)*[1-8] +
- This is fine if the syntax doesn't use parentheses that can confuse you in the definitions; otherwise use some other bracket symbols such as < and >.
- BNF notation can also be expanded to define optional fields, conditional fields, and the like.
- In most realistic formats of any complexity, you won't be able to get everything expressed in this notation—nor is it essential that you do so; additional narrative descriptions may be needed.

3. TEST CASE GENERATION

3.1. Generators, Recognizers, and Approach

- A data-validation routine is designed to recognize strings that have been explicitly or implicitly defined in accordance with an input syntax.
- It either accepts the string, because it is recognized as valid, or rejects it and takes appropriate action. The routine is said to be a **string recognizer**.
- Conversely, the tester attempts to generate strings and is said to be a **string generator**.
- There are three possible kinds of incorrect actions:
  1. The recognizer does not recognize a good string.
  2. It accepts a bad string.
  3. It may accept or reject a good string or a bad string, but in so doing, it fails.
- Even small specifications lead to many good strings and far more bad strings.
- There is neither time nor need to test them all. String errors can be categorized as follows:
  1. *High-Level Syntax Errors*—The strings have violations at the topmost level in a top-down BNF syntax specification.
  2. *Intermediate-Level Syntax Errors*—Syntax errors at any level other than the top or bottom.
  3. *Field-Syntax Errors*—Syntax errors associated with an individual field, where a **field** is defined as a string of characters that has no subsidiary syntax specification other than the identification of characters that compose it. A field is the lowest level at which it is productive to think in terms of syntax testing.
  4. *Delimiter Errors*—Violation of the rules governing the placement and the type of characters that must appear as separators between fields.
  5. *Syntax—Value Errors*—When the syntax of one field depends on values of other fields, there is a possibility of an interaction between a field-value error and a syntax error—for example, when the con-tents of a control field dictate the syntax of subsequent fields. This is a messy business that permits no reasonable approach. It needs syntax testing combined with domain testing, but it's better to redesign the syntax.
  6. *State-Dependency Errors*—The permissible syntax and/or field values is conditional on the state of the system or the routine. A command used for start-up, say, may not be allowed when the system is running. If state behavior is extensive, consider state testing (Chapter 11).
- Errors in the values of the fields or the relation between field values are domain errors and should be tested accordingly.

3.2. Test Case Design

3.2.1. Strategy

- The strategy is to create one error at a time, while keeping all other components of the input string correct; that is, in the absence of the single error, the string would have been accepted.

- Once a complete set of tests has been specified for single errors, do the same for double errors and then triple errors.
- However, if there are of the order of *N* single-error cases, there will be of the order of N^2 double-error and N^3 triple-error cases.
- Once past the single errors, it takes a lot of judicious pruning to keep the number of tests reasonable. This is almost impossible to do without looking at the implementation details.

3.2.2. Top, Intermediate, and Field-Level Syntax Errors

- Say that the topmost syntax level is defined as:

  item ::= *atype / btype / ctype / dtype etype*

  - Here are some obvious test cases:

    1. *Do It Wrong*—Use an element that is correct at some other lower syntax level, but not at this level.

    2. *Use a Wrong Combination.* The last element is a combination of two other elements in a specified order. Mess up the order and combine the wrong things:

    dtype atype / btype etype / etype dtype / etype etype / dtype dtype

    3. *Don't Do Enough*—For example,

    dtype / etype

    4. *Don't Do Nothing*. No input, just the end-of-command signal or carriage return. Amazing how many bugs you catch this way.

    5. *Do Too Much*—For example:

    atype btype ctype dtype etype / atype atype atype /
    dtype etype atype / dtype etype etype / dtype etype[128]

- Focus on one level at a time and keep the level above and below as correct as you can. It may help to draw a definition graph; we'll use the telephone number example (see Figure 9.2).
- Check the levels above and below as you generate cases. Not everything generated by this procedure produces bad cases, and the procedure may lead to the same test case by two different paths.
- The corruption of one element could lead to a correct but different string.
- Such tests are useful because logic errors in the string recognizer might miscategorize the string.
- Similarly, if a test case (either good or bad) can be generated via two different paths, it is an especially good case because there is a potential for confusion in the routine.
- I like test cases that are difficult to design and difficult to recognize as either good or bad because if I'm confused, it's likely that the designer will also be confused.
- It's not that designers are dumb and testers smart, but designers have much more to do than testers.
- To design and execute syntax-validation tests takes 5% to 10% of the effort needed to design, code, test, validate, and integrate a syntax-validation routine.
- The designer has 10 to 20 times as much work to do as does the tester.
- Given equal competence, if the tester gets confused with comparatively little to do, it's likely that the overloaded designer will be more confused by the same case.
- Now look at Figure 9.3. I generated this syntax graph from Figure 9.2 by inserting the subsidiary definitions and simplifying by using regular expression rules.
- The result is much simpler. It can obviously be covered by one test for the normal path and there are far fewer dirty tests:
  1. Start with a *special_digit.*
  2. Start with a *zero.*
  3. Only one *other_digit* before a *zero.*
  4. Only one *other_digit* before a *special-digit.*
  5. Not enough digits.

6. Too many digits.
7. Selected nondigits.

- You could get lavish and try starting with two *zeros*, two *special_digits, zero* followed by *special_digit*, and *special_digit* followed by *zero*, thereby hitting all the double-error cases of interest; but there's not much more you can do to mess up this simplified graph. Should you do it?
- No! The implementation will tend to follow the definition and so will the bugs.
- Therefore, there's a richer possibility for variations of bad strings to which the simplified version is not vulnerable.
- Don't expect to find opportunities for such simplifications in the syntax of mature programming languages—the language designers usually do as much simplification as makes sense before the syntax is released.
- For formats, operator commands, and hidden languages, there are many such opportunities.
- The lesson to be learned from this is that you should always simplify the syntax graph if you can.
- The implementation will be simpler, it will take fewer tests to cover the normal cases, and there will be fewer meaningful dirty tests.

3.2.3. Delimiter Errors

- **Delimiters** are characters or strings placed between two fields to denote where one ends and the other begins.
- Delimiter problems are an excellent source of test cases. Therefore, it pays to identify the delimiters and the rules governing their syntax.

1. *Missing Delimiter*—This kind of error causes adjacent fields to merge. This may result in a different, but valid, field or may be covered by another kind of syntax error.

2. *Wrong Delimiter*—It's nice when several different delimiters are used and there are rules that specify which can be used where. Mix them up and use them in the wrong places.

3. *Not a Delimiter*—There are some characters or strings that are not delimiters but could be put into that position. Note the possibility of changing adjacent field types as a result.

4. *Too Many Delimiters*—The number of delimiters appearing at a field boundary may be variable. This is typical for spaces, which can serve as delimiters. If the number of delimiters is specified as 1 to *N*, it pays to try 0, 1, 2, *N* - 1, *N*, *N* + 1, and also an absurd number of delimiters, such as 127, 128, 255, 256, 1024, and so on.

5. *Paired Delimiters*—These delimiters represent another juicy source of test cases. Parentheses are the archetypal paired delimiters. There could be several kinds of paired delimiters within a syntax. If paired delimiters can nest, as in "(()())", there are a whole set of new evils to perpetrate. For example, "BEGIN...BEGIN...END", "BEGIN...END...END". Nested paired delimiters provide opportunities for matching ambiguities. For example, "((()(()))" has a matching ambiguity and it's not clear where the missing parenthesis belongs.

6. *Tolerant Delimiters*—The delimiter may be optional or several alternate formats may be acceptable. In communications systems, for example, the start of message is defined as ZCZC, but many systems allow any one of the four characters to be corrupted. Therefore, #CZC, Z#ZC, ZC#C, and ZCZ# (where "#" denotes any character) are all acceptable; there are many nice confusing ways to put in a bad character here:

a. A blank.
b. Z or C in the wrong place—CCZC, ZZZC, ZCCC, ZCZZ (catches them every time!).
c. Something off-the-wall—especially important control characters in some other context.

- Tolerance is most often provided for delimiters but can also be provided for individual fields and higher levels of syntax.
- It's a sword of many edges—more than two for sure—all of them dangerous.

- Syntax tolerance is provided for user and operator convenience and in the interest of making the system humane and robust.
- But it also makes the format-validation design job and testing format-validation designs more complicated.
- Format tolerance is sophisticated and takes sophisticated designs to implement and, consequently, many more and more complicated tests to validate.
- If you can't do the whole job from design to thorough validation, there's no point in providing the tolerance.
- Most users and operators prefer a solid system with rigid syntax rules to a system with tolerant rules that don't always work.

3.2.4. Field-Value Errors

- Field-value errors are clearly a domain-testing issue, and domain testing is where it's at.
- Whether you choose to implement field-value errors in the context of syntax testing or the other way around (i.e., syntax testing under domain testing) or whether you choose to implement the two methods as separate test suites depends on which aspect dominates.
- Syntax-testing methods will usually wear out more quickly than will domain testing.
- For that reason, it pays to separate domain and syntax tests into different suites.
- You may not be able to separate the two test types because of (unfortunately) context-dependent syntax—either field values whose domain depends on syntax or syntax that depends on field values (ugh!).
- Here's a reminder of what to look for: boundary values and near-boundary values, excluded values, binary values for integers, and values vulnerable to semantic type changes and representation changes.

3.2.5. Context-Dependent Syntax Errors

- Components of the syntax may be interrelated and may be related by field values of other fields. The first field could be a code that specifies the syntax of subsequent fields. As an example:
  command ::= *pilot_field syntax_option*
  pilot_field ::= 1/2/3/4/5/6/7/8/9
  syntax_option ::= option1 / option2 / option3 / . . .
- The specification further states that option!] must be preceded by "1" as the value of the *pilot-field*. Actually, it would have been better had the specification be written as:
  command ::= 1 *option1* / 2 *option2* / 3 *option3* / . . .
- but that's not always easy to do. The test cases to use are clearly invalid combinations of syntactically valid field values and syntactically valid options.
- If you can rewrite the specification, as in the above example, to avoid such field-value dependencies, then it's better you do so; but if so doing means vast increases in formality, which could be handled more clearly with a side-bar notation that specifies the relation, then it's better to stick to the original form of the specification.
- The objective is to make things as clear as possible to the designer and to yourself, and excessive formality can destroy clarity just as easily as modest formality can enhance it.

3.2.6. State-Dependency Errors

- The string or field value that may be acceptable at one instant may not be acceptable at the next because validity depends on the transaction's or the system's state.
- As an example, say that the operator's command-input protocol requires confirmation of all commands.
- After every command the system expects either an acknowledgment or a cancellation, but not another command.
- A valid command at that point should be rejected, even though it is completely correct in all other respects.

- As another example, the system may permit, as part of a start-up procedure, commands that are forbidden after the system has been started, or it may forbid otherwise normal commands during a start-up procedure.
- A classical example occurs in communications systems.
- The start of message sequence ZCZC is allowed tolerance (see page 301) when it occurs at the beginning of a message.
- However, because the system has to handle Polish language words such as: “zczalny”, “jeszcze”, “deszcz”, and “zczotka” (BEIZ79), the rules state that any subsequent start-of-message sequence that occurs prior to a correct end-of-message sequence (NNNN) must be intolerant; the format is changed at that point and only an exact “ZCZC” will be accepted.
- I divide state-dependency errors into “simple” and “complicated.”
- The simple ones are those that can be described by at most two states.
- All the rest are complicated and are best handled by the methods of Chapter 11.
- The simple ones take two format or two field-value specifications—and require at worst double the work.

3.3. Sources of Syntax

3.3.1. General

- Where do you get the syntax? Here’s another paradox for you.
- If the syntax is served up in a nice, neat, package, then syntax-testing methods probably won’t be effective and if syntax testing is effective, you’ll have to dig out and formally define the syntax yourself. Where do you get the syntax?
- Ideally, it comes to you previously defined, formally defined, in BNF or an equivalent, equally convenient notation.*
- That’s the case for common programming languages, command languages written by and for programmers, and languages and formats defined by a formal standard.

3.3.2. Designer-Tester Cooperation and Design Specifications

- If there is no BNF specification, I try to get the designers to create one—at least the first version of one.
- Realistically, though, if a BNF specification does not exist, the designers will have to create a document that can be easily converted into one or what is she designing to?
- If you get the designer to create the first version of the BNF specification, you may find that it is neither consistent nor complete.
- Test design begins with requests for clarification of that preliminary specification. Many serious bugs can be avoided this way.
- Do it yourself if you can’t get the designers to create the first version of the BNF specification.
- It doesn’t really matter whether it’s complete or correct, as long as it’s down on paper and formal. Present your specification version to the designers and say that tests will be defined accordingly.
- There may be objections, but the result should be a reasonably correct version in short order.
- Using a BNF specification is the easiest way to design format-validation test cases. It’s also the easiest way for designers to organize their work, but sadly they don’t always realize that.
- You can’t begin to design tests unless you agree on what is right or wrong.
- If you try to design tests without a formal specification, you’ll find that you’re throwing cases out, both good and bad, as the designers change the rules in response to the cases you show them.
- If you can’t get agreement on syntax early in the project, put off syntax test design and concentrate on some other area.
- Alternatively, and more productively, participate in the design under the guise of getting a specification tied down.
- You’ll prevent lots of bugs that way.

- It can boomerang, though. I pushed for a BNF specification of operator commands on one project.
- The commands had been adapted from a previous system in the same family whose formats were clearly specified but not in BNF.
- This designer fell in love with BNF and created a monster that was more complicated than a combination of Ada and COBOL—and mostly wrong.
- To make matters worse, his first version of the operator's manual was written in top-down BNF, so operators had to plow through several levels of abstract syntax to determine which keys to hit.
- Good human engineering will dictate simple, clean, easy-to-understand syntax for user and operator interfaces.
- Similarly, internal formats for interprocess communications should also be simple.
- There's usually a topmost syntax level, several field options, and a few subfields.
- Recursive definitions are rare (or should be).
- We do find useful recursive definitions in operating system command languages or data query languages for things such as sorting, data object specifications, and searching; but in general, recursive definitions are rare and more likely to be a syntax-specification error than a real requirement.
- Be suspicious if the syntax is so complicated that it looks like a new programming language. That's not a reasonable thing to expect users or operator to employ.

3.3.3. Manuals as Sources

- Manuals, such as instruction manuals, reference manuals, and operator manuals are the obvious place to start for command languages if there isn't a formal syntax document and you can't get designers to do the job for you.
- The syntax in manuals may be fairly close to a formal syntax definition.
- Manuals are good sources because more often than not, we're dealing with a maintenance situation, rehosting, or a rewrite of an old application.
- But manuals can be mushy because the manual writer tries to convey complicated syntactic issues in a language that is "easy for those dumb users and operators."*
- *Another war story about my favorite bad software vendor, Coddler Incorporated—a pseudonym invented to protect me from lawsuits.
- They had a word processing programming language that was putatively designed for use by word processing operators.
- The language's syntax description in the manual was sketchy and wrong.
- It's the only programming language that ever defeated me because after weeks of trying I still couldn't write more than 20 statements without a syntax error.
- When I asked them for complete documentation of the language's syntax, I was told that, as a mere user, I wasn't entitled to such "proprietary" information. We dragged the syntax out of that evil box experimentally and discovered so many context and state dependencies that it was clear to us that they had broken new grounds in language misdesign.
- What was even worse, when they confronted you with a syntax error, it was presented by reference to the tokenized version of the source after compilation rather than to the source—you guessed it, the token table was also "proprietary."
- We dragged that out experimentally also and discovered even more state dependencies.
- Syntax testing that garbage dump was like using hydrogen bombs against a house of cards.
- The ironic thing about this experience was that we were trying to use this "language" and its "processor" to automate the design of syntax tests for the system we were testing.

3.3.4. Help Screens and Systems

- Putting user information such as command syntax into HELP systems and on-line tutorial is becoming more commonplace, especially for PC software because it's cheaper to install a few

hundred K of HELP material on a floppy than it is to print a few hundred pages of instruction manual.

- You may find the undocumented syntax on these screens.
- If you have both manuals and help systems, compare them and find out which one is correct.

Data Dictionary and Other Design Documents

- For internal hidden languages, your most likely source of syntax is the data dictionary and other design documents.
- Also look at internal interface standards, style manuals, and design practice memos.
- Common subroutine and macro library documents are also good sources.
- Obviously you can't expect designers to hand you the syntax of a language whose existence they don't even recognize.

3.3.6. Prototypes

- If there's a prototype, then it's likely to embody much of the user interface and command language syntax you need. This source will become more useful in the future as prototyping gains popularity. But remember that a prototype doesn't really have to work, so what you get could be incomplete or wrong.

3.3.7. Programmer Interviews

- The second most expensive way to get user and operator command syntax is to drag the information out of the implementing programmer's head by interviews.
- I would do it only after I had exhausted all other sources.
- If you're forced to do this as your only source, then syntax testing may be pointless because a low-level programmer is making user interface or system architecture decisions and the project's probably aground on the reef—it just hasn't sunk yet.
- Syntax testing is then just a cruel way to demonstrate that fact—pitiful.

3.3.8. Experimental

- The most expensive possible way to get the syntax is by experimenting with the running program.
- Think back to the times you've had to use a new system without an instruction manual and of how difficult it was to work out even a few simple commands—now think of how much work that can be for an entire set of commands; but for dirty old code, it's sometimes the only way.
- You got it.
- Take what you know of the syntax and express it in BNF.
- Apply syntax testing to that trial syntax and see what gets accepted, what gets rejected, and what causes crashes and data loss.
- You'll have a few surprises that will cause you to change your syntax graph.
- Change it and start over again until either the money runs out or the program's been scrapped. Looking at the code may help, but it often doesn't because, as often as not, parsing, format validation, domain validation, and processing are hopelessly intertwined and splattered across many components and many levels.
- It's usually pretty tender software.
- If you have to live with it, think in terms of putting a proper format validation front end on such junk (see Section 6 below) and avoid the testing altogether.

3.4. Ambiguities and Contradictions

- Unless it's the syntax of a programming language or a communication format or it's derived from a previous system or from some other source that's been in use for a long time, it's unlikely that the syntax of the formats you're testing will be correct the first time you test it.
- There will be valid cases that are rejected and other cases, valid or otherwise, for which the action is unpredictable.
- I mean fundamental errors in the syntax itself and not in the routines that analyze the format.

- If you have to create the format syntax in order to design tests, you are in danger of creating the format you want rather than the one that's being implemented.
- That's not necessarily bad if what you want is a simpler, more reliable, and easier format to use, implement, and test.
- Ambiguities are easy to spot—there's a dangling branch on the definition tree.
- That is, something appears on the right-hand side of a definition, but there's no definition with that term on the left-hand side.
- An obvious contradiction occurs when there are two or more definitions for the same term.
- As soon as you permit recursive definitions, state dependencies, and context-dependent syntax, the game's over for easy ambiguity and contradiction spotting—in fact, the problem's known to be unsolvable.
- Approaches to detecting ambiguities and contradictions in the general case is a language-validation problem and beyond the scope of this book by a wide margin.
- I'm assuming that we're dealing only with the simpler and more obvious ambiguities and contradictions that become apparent when the format is set down formally (e.g., written in BNF) for the first time.
- The point about syntactic ambiguities and contradictions (as I've said several times before) is that although a specification can have them, a program, because it is deterministic, is always unambiguous and consistent.
- Therefore, without looking at the code, even before the code's been designed, you know that there *must* be bugs in the implementation.
- Take advantage of every ambiguity and contradiction you detect in the format to push the format's design into something that has fewer exception conditions, fewer state dependencies, fewer field correlations, and fewer variations. Keep in close contact with the format's designer, who is often also the designer of the format-analysis routine.
- Maintain a constant pressure of weird cases, interactions, and combinations.
- Whenever you see an opportunity to simplify the format, communicate that observation to the format's designer: he'll have less to design and code, you'll have less to test, and the user will thank you both for a better system.
- It's true that flaws in the syntax may require a more elaborate syntax and a more complicated implementation.
- However, my experience has been that the number of instances in which the syntax can be simplified outnumber by about 10 to 1 the instances in which it's necessary to complicate it.

3.5. Where Did the Good Guys Go?

- Syntax test design is like a lot of other things that are hard to stop once you've started.
- A little practice with this technique and you find that the most innocuous format leads to hundreds of tests; but there are dangers to this kind of test design.

1. *It's Easy to Forget the Normal Cases*—I've done it often. You get so entangled in creative garbage that you forget that the system must also be subjected to good inputs. I've made it a practice to check every test area explicitly for the normal case. Covering the syntax definition graph does it.

2. *Don't Go Overboard with Combinations*—It takes iron nerves to do this. You've done all the single-error cases, and in your mind you know exactly how to create the double—and higher-error cases. And there are so many of them that you can create an impressive mound of test cases in short order. "How can the test miss anything if I've tried 1000 input format errors?," you think. Remind yourself that any one strategy is inherently limited to discovering certain types of bugs. Remind yourself that those N^2 double-error cases and N^3 triple-error cases may be no more effective than trying every value from 1 to 1023 in testing a loop.

Don't let the test become top-heavy with syntax tests at the expense of everything else just because syntax tests are so easy to design.

3. *Don't Ignore Structure*—Just because you can design thousands of test cases without looking at the code that handles those cases doesn't mean you should do it that way. Knowing the program's design may help you eliminate cases wholesale without sacrificing the integrity and thoroughness of the test. As an example, say that operator-command keywords are validated by a general-purpose preprocessor routine. The rest of the input character string is passed to the appropriate handler for that operator command only after the keyword has been validated. There would be no point to designing test cases that deal with the interaction of keyword errors, the delimiter between the keyword and the first field, and format errors in the first field. You don't have to know a whole lot about the implementation. Often, just knowing what parts of the format are handled by which routines is enough to avoid designing a lot of impressive but useless error combinations that won't prove a thing. The bug that could creep across that kind of interface would be so exotic that you would have to design it. If it takes several hours of work to postulate and "design" a bug that a test case is supposed to catch, you can safely consider that test case as too improbable to worry about—certainly in the context of syntax testing.

4. *There's More than One Kind of Test*—Did you forget that you designed path tests and domain tests—that there are state tests to design (Chapter 11), data-flow tests (Chapter 5), or logic-based tests (Chapter 10)? Each model of the system's behavior leads to tests designed from a different point of view, but many of these tests overlap. Although redundant tests are harmless, they cost money and little is learned from them.

5. *Don't Make More of the Syntax Than There Is*—You can increase or decrease the scope of the syntax by falsely making it more or less tolerant than it really is. This may lead to the false classification of some good input strings as bad and vice versa—not a terrible problem, because if there is confusion in your mind, there may be confusion in the designer's mind. At worst, you'll have to reclassify the outcome of some cases from "accept" to "reject," or vice versa.

6. *Don't Forget the Pesticide Paradox*—Syntax tests wear out fast. Programmers who don't change their design style after being mauled by syntax testing can probably be thrashed by any testing technique, now and forever. However they do it, by elegant methods or by brute force, good programmers will eventually become immune to syntax testing.

4. IMPLEMENTATION AND APPLICATION

4.1. Execution Automation

4.1.1. General

- Syntax testing, more than any other technique I know, forces us into test execution automation because it's so easy to design so many tests (even by hand) and because design automation is also easy.
- Syntax testing is a shotgun method which—like all shotgun methods—is effective only if there are a lot of pellets in your cartridge.
- How many ducks will you bring down if you have to throw the pellets up one at a time?
- An automation platform is a prerequisite to execution automation. The typical dedicated (dumb) terminal is next to useless.
- Today, the box of choice is a PC with a hard disc and general-purpose terminal emulator software such as CROSSTALK MK-4 (CROS89).
- If you've still got 37xx or VTxxx terminals, or teleprinters, or even cardwallopers around and someone wants to foist them off on you as test platforms, resist—violently.
- Dumb terminals are hardly better than paper tape and teleprinters.

4.1.2. Manual Execution

- Manual execution? Don't! Even primitive automation methods such as putting test cases on paper tape (see the first edition) was better than doing it manually.
- I found that the only way it could be done by hand was to use three persons, as in the following scenario.
- If that doesn't convince you to automate, then you're into compulsive masochism.
- Use three persons to do it.
- The one at the terminal should be the most fumble-fingered person in the test group.
- The one with the test sheet should be almost illiterate.
- The illiterate calls out one character at a time, using her fingers to point to it, and moving her lips as she reads.
- The fumble-fingered typist scans the keyboard (it helps if he's very myopic) and finally finds it.
  "A" the illiterate calls out.
  "A" the typist responds when he's got his finger on the key. He presses it and snatches it away in fear that it will bite him.
  "Plus" the reader shouts.
  "No, dammit!" the third person, the referee, interrupts (the only one in the group who acts as if she had brains).
  The idiot typist looks for the "DAMMIT" key.* . . .
- *Don't snicker.
- Ask your friends who work in PC software customer service how many times they've had inquiries from panicked novices who couldn't find the "ANYKEY" key—as in ". . . then hit any key."
- So I'm exaggerating: but it's very hard to get intelligent humans to do stupid things with consistency. Syntax testing is dominated by stupid input errors that you've carefully designed.

4.1.3. Capture/Replay

- See Chapter 13 for a more detailed discussion of capture/replay systems.
- **A capture/replay** system captures your keystrokes and stuff sent to the screen and stores them for later execution.
- However you've designed your syntax tests, execute them the first time through a capture/replay system if that's the only kind of execution automation you can manage.
- These systems (at least the acceptable ones) have a built-in editor or can pass the test data to a word processor for editing.
- That way, even if your first execution is faulty, you'll be able to correct it.

4.1.4. Drivers

- Build or buy a **driver—**a program that automatically sequences through a set of test cases usually stored as data.
- Don't build the bad strings (especially) as code in an ordinary programming language because you'll be going down a diverging infinite sequence of test testing.

4.1.5. Scripting Languages

- A **scripting language** is a language used to write test scripts. CASL (CROS89, FURG89) is nice scripting language because it can be used to emulate any interface, work from strings stored as data, provide smart comparisons for test outcome validation, editing, and capture/replay.

4.2. Design Automation

4.2.1. General

- Syntax testing is a good place to begin a test design automation effort because it's so easy and has such a high, immediate payoff.

- It's about the only test design automation area in which you can count on a payback the first time out.

4.2.2. Primitive Methods

- You can do design automation with a word processor.
- If you don't have that, will you settle for a copying machine and a bottle of white-out? Design a covering set of correct input strings.
- If you want to, because you have to produce paper documentation for every test case, bracket your test strings with control sequences such as "$$$XXX" so that you'll be able to extract them later on.
- Let's say you're doing operator commands.
- Pick any command and reproduce the test sheet as often as you need to cover all the bad cases for that command.
- Then, using the word processor's search-and-replace feature, replace the correct substring with the chosen bad substring.
- If you use the syntax definition graph as a guide, you'll see how to generate all the single-error cases by judicious uses of search-and-replace commands.
- Once you have the single-error cases done, go on to the double errors if you don't already have more cases than you can handle.
- With double errors you have to examine each case to be sure that it is still an error case rather than a correct case—similarly for triple and higher errors.
- If you're starting with a capture/replay system, then you can do the editing either in the system's own editor or with a word processor.
- It's really more difficult to describe than to do.
- Think about how you might automate syntax test design with just a copying machine and a hardy typist: then graduate to a word processor.
- If you understand these primitive methods, then you'll understand how to automate much of syntax test design.

4.2.3. Scripting Languages

- A scripting language and processor such as CASL has the features needed to automate the replacement of good substrings by bad ones on the fly.
- You can use random number generators to select which incorrect, single, character will be used in any spot.
- Similarly for replacing incorrect keywords by correct ones and for deciding whether or not to delete mandatory fields.
- You can play all kinds of game with this, but remember that you'll not be able to predict which produced strings are right or wrong.
- This is a good approach to use if your main purpose is to stress the software rather than to validate the format validation software.
- If you want to do it right, whatever language you do it in, you have to get more sophisticated.

4.2.4. Random String Generators

- Why not just use a random number generator to generate completely random strings?
- Two reasons: random strings get recognized as invalid too soon, and even a weak front end will catch most bad strings.
- The probability of hitting vulnerable points is too low, just as it was for random inputs in domain testing—there are too many bad strings in the world.
- A random string generator is very easy to build. You only have to be careful about where you put string terminators such as carriage returns.
- Throw the dice for the string length and then pack random characters (except string terminators) in *front* of the terminator until you've reached the required length.

- Easy but useless. Even with full automation and running at night, this technique caught almost nothing of value.

4.2.5. Getting Sophisticated

- Getting sophisticated means building an anti-parser.
- It's about as complicated as a simple compiler.
- The language it compiles is BNF, and instead of producing output code it produces structured garbage. I'll assume that you know the rudiments of how a compiler works—if not, this section is beyond you.
- As with a compiler, you begin with the lexical scan and build a symbol table. The symbols are single characters, keywords, and left-hand sides of definitions.
- Keep the three lists separate and replace the source symbols with numerical tokens.
- Note that each definition string points to one or more other definition strings, characters, or keywords—i.e., to other tokens in the symbol table.
- There are two ways to screw up—bad tokens and bad pointers.
- Start with a covering set of correct test cases.
- This can be done by hand or by trial and error with random number generators or by using flow-analyzer techniques such as are used in path test generators.
- Given a good string, you now scan the definition tree by using a tree search procedure.
- At every node in the subtree corresponding to your good test case you can decide whether you're going to use an incorrect token or an incorrect branch to a subsidiary definition.
- Use random numbers to replace individual characters, keywords, or pointers to other definitions.
- Double errors work the same way except that they use the single-error strings as a seed.
- Similarly, triple-error cases are built on using the double-error cases as a seed.
- They grow fast.
- Another way to look at automated syntax test generation is to view the normal cases as path test generation over BNF as the source language.
- The errored cases are equivalent to creating **mutations** (BUDD81, WHIT87) of the source "code.
- " There is no sensitization problem because all paths are achievable.
- If you've read this far, then you know that you can't guarantee bad strings, even for single-error cases because that's a known unsolvable problem.
- Double errors increase the probability of correct strings because of error cancellations.
- The only (imperfect) way to sort the good from the bad is to use the BNF specification as data to a parser generator and then use the generated parser to sort for you—it can't be perfect but it should do for simple operator commands.
- What's the point of generating test strings and using an automatically created parser to sort the good from the bad? If you've got such a parser, use *it* instead of the code you're testing?
- If we were dealing with entirely new code and a new command language, it would be better to generate the parser and avoid the testing.
- Using the generated parser as above is useful if it's an older system under maintenance and your objective is to build a big syntax-testing suite where one didn't exist before.

4.3. Productivity, Training, and Effectiveness

- I used syntax test design as basic training for persons new to a test group.
- With very little effort they can churn out hundreds of good tests.
- It's a great confidence builder for people who have never done formal test design before and who may be intimidated by the prospect of subjecting a senior designer's masterpiece to a barrage of calculated heartburn.
- With nothing more sophisticated for automation than a word processor and a copying machine, a testing trainee can usually produce twenty to thirty fully documented test cases *per hour* after a few days of training.

- Syntax testing is also an excellent way of convincing a novice tester that testing is infinite and that the tester's problem is not generating tests but knowing which ones to cull.
- When my trainees told me that they had run out of ideas, it was time to teach them syntax testing.
- I would always ask them to produce *all* single-, double-, and triple-error cases for a few well-chosen operator commands. Think about it.

4.4. Ad-Lib Tests

- Whenever you run a formal system test there's always someone in the crowd who wants to try ad-lib tests.
- And almost always, the kind of test they want to ad-lib is an input-syntax error test. I used to object to adlibbing, because it didn't prove anything—I thought.
- It doesn't prove anything substantive about the system, assuming you've done a good job of testing—which is why I used to object to it.
- It may save time to object to ad-lib tests, but it's not politic.
- Allowing the ad-lib tests demonstrates that you have confidence in the system and your test.
- Because a system wide functional demonstration should have been through a dry run in advance, the actual test execution is largely ceremonial (or should be) and the ad-libbers are part of the ceremony, just as hecklers are part of the ball game—it adds color to the scene.
- You should never object if the system's final recipient has cooked up a set of tests of his own.
- If they're carefully constructed, and well documented, and all the rest, you should welcome yet another independent assault on the system's integrity.
- Ad-lib tests aren't like that.
- The customer has a hotshot operator who's earned a reputation for crashing any system in under 2 minutes, and she's itching to get her mitts on yours.
- There's no prepared set of tests, so you know it's going to be ad-libbed. Agree to the ad-libbing, but only after all other tests have been done. Here's what happens:

  1. Most of the ad-lib tests will be input strings with format violations, and the system will reject them—as it should.

  2. Most of the rest are good strings that look bad. The system accepts the strings and does as it was told to do, but the ad-lib tester doesn't recognize it. It will take a lot of explanation to satisfy the customer that it was a cockpit error.

  3. A few seemingly good strings will be correctly rejected because of a correlation problem between two field values or a state dependency. These situations will also take a lot of explanation.

  4. At least once, the ad-lib tester will shout "Aha!" and claim that the system was wrong. It will take days to dig out the documentation that shows that the way the system behaves for that case is precisely the way the customer insisted that it behave—over the designer's objections.

  5. Another time the ad-lib tester will shout "Aha!" but, because the inputs weren't documented and because nonprinting characters were used, it won't be possible to reproduce the effect. The ad-lib tester will be forever convinced that the system has a flaw.

  6. There may be one problem, typically related to an interpretation of a specification ambiguity, whose resolution will probably be trivial.
- This may be harsh to the ad-lib testers of the world, but such testing proves little or nothing if the system is good, if it's been properly tested from unit on up, and if there has been good quality control.
- If ad-lib tests do prove something, then the system's so shaky and buggy that it deserves the worst that can be thrown at it.

Software Testing Methodologies Unit IV

5.TESTABILITY TIPS

5.1. The Tip

- Here's the whole testability tip:
  1. Bring the hidden languages out of the closet.
  2. Define the syntax, formally, in BNF.
  3. Simplify the syntax definition graph.
  4. Build a parser.
- I'll quell the objections to the last step by pointing out that building a minicompiler is a typical senior-year computer science project these days.
- I find that interesting because although most computer science majors build a compiler once in their lifetime, they'll never have a chance to build a real compiler once they're out in the world—we just don't need that many programming language compilers.
- So drag out the old notebooks to remember how it was done and if you learned your programming before compiler building was an undergraduate exercise, get one of the kids to do it for you.

5.2. Compiler Overview

- This overview is superficial and intended only to illustrate testability issues. Compilation consists of three main steps: **lexical analysis**, **parsing,** and **code production.**
- In our context, we deal most often not with a compiler as such, but with an **interpreter;** but if we're testing hidden languages, then indeed we may be interested in a compiler.
- The main difference between an interpreter and compiler is that an interpreter works one statement at a time and does the equivalent of code production on the fly.

  1. *Lexical Analysis*—The lexical analysis phase accomplishes the following:

  a. The analyzer knows enough about parsing to identify individual fields, where we define a **field** as a linguistic element that uses no lower-level definitions. That is, a field is defined solely in terms of primitive elements such as characters.

  b. Identifies interfield separators or delimiters.

  c. Classifies the field (e.g., integer, string, operator, keyword, variable names, program labels). Some fields, such as numbers or strings, may be translated at this point.

  d. New variable names and program labels are put into a **symbol table** and replaced by a pointer to that table. If a variable name or program label is already in the table, its appearance in the code is replaced by the pointer. The pointer is an example of a **token.**

  e. Keywords (e.g., STOP, IF, ELSE) are also replaced by tokens as are single-character operators. Numbers and strings are also put in a table and replaced by pointers.

  f. Delimiters are eliminated where possible, such as interfield delimiters. If the language permits multiple statements per line and there are statement delimiters, statements will be separated so that subsequent processing will be done one statement at a time. Similarly, multiline statements are combined and thereafter treated as a single string.

  g. The output of the lexical analysis phase is the partially translated version of the source in which all linguistic components have been replaced by tokens. The act of replacing these components by tokens is called **tokenizing**.

  2. *Parsing*—Parsing is done on tokenized strings. There are many different strategies used for parsing, and they depend on the kind of statement, the language, and the compiler's objectives. There is also a vast literature on the subject. For general information you can start with LEER84 or MAG184. From our point of view, the validation aspect of parsing consists of showing that the string to be parsed corresponds to a path in the syntax graph. The output of the parser is a tree with the statement identifier at the top, primitive elements (e.g., characters and keywords) at the bottom, and with intermediate nodes corresponding to definitions that were traversed along the path through the syntax graph.

3. *Code Production*—Code production consists of scanning the above tree (bottom-up, say) in such a way as to assure that all objects needed are available when they are needed and then replacing the tokens with sequences of instructions that accomplish what the tokens signify. From our point of view and our typical use of syntax testing, the equivalent to code production is a call to a worker subroutine or transfer of control to the appropriate program point.

5.3. Typical Software

- Unlike the above operation with its clear separation of lexical analysis, parsing, and production, the typical software for (operator command, say) syntax validation and command processing follows what I like to call:
  (*lex-a-little* + *parse_a_little* + *process_a_little)**
- Because the three aspects of command interpretation are hopelessly intermixed, a single bug can involve all three aspects. In other words, the ground for bugs is much more fertile.

5.4. Separation of Phases

- Separation of the three phases means that it is virtually impossible for a bug to involve, say, the interaction between processing, say, and parsing.
- We can handle the lexical testing by low-level syntax testing based on one field at a time and be confident that we can ignore lexical-level field interactions, except where field delimiters are involved.
- Similarly, because most delimiters are eliminated during lexical analysis, we don't have to bother with combinations of syntax errors and long delimiter strings.
- Lexical-parsing separation means that test strings with combined lexical and syntax errors will not be productive.
- Parsing-processing separation means that we can separate domain testing from syntax testing.
- Domain analysis is the first stage of processing and follows parsing, and it is therefore independent of syntax.
- The bottom line of phase separation is the wholesale elimination of possible double-error and higher-order vulnerabilities and therefore the need to even consider such cases.
- In addition to more robust software that's easier to test, there's a payoff in maintenance.
- The lexical definitions, the syntax, and the equivalent of code that points to working subroutines that do the actual processing can all be stored in tables rather than as code.
- Separation means separate maintenance. If a processing routine is wrong, there's no need to change the lexical analyzer or parser.
- If a new command is to be added, chances are that only the parser and keyword table will be affected.
- Similarly for enhancing existing commands.
- What's the price? Possibly more memory, possibly more processing time, but probably neither.
- The ad hoc *lex_a_little*, *parse_a_little* code is a jumbled mess that often contains a lot of Code redundancy and wasted reprocessing.
- My own experience has been that in every instance where we replaced an old-style format analyzer and processor with an explicit lex analyzer and parser, even though we had planned on more time and more space, to our surprise the new software was tighter and faster.

5.5. Prerequisites

- The language must be decent enough so that it is possible to do lexical analysis before parsing and parsing before processing.
- That means that it is possible to pull out the tokens in a single left-to-right pass over the string and to do it independently of any other string or statement in the language.
- The kind of thing that can't be handled for example, are formats such as:

- "If the first character of the string is an alpha, then every fourth character following it is a token delimiter and the last token is a symbol; but if the first character is numeric, only spaces and parentheses are delimiters and any contiguous string of alphas unbroken by a delimiter is a symbol."—hopeless.
- The above is an example of a **context-dependent language**. Languages with more virtuous properties are called **context-free**.
- We're not dealing with general purpose programming languages here but with simpler minilanguages such as human-interface languages and internal hidden languages.
- The only excuse for context dependencies is that they were inherited.
- If it's an internal language then it can, and should, be changed to remove such dependencies.
- If it's a human-interface language, then the context dependencies must be ripped out because humans can't really deal with such command structures.

LOGIC BASED TESTING

(1) Motivational Overview:

(i) Programmers and Logic:

- Logic is used in programming.
- Logic in its simple form is Boolean algebra.

(ii) Hardware logic testing:

- Hardware logic test design is automated.
- Many test methods developed for hardware logic can also be adapted to software logic testing.

(iii) Specification Systems and Languages:

- We need Specifications and requirements in test development and programming development.
- As programming and test techniques have improved the bugs shifted to requirements and their specifications.
- These bug range from 8% to 30% of the total.
- The trouble with specification is that they are very hard to express. So Boolean algebra is used for all logic systems.
- Higher order logic systems are needed and used for formal specifications.

(iv) Knowledge based systems or Expert System:

- A system which is based on knowledge is known as knowledge based systems.
- The knowledge based systems is also needed in a programming construct.
- The knowledge based systems is also come from a domain such as medicine, law or civil engineering.
- One implementation of knowledge based system 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 then processed through the rule.
- The processing is done by a program called the inference engine.

(v) Overview:

- We start with decision tables because they are extensively used in business data processing.
- Next Boolean algebra is used.

(2) Decision Tables:

(i) Definition and Notation

- A decision table is a tabular form that consists of a set of conditions and their respective actions. The decision tables provide a useful basis for program and test design.
- It consists of four parts they are
  1. Condition Stub
  2. Action Stub
  3. Condition entry
  4. Action entry.
- The condition stub is a list of names of conditions. The action stub consists of a list of names of actions
- Each column of the table consists of a rule.
- A rule specifies whether a condition should or should not be met.
- YES means the condition must met. NO means the condition does not be met and I means that the condition plays no part in the rule or it is immaterial to that rule.

- If the condition is met and if the action entry is YES then the action will taken place, if NO the action will not taken place.

Condition entry →

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

↑ Condition Stub ↓ ↑ Action Stub ↓

Action entry →

- From the above table, 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. So replace condition with predicate.
- If a condition is met then the predicate is true. Similarly for not met is false.
- Now we can say that Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1) or if predicates 1,3 and 4 are true (rule 2).
- Action 2 will be taken if all the predicates are false (rule 3).
- Action 3 will be taken place if predicate 1 is false and predicate 4 is true (rule 4).
- Here we need a default rule that specifies the default action to be taken when all other rules fail. The default rules for the above table are show below.

	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

- If the set of rules covers all the combinations of TRUE / FALSE (YES/ NO) for the predicates, a default specification is not needed.

(ii) Decision-Table Processors

- Decision tables can be automatically translated into code and decision table represent higher level language. The decision table's translator checks the source decision table for consistency and completeness and fills in any default rules.
- First it observes rule1. If the rule is satisfied, the corresponding action is executed.
- Otherwise rule 2 is tried. This process is repeated until a rule is satisfied or no rule is satisfied.
- If the rule is satisfied then the corresponding action will take place. If the rule is not satisfied then the default action taken place.

- The advantages of using decision tables are: it provides clarity, it provides relation to specification, and it provides maintainability. The main drawback is object code inefficiency.

(iii) Decision-Tables as a basis for Test case Design:

- If a specification is implemented as a decision table, then decision tables are used for test case design.
- Similarly, if a program's logic is implemented as decision tables, then decision tables also used for test case design.
- If this is so, then the consistency and completeness of the decision table is checked by the decision table processor.
- It is not desirable to implement program as decision table because restrictions in decision table language.
- The following are restrictions.
  1. The specifications are specified.
  2. The order in which the predicates are evaluated does not effect the resulting action.
  3. The order in which the rules are evaluated does not effect the resulting action.
  4. Once a rule is satisfied and an action is executed, no other rule need to be examined.
  5. If several actions can result from satisfying a rule, the order in which the actions are executed does not matter.
- It is clear from the above restrictions that action selected is based on the combination of predicate truth values. Let us consider an automatic teller machine.
- The first condition is that the card should be valid.
- The second condition is the correct password should be entered.
- The third condition is that the sufficient money should be present in the account.
- Depending on the conditions, respective actions are executed.

(iv) Expansion of Immaterial Cases:

- In decision table immaterial entries are denoted by 'I '.
- If there are n predicates in the decision table then 2^n combination of truth values should be considered.
- The expansion is done by converting each I entry into two entries one with YES and other with NO. Each I entry in a rule double the number of cases.

	← Rule 2 →		← Rule 4 →			
	RULE 2.1	RULE 2.2	RULE 4.1	RULE 4.2	RULE 4.3	RULE 4.4
CONDITION 1	YES	YES	NO	NO	NO	NO
CONDITION 2	YES	NO	YES	YES	NO	NO
CONDITION 3	YES	YES	NO	YES	YES	NO
CONDITION 4	YES	YES	YES	YES	YES	YES

- In the previous table rule 2 contains one I entry and therefore it expands into two equivalent sub rules.
- Rule 4 contains two I entries and therefore it expands into four equivalent sub rules.
- The expansion of rules 2 and 4 are shown in the above table.
- The following table is an example of an inconsistent specification in which the expansion of two rules gives a contradiction.
- Here rules 1 and 2 are contradictory, because two column entries 1.2 & 2.3 are same.
- Therefore action 1 or action 2 is taken depending on which rule is evaluated first

	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 2.3	RULE 2.4
CONDITION 1	YES	YES	YES	YES
CONDITION 2	YES	NO	NO	NO
CONDITION 3	YES	YES	YES	NO
CONDITION 4	NO	NO	NO	NO
ACTION 1	YES	YES	NO	NO
ACTION 2	NO	NO	YES	YES

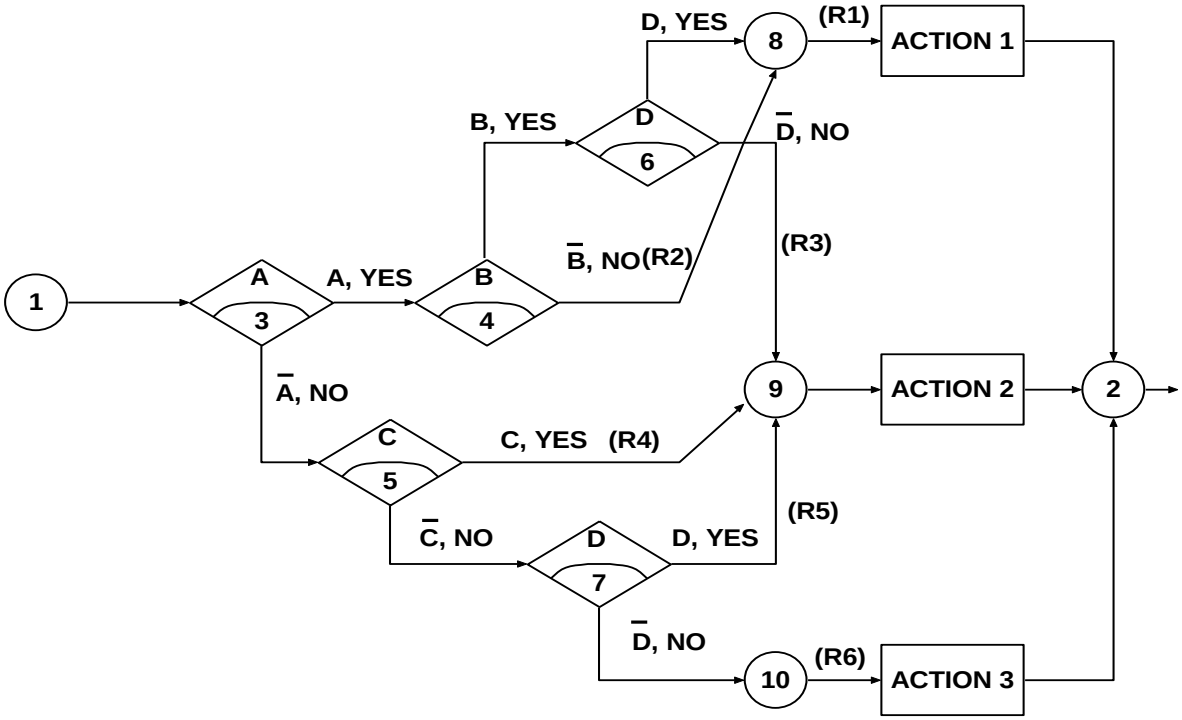
(v) Test case Design:

- Test case design by decision tables starts with examining the specification's consistency and completeness.
- This is done by expanding all immaterial cases and checking the expanded tables.
- Once the specification is verified next to show the correct action.
- The following rules are followed while designing test cases.

1. If there are k rules over n-binary predicates, there are atleast k cases and at most 2^n cases
2. The order in which the conditions are evaluated cannot be altered. But if the order is to be altered then the test cases are increased.
3. The order in which the rules are evaluated cannot be altered. But if the order is to be altered then the rules are interchanged pair wise and tested.
4. Identify the places where the rules are invoked.
5. Identify the places where the actions are initiated.

(vi) Design Tables and Structure:

- The main purpose of a decision table is to check the structure of a program.
- It can be represented in the form of a decision tree.
- The following figure shows a program segment that consists of a decision tree.



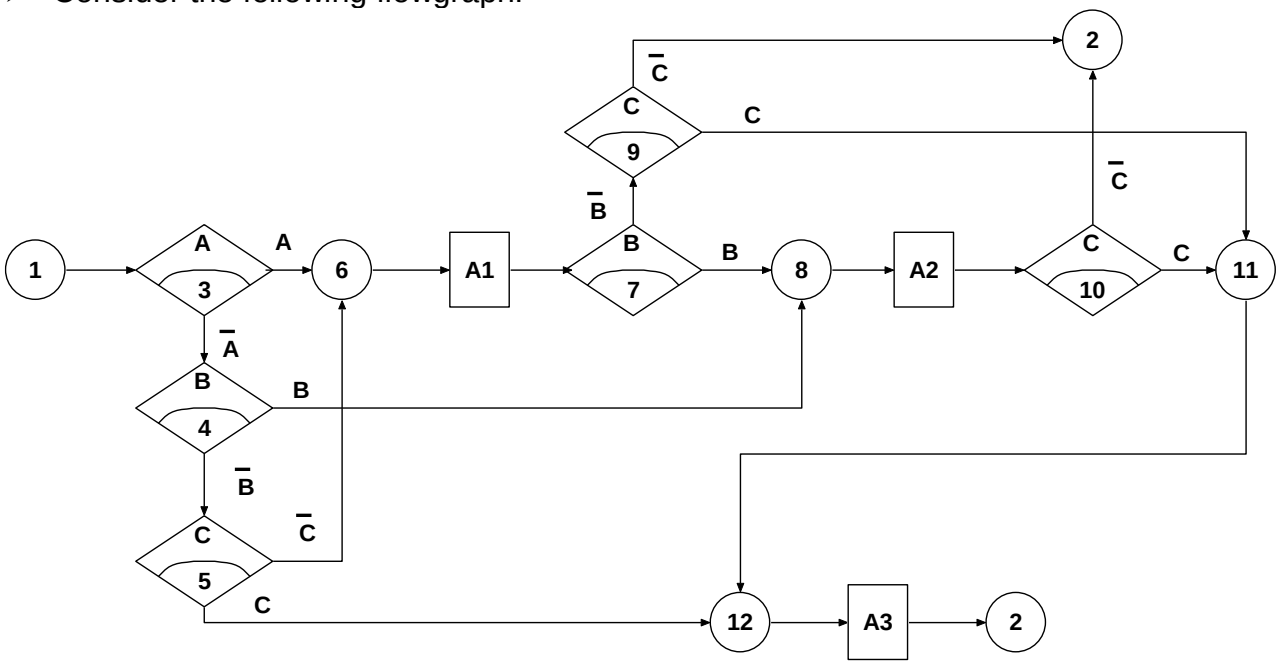
- The decision table corresponding to the above figure is.

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

- If the decision appears on a path put YES or NO.
- If the decision does not appear on the path, put I.
- Rule 1 does not contain decision C, therefore its entries are YES, YES, I, YES.
- Expanding the immaterial cases for the above table is shown in the following table.

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

- Sixteen cases are represented in the previous table and no cases appear twice.
- Therefore the flowgraph appears to be complete and consistent.
- Count the number of Y's and N's in each row. They should be equal.
- Consider the following flowgraph.



1. If condition A is met, do process A1. If condition B is met, do process A2
2. If condition C is met, do process A3
3. If none of the condition is met, do process A1, A2, and A3.
4. When more than one process is done, process A1 must be done first, then A2 and then A3.

- The following table shows the conversion of this flowgraph into a decision table.

	$\overline{A}\ \overline{B}\ \overline{C}$	$\overline{A}\ \overline{B}\ C$	$\overline{A}\ B\ C$	$\overline{A}\ B\ \overline{C}$	$A\ B\ \overline{C}$	$A\ B\ C$	$A\ \overline{B}\ C$	$A\ \overline{B}\ \overline{C}$
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

(3) Path Expressions:

(1) General:

(i) Model:

- Logic based testing is a structural testing when it is applied to structure and it is functional testing when it is applied to a specification.
- In logic based testing we focus on the truth values of control flow predicates.

(ii) Predicates and Relational Operators:

- Predicate is defined as a process which gives truth value as its output.
- Predicates are based on relational operators such as >, >=, =, < , <=
- The other relational operators are is a member of, is a subset of, is a substring of, is a sub graph of etc.

(iii) Case statements and Multivalued Logics :

- Predicates are not restricted to binary truth values (TRUE/ FALSE).
- There are multiway predicates, or multivalued logic.
- Multiway predicates include FORTRAN's 3-way, if case statements.
- Multivalued logic includes post algebra which is responsible for evaluating the structure of predicates. These post algebra logics are very difficult to implement.

(iv) What goes wrong with predicates :

- There are many situations where something can go wrong with predicates.

1. The wrong relational operator is used. Eg. > instead of <=
2. The predicate expression of a compound predicate is incorrect. Eg. A + B instead of AB
3. The wrong operands are used. Eg A>X instead of A > Z
4. If there is a process that leads to faulty predicate.

- The first two errors can be found using logic based testing, where as last two errors can be detected using data flow testing.

(v) Overview :

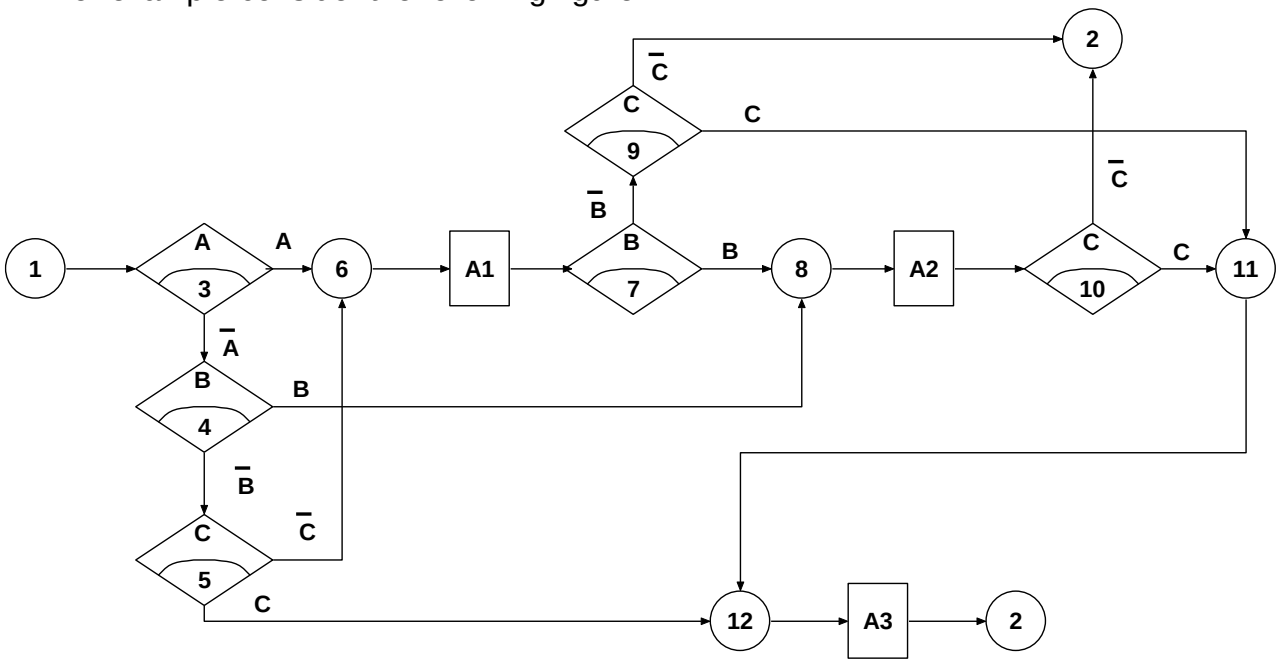
- We start by generating path expressions by path tracing. This time we convert the path expressions into Boolean algebra, using the predicates truth values as weights.

(2) Boolean Algebra:

(i) Notation:

- There are only two numbers in Boolean algebra i.e. Zero (0) and One (1).
- One means always true and zero means always false.

- Label each decision with an upper case letter that represents the truth value of the predicate.
- The YES or TRUE branch is labeled with a letter and the NO or FALSE branch with the same letter overscored.
- For example consider the following figure.



- In the above figure the straight through path which gives via nodes 3,6,7,8,10,11,12,2 has a truth value of ABC.
- The path via nodes 3,6,7,9,2 has a value of ABC
- If two or more paths merge at a node then it is expressed by use of a plus sign (+) which means OR.
- Using the above we can write

$N6 = A + \overline{A}\,\overline{B}\,\overline{C}$

$N8 = (N6)\,B + \overline{A}\,B$

$N11 = (N8)\,C + (N6)\,\overline{B}\,C$

$N12 = N11 + \overline{A}\,\overline{B}\,C$

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

(ii) The rules of Boolean Algebra:

- Boolean algebra has three operators.
- x means AND. Also called multiplication. A statement such as AB means A and B both true.
- + means OR. Also called addition. A statement such as A + B mean either A is true or B is true or both.
- $\overline{A}$ means NOT. Also called negation or complementation.
- Ex $\overline{A}$ is true only when statement A is false.
- The Laws of Boolean algebra is shown below.

1. : $A + A = A$

   $\overline{A} + \overline{A} = \overline{A}$

2. : $A + 1 = 1$

10 $A\,\overline{A}=0$

11. $\overline{\overline{A}} = A$

12. $\overline{0} = 1$

3. : $A + 0 = A$
4. : $A + B = B + A$
5. : $A + \overline{A} = 1$
6. : $A\ A = A$
   $\overline{A}\ \overline{A} = \overline{A}$
7. : $A \times 1 = A$
8. : $A \times 0 = 0$
9. : $AB = BA$
13. $\overline{1} = 0$
14. De Morgan's Law: $\overline{A + B} = \overline{A}\,\overline{B}$
15. $\overline{A\,B} = \overline{A} + \overline{B}$
16. Distributive Law: $A\,(B + C) = AB + AC$
17. $(AB)\,C = A(BC)$
18. $(A + B) + C = A + (B + C)$
19. $A + \overline{A}\,B = A + B$
20. $A + AB = A$

- Individual letters in a Boolean algebra expression are called literals.
- The product of several literals is called a product form (eg: ABC, DE).

(iii) Examples:

- The path expressions are simplified by applying the rules.

$N6 = A + \overline{A}\ \overline{B}\ \overline{C}$

$= A + \overline{B}\,\overline{C}$ [since let $D = \overline{B}\,\overline{C}$, $A + \overline{A}\,\overline{B}\,\overline{C} = A + \overline{A}\,D = A + D = A + \overline{B}\,\overline{C}$]

$N8 = (N6)\,B + \overline{A}\,B$ $= (A + \overline{B}\,\overline{C})\,B + \overline{A}\,B$

$= A\,B + \overline{B}\,\overline{C}\,B + \overline{A}\,B$ $= (A\,B + B\,\overline{B}\,\overline{C}) + \overline{A}\,B$

$= AB + 0\,\overline{C} + \overline{A}B$ $= AB + \overline{A}\,B$

$= (A + \overline{A})\,B$ $= 1 \times B$

$= B$

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

$= B\,C + A\,\overline{B}\,C + 0$ $= C\,(B + A\,\overline{B})$

$= C\,(B + \overline{B}A)$ $= C\,(B + A)$

$= C\,B + C\,A$ $= AC + BC$

$N12 = N11 + \overline{A}\,\overline{B}\,C$

$= AC + BC + \overline{A}\,\overline{B}\,C$ $= BC + \overline{A}\,\overline{B}\,C + AC$

$= C\,(B + \overline{A}\,\overline{B}) + AC$ $= C\,(\overline{A} + B) + AC$

$= C\overline{A} + AC + BC$ $= C(A + \overline{A}) + BC$

$= C\,(1) + BC$ $= C + BC$

$= C\,(1 + B)$ $= C(1)$

$= C$

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

$= C + B\,\overline{C} + (A + \overline{B}\,\overline{C})\,\overline{B}\,\overline{C}$

$= C + B\,\overline{C} + A\,\overline{B}\,\overline{C} + \overline{B}\,\overline{C}\,\overline{B}\,\overline{C} = C + B\,\overline{C} + A\,\overline{B}\,\overline{C} + \overline{B}\,\overline{C}$

$= C + B\overline{C} + \overline{B}\overline{C}(1+A)$ $= C + B\overline{C} + \overline{B}\overline{C}$

$= C + \overline{C}(B + \overline{B})$

$= C + \overline{C}(1)$

$= C + \overline{C}$

$= 1$

(iv) Paths and domains:

- Consider a loop free entry / exit path and assume all predicates are simple.
- Each predicate on the path is denoted by a capital letter either overscored or not.
- The result is a term that consists of the product of several literals. For ex: $\overline{A}$ B C.
- If a literal appears twice in a product term then one appearance can be removed and the decision is redundant. For ex: consider C C, $\overline{B}$ $\overline{B}$ here we have to take only one C & one $\overline{B}$
- If a literal appears both barred and un barred in a product term then the term is equal to zero and the path is un achievable.
- A product term on an entry / exit path specifies a Domain.
- For compound predicates there is a provision of separate path for each product term.
- For example, we can implement ABC + DEF +GH as one path using a compound predicate or as three separate paths i.e. ABC, DEF, GH and specify three separate domains.
- Let us say we have a specification such that there is one and only one product term for each domain then represent these domains as D_1, D_2, D_3, ……D_m.
- Consider any of these product terms D_i, D_j.
- For every i not equal to j, D_i, D_j equal to zero. If not equal to zero, then there is an overlap of the domains which is a contradictory domain specification.
- The sum of all the D_i must equal to 1 else there is an ambiguity.

(v) Test case design:

- Let us consider a hierarchy of test cases for a loop that has a compound predicate.
- The routine has a single entry and single exit and has no dead end code.
- Because the predicates may be compound, the Boolean algebra expression of a domain will be a sum of products after simplification.
- We can build a hierarchy of test strategies by considering how we test for each domain.
- Here consider
  1. Simplest: Use any prime implicant in the expression. Suppose ABC + AB + DEF reduces by AB + DEF, then AB, DEF are called prime implicant.
  2. Prime implicant cover: Pick input values so that there is at least one path for each prime implicant at the node.
  3. All Terms: Test all expanded terms for that node. For example in previous figure the node 6 has five terms.

$$N6 = A + \overline{A}\,\overline{B}\,\overline{C}$$

$$= AB(C + \overline{C}) + A\overline{B}(C + \overline{C}) + \overline{A}\,\overline{B}\,\overline{C}$$

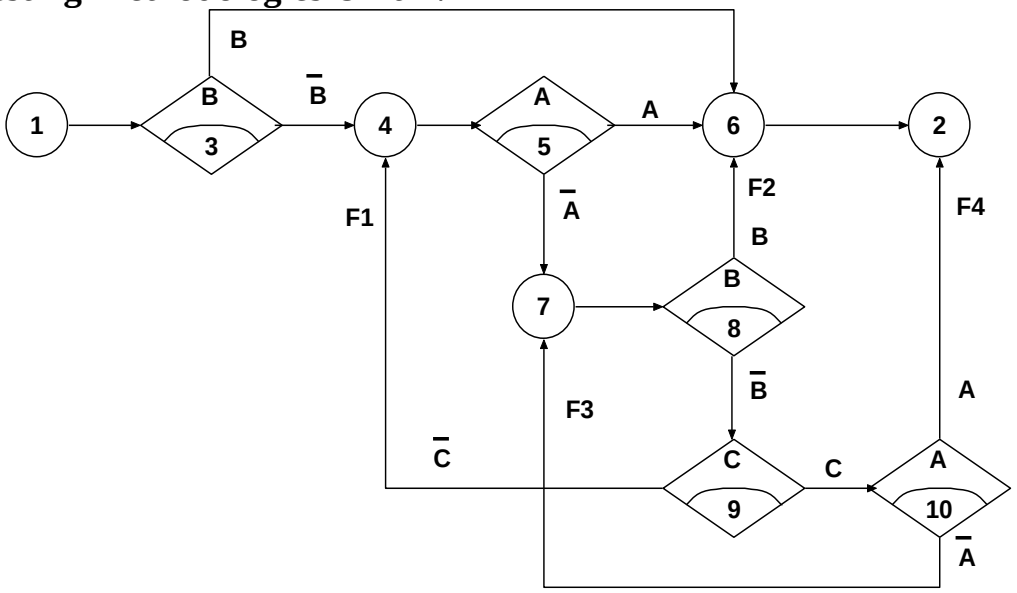
$$= A\,B\,C + A\,B\,\overline{C} + A\,\overline{B}\,C + A\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C}$$

Here there are totally five terms. Similarly for node 8 has 4 terms & node 12 has 4 terms. There is at least one path for each term.

  4. Path dependence: Because in general the truth value of a predicate is obtained by interpreting the predicate, its value may depend on the path taken there.

(3) Boolean equations:

- Loops complicate things because we may have to solve a Boolean equation to determine what predicate value combinations lead to where. Consider the following flowgraph



- Here the link name F1, F2, F3, F4 represents the Boolean expression corresponding to that link.

$N4 = \overline{B} + F1$

$= \overline{B} + (N7)\,\overline{B}\,\overline{C}$

$= \overline{B}\,(1 + (N7)\,\overline{C})$

$= \overline{B}$

$N6 = (N4)\,A + B$

$= \overline{B}\,A + B$

$= A + B$

$N7 = (N4)\,\overline{A} + F3$

$= \overline{A}\,\overline{B} + (N7)\,\overline{B}\,C\,\overline{A}$

$= \overline{A}\,\overline{B}$

$N2 = N6 + F4$

$= A + B + (N7)\,A\,\overline{B}\,C$

$= A + B\,(1 + (N7)\,A\,\overline{B}\,C)$

$= A + B$

Example:

(1) Demonstrate by means of truth tables the validity of the following theorems of Boolean Algebra.

(i) Associative laws

(ii) De Morgan's theorems for three variables

(iii) Distributive law of + over.

(Ans) **(i) Associative laws**

(a) Associative law of addition

$(A + B) + C = A + (B + C)$

Let TRUE= T & FALSE = F then (A + B) +C & A + (B +C) is given by

A	B	C	(A+B)	(A+B)+C	(B+C)	A+(B+C)
T	T	T	T	T	T	T
T	T	F	T	T	T	T
T	F	T	T	T	T	T
T	F	F	T	T	F	T
F	T	T	T	T	T	T
F	T	F	T	T	T	T
F	F	T	F	T	T	T
F	F	F	F	F	F	F

From the above table it shows that

$(A + B) + C = A + (B + C)$

(b) Associative law of multiplication

$(A \times B) \times C = A \times (B \times C)$

Let TRUE= T & FALSE = F then $(A \times B) \times C$ & $A \times (B \times C)$ is given by

A	B	C	(AxB)	(AxB)xC	(BxC)	Ax(BxC)
T	T	T	T	T	T	T
T	T	F	T	F	F	F
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	F	F	T	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

From the above table it shows that

$(A \times B) \times C = A \times (B \times C)$

(ii) De Morgan's law

(a) $\overline{(A + B) + C} = \overline{A}\ \overline{B}\ \overline{C}$

Let TRUE= T & FALSE = F then $\overline{(A + B) + C}$ & $\overline{A}\ \overline{(B}\ \overline{C)}$ is given by

A	B	C	(A+B)	(A+B)+C	$\overline{(A + B) + C}$
T	T	T	T	T	F
T	T	F	T	T	F
T	F	T	T	T	F
T	F	F	T	T	F
F	T	T	T	T	F
F	T	F	T	T	F
F	F	T	F	T	F
F	F	F	F	F	T

A	B	C	$\overline{A}$	$\overline{B}$	$\overline{C}$	$(\overline{A} \times \overline{B})$	$(\overline{A} \times \overline{B}) \times \overline{C}$
T	T	T	F	F	F	F	F
T	T	F	F	F	T	F	F
T	F	T	F	T	F	F	F
T	F	F	F	T	T	F	F
F	T	T	T	F	F	F	F
F	T	F	T	F	T	F	F
F	F	T	T	T	F	T	F
F	F	F	T	T	T	T	T

From the above two tables it is clear that

$$\overline{(A + B) + C} = \overline{A} \times (\overline{B} \times \overline{C})$$

(b) $(A \times B) \times C = (\overline{A} + B) + \overline{C}$

Let TRUE= T & FALSE = F then $\overline{(A \times B) \times C}$ & $\overline{A} + (\overline{B} + \overline{C})$ is given by

A	B	C	(AxB)	(AxB)xC	$\overline{(A \times B) \times C}$
T	T	T	T	T	F
T	T	F	T	F	T
T	F	T	F	F	T
T	F	F	F	F	T
F	T	T	F	F	T
F	T	F	F	F	T
F	F	T	F	F	T
F	F	F	F	F	T

A	B	C	$\overline{A}$	$\overline{B}$	$\overline{C}$	$(\overline{A} + \overline{B})$	$(\overline{A} + \overline{B}) + \overline{C}$
T	T	T	F	F	F	F	F
T	T	F	F	F	T	F	T
T	F	T	F	T	F	T	T
T	F	F	F	T	T	T	T
F	T	T	T	F	F	T	T
F	T	F	T	F	T	T	T
F	F	T	T	T	F	T	T
F	F	F	T	T	T	T	T

From the above two tables it is clear that

$$\overline{(A \times B) \times C} = \overline{A} + (\overline{B} + \overline{C})$$

(iii) Distributive law of + over

Distributive law of + over

$$A + (B \times C) = (A + B) \times (A + C)$$

Let TRUE= T & FALSE = F then $A + (B \times C)$ & $(A + B) \times (A + C)$ is given by

A	B	C	(BxC)	A+(BxC)	(A+B)	(A+C)	(A+B) x (A+C)
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

From the above table it shows that

$$A + (B \times C) = (A + B) \times (A + C)$$

(2) Demonstrate by means of truth tables the validity of the following theorems of Boolean Algebra.

(i) Commutative laws
(ii) Absorption law
(iii) Idempotency laws

(i) Commutative laws

(a) Commutative law of addition

A + B = B + A

Let TRUE= T & FALSE = F then A + B & B +A is given by

A	B	A+B	B	A	B+A
T	T	T	T	T	T
T	F	T	F	T	T
F	T	T	T	F	T
F	F	F	F	F	F

From the above table it shows that A + B = B + A

(b) Commutative law of multiplication

A x B = B x A

Let TRUE= T & FALSE = F then A x B & B x A is given by

A	B	AxB	B	A	BxA
T	T	T	T	T	T
T	F	F	F	T	F
F	T	F	T	F	F
F	F	F	F	F	F

From the above table it shows that A x B = B x A

(ii) Absorption law

Absorption law

$A + \overline{A}\,B = A + B$

Let TRUE= T & FALSE = F then $A + \overline{A}\,B$ & A +B is given by

A	$\overline{A}$	B	$\overline{A}$ x B	A + $\overline{A}$ x B	A+B
T	F	T	F	T	T
T	F	F	F	T	T
F	T	T	T	T	T
F	T	F	F	F	F

From the above table it shows that $A + A\,B = A + B$

(iii) Idempotency laws

Idempotency law of addition $A + A = A$; $\overline{A} + \overline{A} = \overline{A}$

Idempotency law of multiplication $A \times A = A$; $\overline{A} \times \overline{A} = \overline{A}$

Let TRUE= T & FALSE = F

A	A	A+A	$\overline{A}$	$\overline{A}$	$\overline{A} + \overline{A}$	A x A	$\overline{A} \times \overline{A}$
T	T	T	F	F	F	T	F
F	F	F	T	T	T	F	T
T	T	T	F	F	F	T	F
F	F	F	T	T	T	F	T

From the above table it shows that $A + A = A$; $\overline{A} + \overline{A} = \overline{A}$

$A \times A = A$; $\overline{A} \times \overline{A} = \overline{A}$

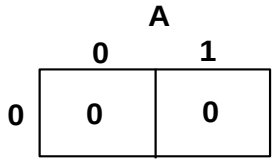
(4) KV Charts:

(i) The Problem:

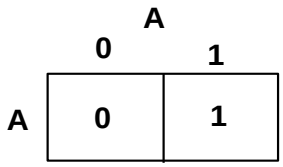
- The Karnaugh-Veitch chart is known by combination of Karnaugh and Veitch with any one of map, chart, and diagram. This chart reduces Boolean algebraic manipulations to graphical trivia.
- Beyond six variables these diagrams get cumbersome and other techniques such as the Quine-McCluskey method should be used.

(ii) Simple Forms:

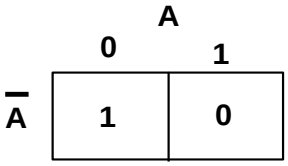
- The following figure shows all the Boolean functions of a single variable A and their equivalent representation as a KV chart.



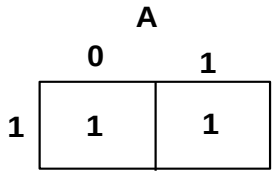
The function is never true



The function is true when A is true

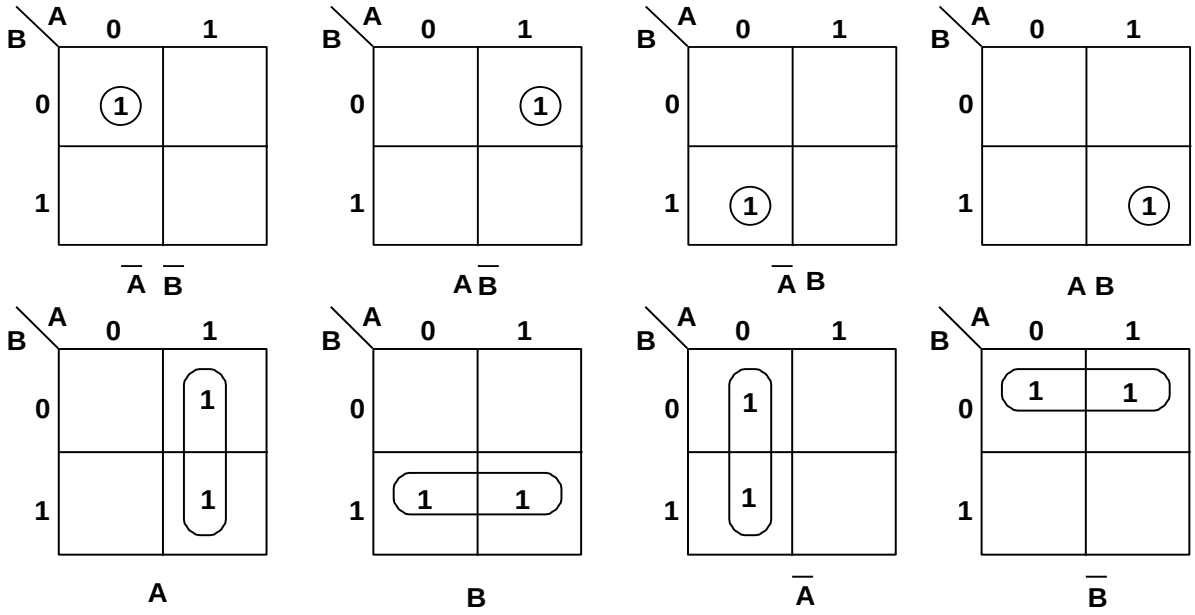


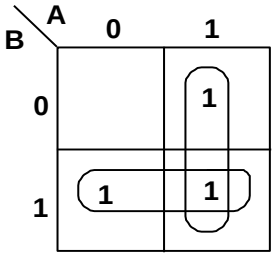
The function is true when A is false



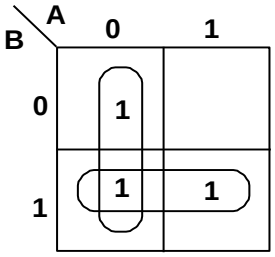
The function is always true

- The following figure shows sixteen possible functions of two variables.

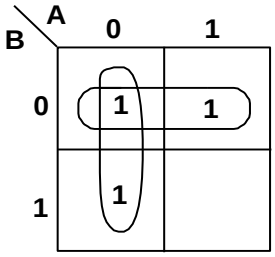




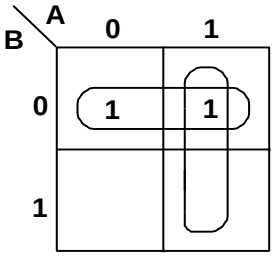
$A + B$



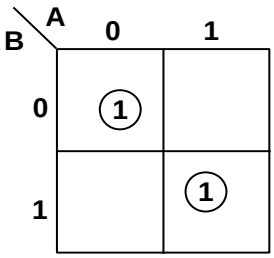
$\overline{A} + B$



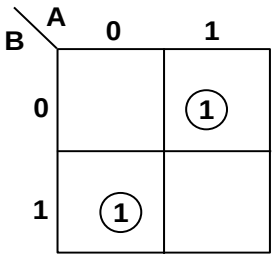
$\overline{A} + \overline{B}$



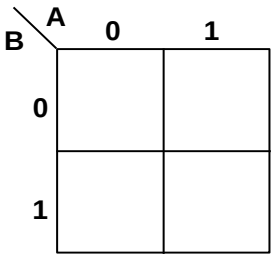
$\overline{B} + A$



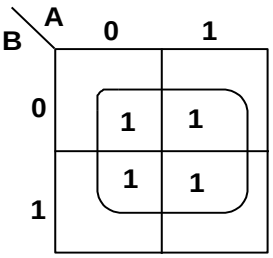
$\overline{A}\ \overline{B} + AB$



$A\ \overline{B} + \overline{A}\ B$



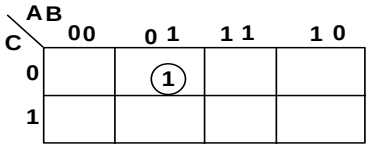
Universal False



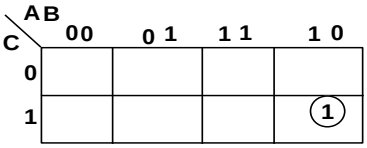
Universal True

(iii) Three Variables:

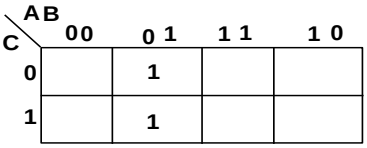
- KV charts for three variables are shown below. A few examples are shown.



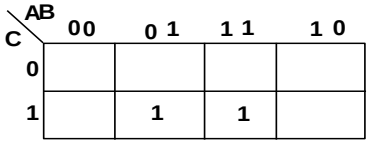
$A\ B\ C$



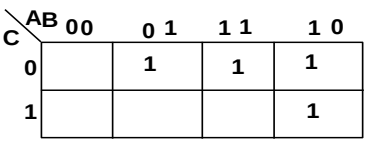
$A\ B\ C$



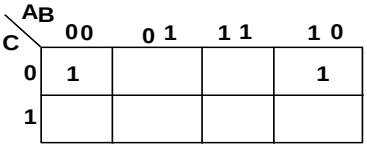
$A\ B$



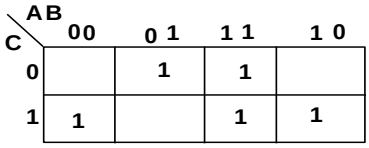
$B\ C$



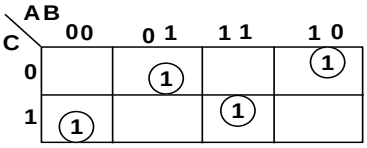
$B\ C + A\ B$



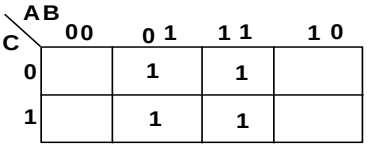
$B\ C$



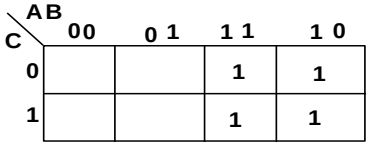
$B\ C + A\ B + B\ C$



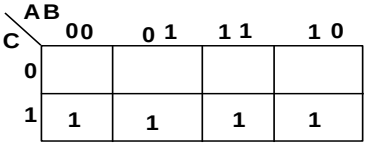
$ABC + ABC + ABC + ABC$



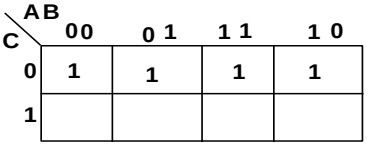
B



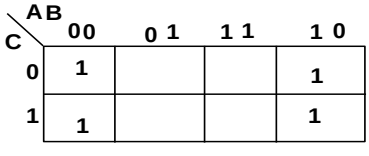
A



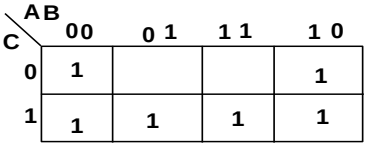
C



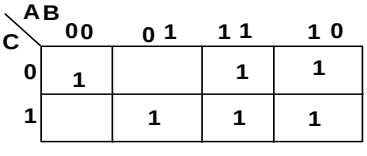
C



B



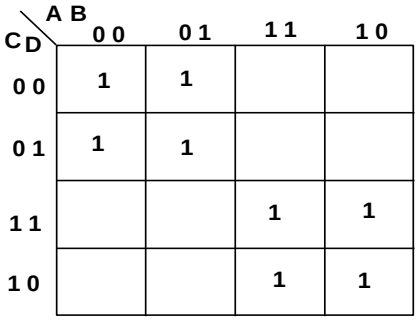
$B + C$



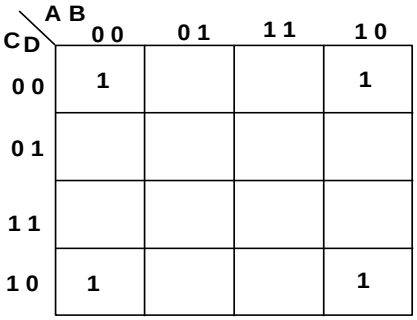
$A + BC + BC$

(iv) Four Variables:

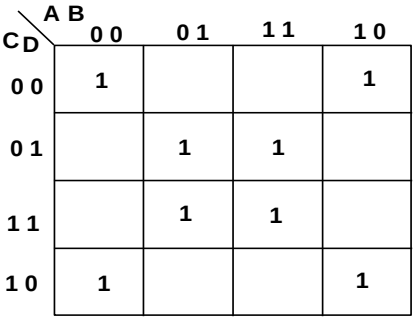
- The same principles hold for four or more variables



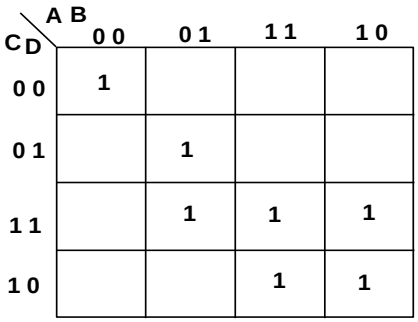
AC + AC



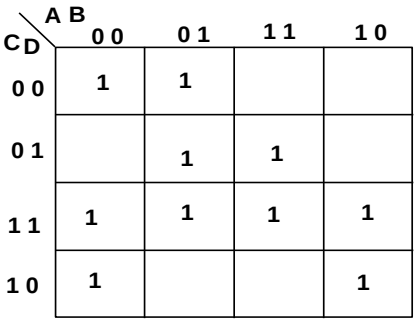
BD



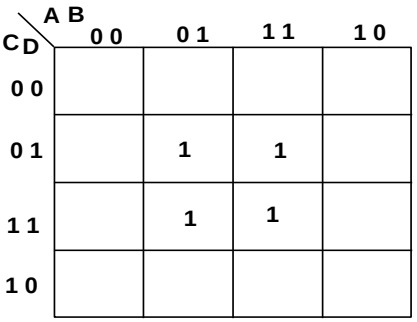
BD + BD



ABCD + ABD + AC



ABD + BD + BC



BD

Examples:

(i) Using a Karnaugh map minimize

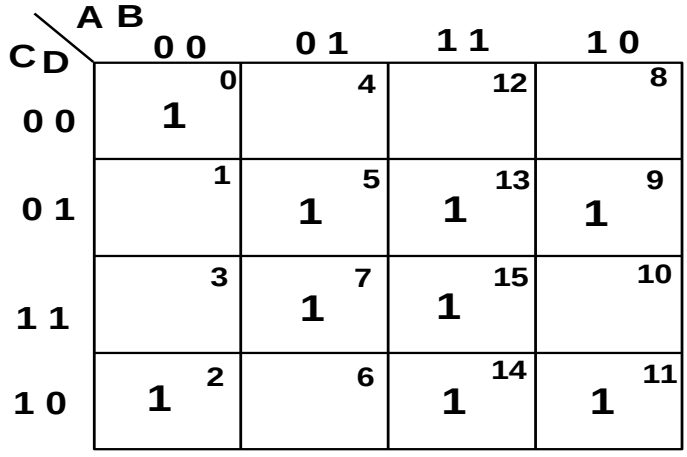
F= A B C D + A B C D + A B C D + A B C D + A B D + B C D + A B C D

Ans: The Standard SOP form is:

F(A,B,C,D)=A B C D + A B C D + A B C D + A B C D + A B D (C + C) + (A + A) B C D + A B C D

= A B C D + A B C D + A B C D + A B C D + A B C D + A B C D + A B C D

+ A B C D + A B C D



The minimized function is: A B D + B D + A C D + A C D

(ii) Minimize the function using Karnaugh map method

F(A,B,C,D)= ∑(1,2,3,8,9,10,11,14) + ∑d (7,15)

Ans:

CD \ AB	00	01	11	10
00	0	4	12	8 1
01	1 1	5 1	13	9 1
11	3 1	7 d	15 d	10 1
10	2 1	6	14 1	11 1

The minimized function is: A B + A C + A B D + A B C

(iii) Reduce the following function using Karnaugh Map method

F(A,B,C,D)= π(4,5,6,7,8,12,13) + d(1,15)

Ans:

CD \ AB	00	01	11	10
00	0	4 0	12 0	8 0
01	1 d	5 0	13 0	9
11	3	7 0	15 d	10
10	2	6 0	14	11

The minimized function is: (B + D) (A + B) (A + C +D)

(5) Specifications:

(i) General:

- Using KV charts specification is validated. The procedure is given below.
  1. Rewrite the specification with consistent language.
  2. Identify the predicates. Name with suitable letters such as A, B, C,..
  3. After predicate identification, rewrite the specification into logical or Boolean connectives such as AND, OR, NOT.
  4. This rewritten specification is then transformed into set of Boolean expressions.
  5. Identify the default action if any.

6. Enter the Boolean expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps.
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 back and get a clarification, explanation or revision.
10. If the default cases were not specified explicitly, translate the default cases back and get a confirmation.

(ii) Finding and translating the logic:

- The formation of specifications into sentences is given below.
- Specifications are formed into sentences by using the following IF-THEN format.
- IF represents predicate, THEN represents action.
- Hence predicates are used by applying certain Boolean connectives like AND, OR, and NOT and represented by A1, A2, A3.
- The different phrases which can be used for the words are
  IF: if, if and when, only if, only when, based on, because, but etc.
  THEN: then, assign, shall, should, will, would, do etc.
  AND: all, and, as well as, both, but, in conjunction with, coincidental with etc.
  OR: or, either-or, and, and if..then, and/or, in addition to, otherwise etc.
  NOT: but, but Not, excluding, less, neither, never, besides etc.
  EXCLUSIVE OR: but, by contrast, conversely, nor etc.
  IMMATERIAL: irrelevant, independent of, irregardless, irrespective, whether or not etc.
- Other than these, some other dangerous phrases also exist such as respectively, similarly etc.
- Now we have a specification of the form
  IF A AND B AND C, THEN A1
  IF C AND D AND F, THEN A3
  IF A AND B AND D, THEN A2

(iii) Ambiguities and Contradictions:

- The problem of ambiguity occurs, when more than one action is activated by many boxes of KV chart or any box is empty in KV chart.
- Let us consider an ambiguous specification that is

$$A1 = B\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C}\,D$$
$$= (A + \overline{A})\,B\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C}\,D$$
$$= A\,B\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C}\,D$$

$$A2 = A\,\overline{C}\,\overline{D} + A\,\overline{C}\,D + A\,\overline{B}\,\overline{C} + A\,B\,\overline{C}$$
$$= A(B + \overline{B})\,\overline{C}\,\overline{D} + A(B + \overline{B})\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,(D + \overline{D}) + A\,B\,\overline{C}(D + \overline{D})$$
$$= A\,\overline{B}\,\overline{C}\,\overline{D} + A\,B\,\overline{C}\,\overline{D} + A\,B\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,\overline{D}$$
$$+ A\,B\,\overline{C}\,D + A\,B\,\overline{C}\,\overline{D}$$
$$= A\,\overline{B}\,\overline{C}\,\overline{D} + A\,B\,\overline{C}\,\overline{D} + A\,B\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,D$$

$$A3 = B\,D + B\,C\,\overline{D}$$
$$= (A + \overline{A})\,B\,(C + \overline{C})\,D + (A + \overline{A})\,B\,C\,\overline{D}$$
$$= A\,B\,C\,D + A\,B\,\overline{C}\,D + \overline{A}\,B\,C\,D + \overline{A}\,B\,\overline{C}\,D + A\,B\,C\,\overline{D} + \overline{A}\,B\,C\,\overline{D}$$

ELSE = $\overline{B}\ C + A\ \overline{B}\ \overline{C}\ D$

$=(A + \overline{A})\ \overline{B}\ C\ (D + \overline{D}) + \overline{A}\ \overline{B}\ \overline{C}\ \overline{D}$

$= A\ \overline{B}\ C\ D + A\ \overline{B}\ C\ \overline{D} + \overline{A}\ \overline{B}\ C\ \overline{D} + \overline{A}\ \overline{B}\ C\ D + \overline{A}\ \overline{B}\ C\ \overline{D}$

- Here 1,2,3 represents the actions and the 4th specifies the default case.
- Now represent these specifications as follows.

CD \ AB	00	01	11	10
00	4	1	1,2	2
01		3	2,3	1,2
11	4	3	3	4
10	4	3	3	4

- In this case the ambiguity occurs in the case of $\overline{A}\ \overline{B}\ \overline{C}\ D$, this gives many inconsistent or contradictory solutions.
- There are several boxes that call for more than one action.
- In $A\ B\ \overline{C}\ \overline{D}$ both action 1 and action 2 shall be taken.
- For unspecified default action do the following
  - Insert explicit entries in the KV chart.
  - Apply negation.
  - Provide an equivalent expression as a default statement.

(iv) Don't care and Impossible terms:

- Don't care terms (Ø) are the terms or conditions using which logic is simplified through KV chart.
- The value of Ø can be either 0 or 1.
- Consider the following three impossible things.
  1. Creation of a universal program verifier
  2. Knowing both the exact position and the exact momentum of a fundamental particle.
  3. Knowing what happened before that started the universe.
- Basically impossible conditions are used to simplify the logic.
- The two types of impossible conditions are
  1. The condition cannot be created or improbable
  2. The condition results from forcing a complex continuous one into a binary logical one.

Logic Simplification:

- The steps involved in simplifying the logic are as follows.
  1. Identify all impossible and illogical cases.
  2. Next avail these cases effectively
  3. For this purpose KV chart is used
  4. Use the symbol Ø which is to be interpreted as 0 or 1.

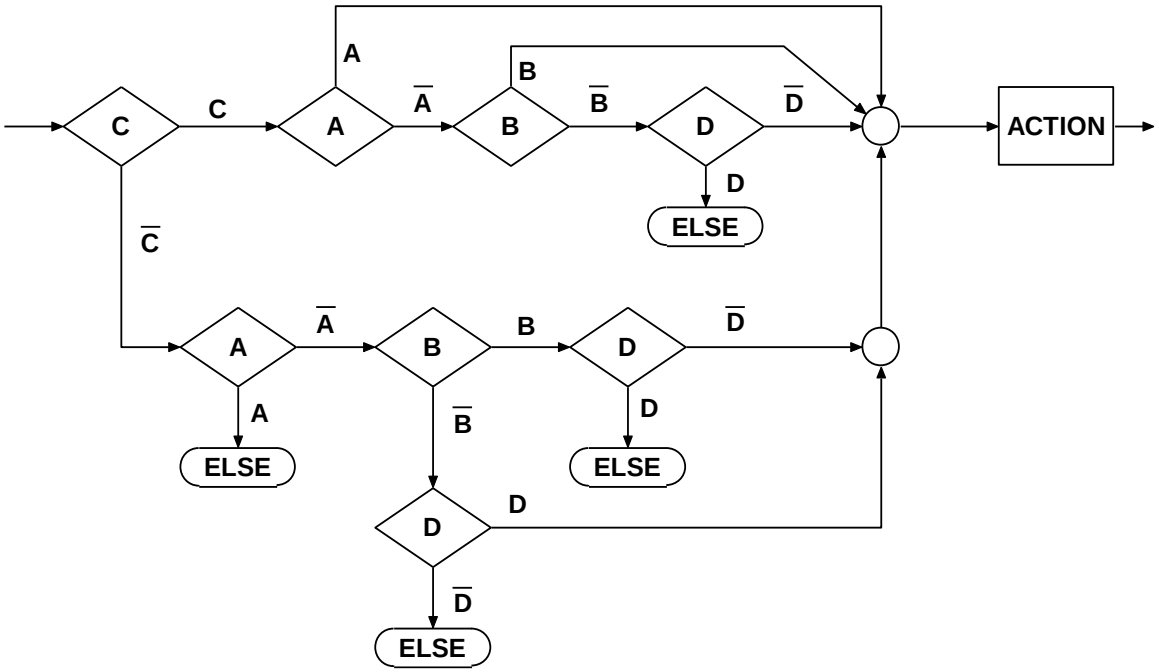
C D \ A B	0 0	0 1	1 1	1 0
0 0	Ø	1		
0 1	1	Ø	Ø	
1 1	Ø	1	1	1
1 0	1	1	1	1

The minimized function is: $C\,\overline{D} + \overline{A}\,B\,\overline{C}\,D + C\,B + C\,A + \overline{A}\,B\,\overline{D}$ ⟶ (1)

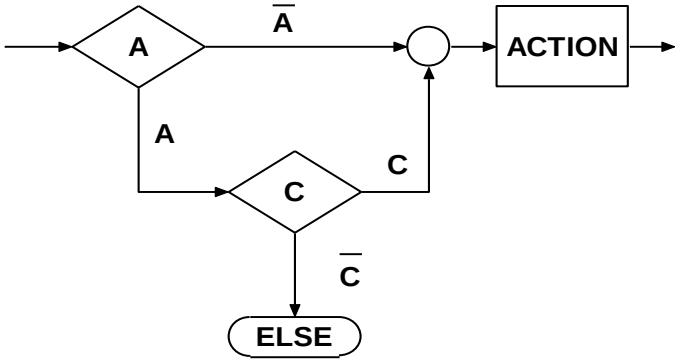
By taking impossible conditions we get $C + \overline{A}$. ⟶ (2)

The corresponding control flowgraphs for equations (1) and (2) are defined as follows.

Control flowgraph for equation (1)



Control flowgraph for equation (2)



UNIT –V
STATES, STATE GRAPHS, AND TRANSITION TESTING

(1) State Graphs:

(i) States(public question)

- State is a condition or situation during which an object undergoes throughout its life time.
- States are represented by nodes.
- States are numbered or identified by characters or words or whatever else is convenient.
- A state graph consists of a set of states in order to represent the behavior of the system.
- To understand the concept of states let us consider the following examples.

Example 1: A program that detects the character sequence ZCZC can be in the following states.

1. Neither ZCZC nor any part of it has been detected.
2. Z has been detected.
3. ZC has been detected.
4. ZCZ has been detected.
5. ZCZC has been detected.

Example 2: A moving automobile whose engine is running can have the following states with respect to transmission.

1. Reverse gear.
2. Neutral gear.
3. First gear.
4. Second gear.
5. Third gear.
6. Four gear.

Example 3: A person's checkbook can have the following states with respect to bank balance.

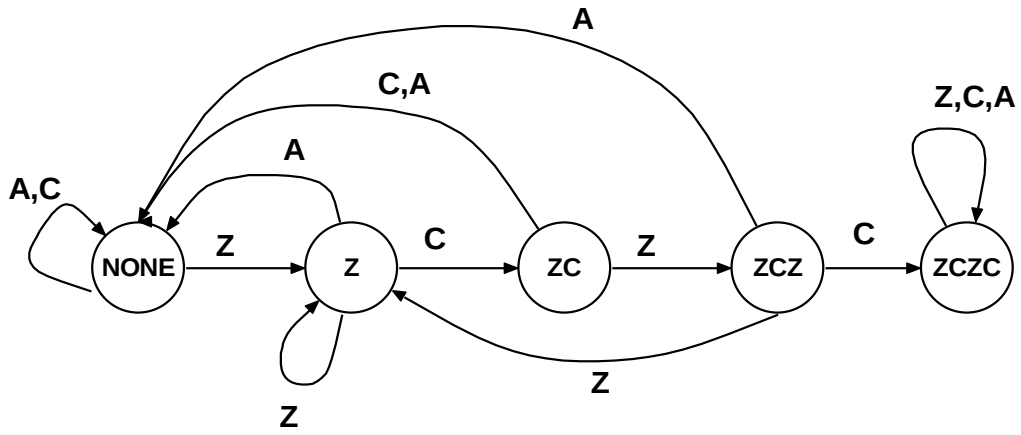
1. Equal.
2. Less than.
3. Greater than.

Example 4: A word processing program menu can be in the following states with respect to file transmission.

1. Create document.
2. Copy document.
3. Delete document.
4. Rename document.
5. Compress document.
6. Saving document
7. Copy disc.
8. Format disc
9. Backup disc
10. Recover from backup

(ii) Inputs and Transitions:(public question)

- Some thing is modeled and given is called input. Input may be values or variables.
- A state graph takes input provided to states.
- As a result of these inputs the state changes is known as transition.
- That is changing from one state to other state is called transition.
- Transitions are denoted by links that join the states.
- The input that causes the transition is represented on the link. So the inputs are link weights.
- A finite state machine is represented by a state graph having a finite number of states and a finite number of transitions between states.
- The ZCZC detection example can have the following types of inputs.
    1. Z
    2. C
    3. Any character other than Z or C which will be denoted by A.



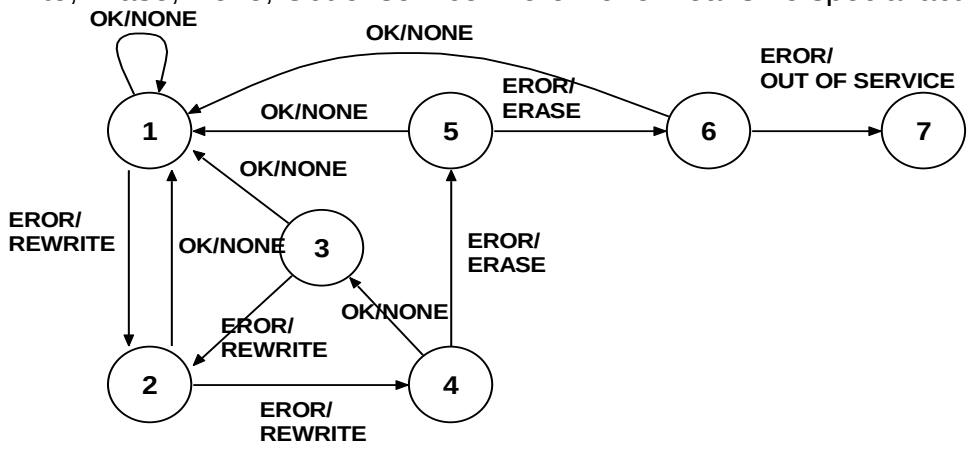
The above state graph is interpreted as follows.

1. If a system is in the NONE state, and it receives A or C then it is in NONE state only.
2. In NONE state if Z is received, the system enters into Z state. In Z state if it receives Z it will remain in the same state. If C is received it will go to the ZC state or if any other character say A is received then it will go back to the NONE state.
3. In ZC state if it receives Z it will enter into ZCZ state. If C or A is received it enter into NONE state.
4. In ZCZ state if it receives Z it enter into the Z state. If A is received it enters into the NONE state.
5. In ZCZ state if it receives C it enter into the ZCZC state. In ZCZC state if it receives Z or C or A then it will remain in the same state only.

(iii) Outputs:

- Outputs are based on the input values.
- When an input is applied to a state it is processed in order to produce an output.
- Each input and output of the state graph is separated by a slash '/' symbol.
- Outputs are also link weights. If more than one input having the same output than it can be represented by input1, input 2, input 3…/output.

Example: Let us consider a tape control recovery system. This system contains two inputs OK & Error. OK means "No write errors". Error means "There may be write errors". The outputs are Rewrite, Erase, None, Out of service. Here None means no special action is taken.



- At state 1 if no write errors are detected (input = OK) no special action is taken (output=NONE). If error is detected (input=ERROR) backspace the tape one block and rewrite the block (output =REWRITE) i.e. enter into state 2.

- At state 2 if the rewrite is successful (input= OK) no action is taken (output=NONE) and return to state 1.
- If the rewrite is not successful try another back space and rewrite (output=REWRITE) i.e. enter into state 4.
- If there are two successive rewrites and a third error occurs then backspace ten centimeters and erase (output=ERASE) i.e. from state 4 to state 5.
- If there are two successive rewrites and a third no error occurs then it enter into state 3 & then state 1. At state 3 if any error is detected then it enter into state 2 and rewrite.
- At state 5 if the erasure works (input=OK) no action is taken and return to initial state.
- If it does not work, backspace another ten centimeters and erase. i.e. enter into state 6.
- At state 6 if the erasure works (input=OK) no action is taken and return to initial state
- If the second erasure does not work put the tape control out of service i.e enter into state 7

(iv) State Table:

- If state graph has a large number of states and transitions, then it is difficult to follow them.
- Therefore a state table is used, as an easiest way to represent all the states, inputs, transitions and outputs of the state graph.
- A state table is defined as a tabular representation of a state graph.
- It consists of
  1. Each row represents a state.
  2. Each column represents an input condition.
  3. The box at the intersection of row and column represents the next state and the output.
- The state table for the tape control system is shown below.

STATE	OK	ERROR
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7	……	……

. **(v) Time Versus Sequence:**

- State graphs don't represent time-they represent sequence.
- A transition might take microseconds or centuries.
- A system may be in one state for milliseconds or years.
- The finite state machine model can be elaborated to include notions of time in addition to sequence, such as Petri nets.

(vi) Software Implementation(public question)

1. Implementation and Operation:

- Here four tables are involved.

1. First table encode the input value. i.e. INPUT_TABLE_CODE.
2. A table that specifies the next state i.e. TRANSITION_TABLE
3. A table that specifies the output. i.e. OUTPUT_TABLE
4. A table that stores the present state of every device. i.e. DEVICE_TABLE.

This routine operates as follows.

```
BEGIN
PRESENT_STATE:=DEVICE_TABLE
ACCEPT INPUT_VALUE
INPUT_CODE:=INPUT_CODE_TABLE
```

```
POINTER:=INPUT_CODE#PRESENT_STATE
NEW_STATE:=TRANSITION_TABLE
OUTPUT_CODE:=OUTPUT_TABLE
CALL OUTPUT_HANDLER
DEVICE_TABLE:=NEW_STATE
END
```

Steps:

1. The present state is fetched from memory.
2. The present input value is fetched. If it is numerical it can be used directly. If it is not numerical encode into a numerical value.
3. The present state and input code are combined.
4. The output table contains a pointer to the routine to be executed.
5. The same pointer is used to fetch the new state value, which is then stored.

2. Input encoding and Input Alphabet:

- Only the simplest finite state machines can use the inputs directly.
- In ZCZC detector there are 256 possible ASCII characters. But we are taken Z, C and OTHER.
- The input encoding here is for OTHER=0, for Z=1, for C=2.
- The different encoded input values are called the input alphabet.

3. Output encoding and Output Alphabet:

- A single character output for a link is rare.
- So we want to output a string of characters.
- These can be encode into a convenient output alphabet.

4. State codes and State-Symbol products:

- The term state-symbol product is used to convert the combined state and input code into a pointer to compact table.

5. Application Comments for Designers:

- An explicit state table implementation is advantageous when either the control function is likely to change in the future or when the system has many similar, but slightly different control functions.

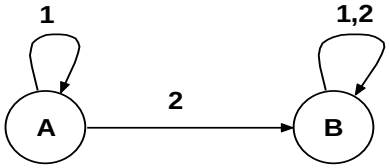
6. Application Comments for Testers(Public Question)

- Independent testers are not usually taken with either implementation details or the economics of this approach.
- If the programmers have implemented an explicit finite state machine then much of our work has been done for us.
- Sometimes showing the programmers the kinds of tests developed from a state graph description can lead them to consider it as an implementation technique.

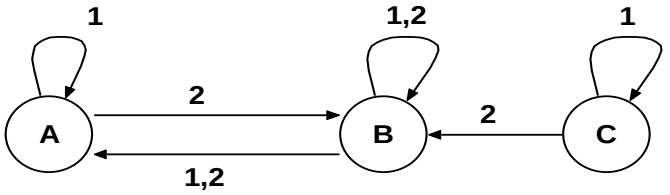
(2) Good State Graphs and Bad State Graphs: (public question)

(i) General:

- In testing we deal with a good state graph and also with a bad one.
- The following figure shows examples of improper or bad state graphs.

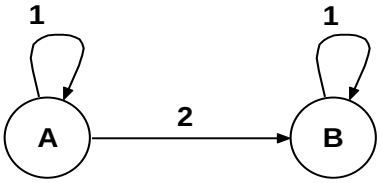


In state B the initial state can never be entered again

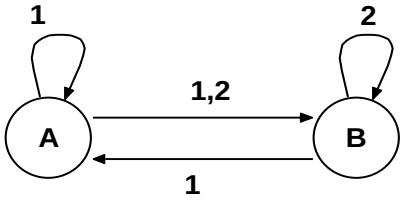


State C cannot be entered

State A,B are not reachable



No transition is specified for an input of 2 when in state B



Two transitions are specified for an input of 1 in state A

(2) State Bugs(public question)

- The bugs in states are called state bugs. The state bugs arise due to the following reasons.

1. Number of States:

- A State graph consists of the number of states. It represents behavior of the system.
- In practice the state is directly or indirectly recorded.
- State table is used to record the number of states of the state graph.
- In state table the state bugs are occurred because of missing states.
- That is in state table if the number of states are not recorded or missed then the result might be the bugs.
- To find the missing states, first find the number of states
- The number of states is founded by as follows.
  1. Identify all the component factors of the state.
  2. Identify all the allowable values for each factor.
  3. Now the number of states is the product of the factors and allowable values.
- Functional specifications are used to find the factors of the state. They may also helpful to find the number of possible values for each factor.

2. Impossible States:

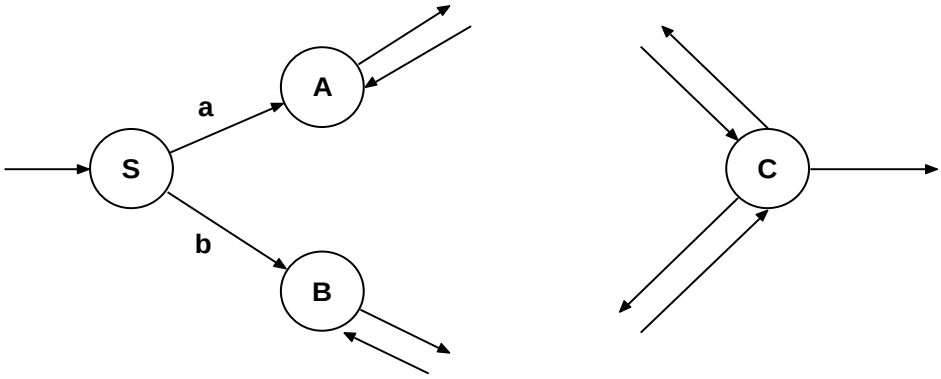
- A state that is not possible is called impossible states.
- For example a broken engine cannot run, so running a broken engine state is impossible state.
- There are some combination of factors that are impossible, they are
  GEAR: R, N, 1, 2, 3, 4 = 6 factors
  DIRECTION: forward, reverse, stopped = 3 factors
  ENGINE: running, stopped = 2 factors
  TRANMISSION: ok, broken = 2 factors
  ENGINE: ok, broken = 2 factors

TOTAL = 6 x 3 x 2 x 2 x2 =144 states.

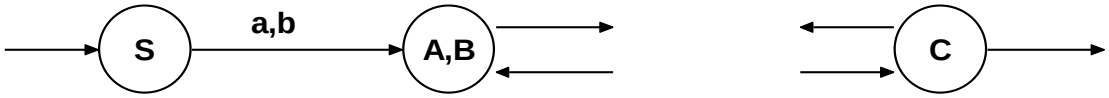
- A broken engine cannot run so the combination of engine is 3 states. Therefore the total number of states is 108. A car with a broken transmission does not move for long, there by further decreasing the number of states.

3. Equivalent States:

- Two states A, B are equivalent if every sequence of inputs starting from one state (s) produces exactly the same sequence of outputs.
- Let us take an example of two equivalent states.
- In the below figure, let us assume the system is in state S.
- An input of 'a' begins a transition to state A and an input of 'b' begins a transition to state B from S.
- If all the sequence of inputs from the state A generates exactly the same sequence of outputs as the other state B, then we say that these two states are equivalent.

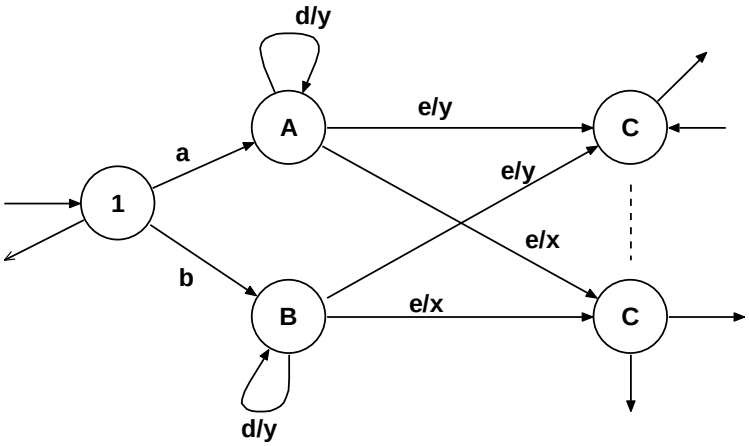


- Because these two states are treated equally, the state graph can be minimized by combining these two equivalent states as shown in the following figure.



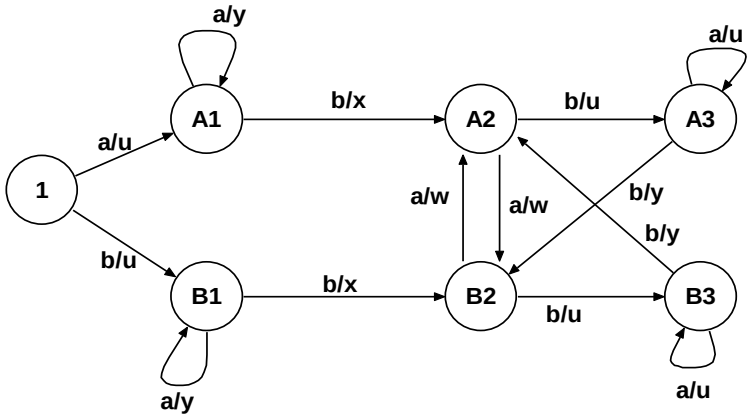
- Equivalent states can be recognized by the following procedure.

1. The two states are differentiated only by the different input values. For example Consider the following figure.



Here except a, b inputs, the system behavior in two states A, B are identical for every input sequence.

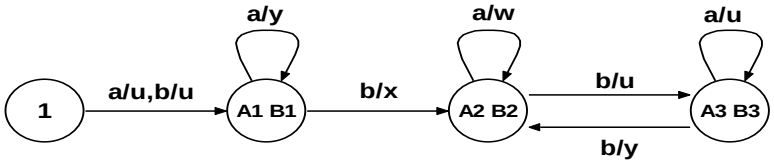
2. There are two set of rows which except for the state name, have identical state graphs with respect to transitions and outputs. The two sets can be merged. Let consider the following equivalent states.



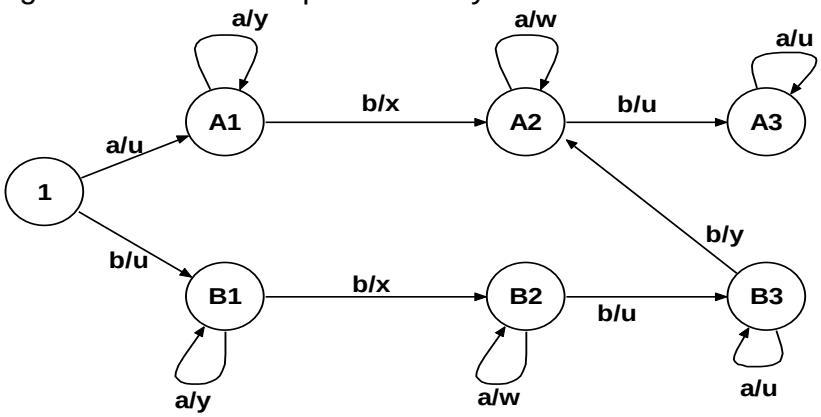
The Decision table to the above figure is shown below:

STATE	OK	ERROR
1	A1/u	B1/u
A1	A1/y	A2/x
B1	B1/y	B2/x
A2	B2/w	A3/u
B2	A2/w	B3/u
A3	A3/u	B2/y
B3	B3/u	A2/y

The merged equivalent states are represented by as follows



The Unmergeable states are represented by as follows



(3) Transition Bugs(public question)

- The connectivity between two or more states is known as transition.
- The bug in transition is called Transition Bug.

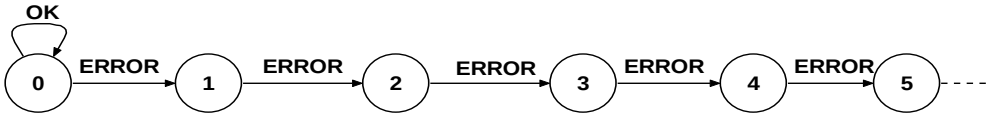
1. Unspecified and Contradictory Transitions:

- A transition is specified between states. If a transition may occur between states and not specified (i.e. unspecified transition) then the transition bug occurs.
- If a transition is not possible in the state then there must be a method that prevents the occurrence of input in that state.

- If there is no such method available then the occurrence of input becomes inefficient.
- So to avoid transition bug one transition must be specified for every input state combination.
- A program does not contain contradictions, if one input must be processed at a time to produce desired output. If a transition does not possible between states and the transition is specified then a contradictory transition may occur.
- That is if a programmer does not take all the measures of a program then contradictory transitions may occur because of transitions may not be possible between some of the states. For example if a single bit of a state is misplaced by the programmer then it doubles the number of states in the state graph and performs the contradictory transitions. This contradiction gives a transition bug.

2. Example(public question)

- The following example shows how to convert a specification into a state graph and how contradictions can come out.(public question)



Rule 1:

- The program will maintain an error counter which will be incremented whenever there is an error. Here there are only two input values OK, ERROR.
- These values make it easier to detect ambiguities and contradictions in a state table.

INPUT

STATE	OK	ERROR
0	0/NONE	1/
1		2/
2		3/
3		4/
4		5/
5		6/
6		7/
7		8/

Rule 2: If there is an error rewrite the block.

INPUT

STATE	OK	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE
4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there are three errors, erase 10 centimeters of tape and rewrite the block.

INPUT

STATE	OK	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE,ERASE,REWRITE

3		4/REWRITE,ERASE,REWRITE
4		5/REWRITE,ERASE,REWRITE
5		6/REWRITE,ERASE,REWRITE
6		7/REWRITE,ERASE,REWRITE
7		8/REWRITE,ERASE,REWRITE

Rule 4: If there are three erasures and another error occur, then put out of service.

INPUT

STATE	OK	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/ERASE,REWRITE
3		4/ERASE,REWRITE
4		5/ERASE,REWRITE
5		6/OUT
6		
7		

Rule 5:

- If the erasure was successful return to the normal state and clear the error counter.

INPUT

STATE	OK	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/ERASE,REWRITE
3	0/NONE	4/ERASE,REWRITE
4	0/NONE	5/ERASE,REWRITE
5	0/NONE	6/OUT
6		

Rule 6:

- If the rewrite was unsuccessful increment the error counter, and try another rewrite.

Rule 7:

- If the rewrite was successful decrement the error counter and return to the previous state.

INPUT

STATE	OK	ERROR
0	0/NONE	1/REWRITE
1	0/NONE	2/REWRITE
2	1/NONE	3/ERASE,REWRITE
3	0/NONE 2/NONE	4/ERASE,REWRITE
4	0/NONE 3/NONE	5/ERASE,REWRITE
5	0/NONE 4/NONE	6/OUT
6		

Rule 7 A:

- If there have been no erasures and the rewrite is successful return to the previous state.

3. Unreachable States:

- ❖ An un reachable state is like unreachable code If a transition is not specified between two states then that states are unreachable. That is if any incorrect transition occurs then the state becomes unreachable.
- ❖ There may be a transition from unreachable states to other states.

4. Dead States:

- ❖ A dead state is a state that once entered cannot be left.
- ❖ In programming, a set of states may be dead because a program has two stages.
- ❖ In the first stage an initialization process takes place that consists of number of states to be initialized.
- ❖ In the second stage strongly connected set of functional states takes place in which operations of the states cannot be completed. So the functional states become dead states. The only solution to this problem is system restart.

(4) Output Errors:

- ➢ The errors in output are called output errors.
- ➢ The states, the transitions, and the inputs may be correct & there may be no dead or unreachable states, but the output for the transition may be incorrect.
- ➢ Output actions must be verified independently for states and transitions.

(5) Encoding Bugs:(public question)

- ➢ Encoding is a process of converting or coding the inputs, transitions, and outputs of the state.
- ➢ Encoding process is applied in both explicit and implicit finite state machines.
- ➢ The encoding bugs are more common at the time of input coding, output coding and state coding in an explicit state machine.
- ➢ The encoding bugs may also exist in an implicit finite state machine, because of different views made by programmer and tester.
- ➢ The behavior of a finite state machine is invariant under all encodings.
- ➢ That is say that the states are numbered 1 to n.
- ➢ If you renumber the states by an arbitrary permutation, the finite state machine is unchanged. Similarly for input and output code is unchanged.
- ➢ Therefore if you present your version of the finite state machine with a different encoding and if the programmer objects to renaming then there is encoding bugs.
- ➢ You may have to look at the implementation for these, especially the data dictionary.
- ➢ The implementation of the fields as bunch of bits tells you the potential size of the code.
- ➢ If the number of code value is less than this potential, there is an encoding process.
- ➢ In strongly typed languages with user defined semantic types the encoding process is probably a type conversion a set membership to integer.
- ➢ Again you may have to look at the program to spot potential bugs of this kind.

(3) State Testing:

(i) Impact of Bugs:

- ➢ Let us say that a routine is specified as a state graph that has been verified as correct in all details.
- ➢ From the following the bugs may occur.
  1. Wrong number of states
  2. Wrong transition
  3. Wrong output for a given transition
  4. Pair of states are wrongly made equivalent
  5. Set of states are split to create in equivalent duplicates.
  6. Set of states become dead.

7. Set of states become unreachable.

(ii) Principles:(public question)

- State testing is defined as a functional testing technique to test the functional bugs in the entire system.
- The principles for state testing are very similar to the principles of path testing.
- For path testing it is not possible to test every possible path in a flowgraph.
- Similarly for state testing it is not possible to test every possible path in a state graph.
- In a state graph a path is a sequence of transitions caused by a sequence of inputs.
- In state testing the primary interest is given to the states and transitions rather than outputs.
- In state testing define a set of covering input sequences and for each step in each input sequence define the expected next state, the expected transition and the expected output code.
- A set of tests consists of three sets of sequences
  1. Input sequences.
  2. Corresponding transitions
  3. Output sequences.

(iii) Limitations and extensions:

The limitation is: State transition coverage in a state graph does not guarantee complete testing.

The extension:

- Chow defines a hierarchy of paths and methods for combining paths.
- The simplest is called a 0 switch which corresponds to test each transition independently.
- The next level consists of testing transition sequences consisting of two transitions called 1 switch. The maximum length switch is an n-1 switch where n is the number of states.

The different advantages are

- State testing is useful when the error corrections are less expensive.
- State testing is also useful when the testers want to detect a specified input.
- A state testing is specifically designed for catching the deep bugs.
- A state testing provides easiness during the design of tests.

The different disadvantages are

- State testing does not provide through testing because when a test is completed there might be some bugs remains in the system. Testers require large number of input sequences to catch transition errors, missing states etc..

(iv) What to model:

- Combination of hardware & software can be modeled sufficiently complicated state graph.
- The state graph is behavioral model that is it is functional rather than structural.

(v) Getting the data:

- Here labor intensive data gathering is needed and needs more meetings to resolves issues.

(vi) Tools:

- Tools for hardware logic designs are needed.

(4) Testability tips:

(i) A balm for programmers:

- The key to testability design is easy that is we can easily build explicit finite state machines.

(ii) How big How small:

- For two finite state machines there are only eight good and bad ones.
- For three finite state machines there are eighty possible good and bad one.
- Similarly for Four state machines 2700 most of which are bad and for five state machines 275000 most of which are bad. For six state machines 100 millions most of which are bad.

(iii) Switches, Flags and unachievable paths :

- The functionality of switches and flags are almost similar in the state testing.

- Switches or flags are used as essential tool for state testing to test the finite state machine in every possible state.
- A flag or switch value is set at the initial process of finite state machine, and then this value is evaluated and tested.
- Depending on its value the specific path is selected to find an easiest way for testing the finite state machine.
- Mostly the switch or flag works on true or false condition.
- In figure a flag is set to p in the program. This p variable is assigned to some value which can be evaluated.
- Depending on its value a path is separated into branches in order to proceed testing in either way that is u or x
- This also can be done by removing a flag and separating v path into two different paths w,y as shown in the above figure.
- Unachievable paths those paths which don't interact with each other.
- Here there are four paths u,w,x,y in that two are not achievable and two are achievable.
- That is u is not achievable to path y and path x is not achievable to path w & u is achievable to path w and path x is achievable to path y.
- Finally both the paths uw and xy are needed to cover the branches.
- In the above figure there are three flag variables p,q,r in the program.
- These variables are assigned some values that can be evaluated and based on which the paths are separated into branches.
- The main benefit of using this implementation is to remove the unnecessary combination from the decision tree as shown in the figure c.

(iv) Essential and inessential finite state behavior:

- To understand an essential and inessential finite state behavior, we need to know the concept of finite state machines and combinational machines.
- There is a difference between finite state machines and combinational machines in terms of quality.
- In combinational machines a path is chosen depending on the truth values of predicates.
- The predicate truth values are the values which once determined will never change and always remains constant.
- In these machines a path is equivalent to a Boolean algebraic expression over the predicates
- Further more it does not matter in which order the decisions are made.

(v) Design guide lines:

- Fine state machine is represented by a state graph having a finite number of states and a finite number of transitions between states
- Finite state machine (FSM) is a functional testing tool and programming testing tool.
- That is it is an essential tool for state testing in order to identify or model the behavior of software.
- The different guide lines are given below.
  1. Initially learn the procedure of finite state machine that are used in both hardware and software.
  2. Design an abstract machine in such a way that it works properly and satisfies the user requirements.
  3. Design an explicit finite state machine.
  4. Prototype test must be conducted thoroughly to determine the processing time and space of explicit finite state machine design.
  5. If time or space is more effecting the overall system, then use shortcuts to complete the design process.

6. If there are more than a few numbers of states then use hierarchical design to represent them.
7. If there is large number of states then software tools and programming languages must be developed.
8. The capability to initialize to an arbitrary state must be inbuilt together with the transition verification instrumentation.

GRAPH MATRICES AND APPLICATIONS

(1) Motivational Overview:

(1) What are the problems with pictorial graphs?

Problems with pictorial graphs:

1. Tracing a path in a pictorial graph is difficult task.
2. There is every possibility of having an error while tracing i.e. we can miss a link or cover some links twice.
3. Even yellow marking pen also not be reliable because once the concentration is lost during marking; we will lose the position to be marked.
4. It is very difficult to generate test cases for a pictorial graph
5. The time is also wasted if pictorial graphs are used.

(2) What are the graph matrices and their applications?

(i) Graph Matrices:

- The matrix in which every node of a graph is represented by one row and one column is called a graph matrix. or The matrix that represents the structure of a graph is known as graph matrix.
- In a graph matrix each row and each column intersection represents, the relationship between the respective row nodes and column nodes.
- A graph is an abstract representation of a software structure.
- A graph can be traced thoroughly to perform a check for covering paths, sensitizing paths, predicate expressions etc.
- Here we use either pictorial graphs or graph matrices.
- Tracing a path in a pictorial graph is difficult task.
- There is every possibility of having an error while tracing i.e. we can miss a link or cover some links twice. Even yellow marking pen also not be reliable because once the concentration is lost during marking; we will lose the position to be marked.
- Graph matrices are introduced to overcome these problems.
- A graph matrix is purely based on matrix methods.

(ii) Applications:

(i) Tool Building:

- Using matrix representation and its methods we construct test tools.
- It is more difficult to generate test cases for a pictorial graph than the graph matrix.

(ii) Doing and understanding testing theory:

- Theoretically speaking, graphs are the simple structures but when used in theorem proving we use graph matrices because pictorial graphs will omit some important algorithms.

(iii) The Basic Algorithms:

- The basic algorithms represent a basic tool kit. The basic tool kit consists of
  1. Matrix multiplication is used to derive the path expression from every node to every other node.
  2. A partitioning algorithm is used for eliminating loops from graphs.
  3. A collapsing process is used to get the path expression.

(3) Write relative merits and demerits of different Graph Matrix representations?

(i) Merits:

1. Using matrix representation and its methods we construct test tools.
2. Matrix representation gives the best results.
3. Graph matrices are used for developing algorithms and proving theorems of graphs.
4. Linked list representation is used to represent graph matrices.

(ii) Demerits:

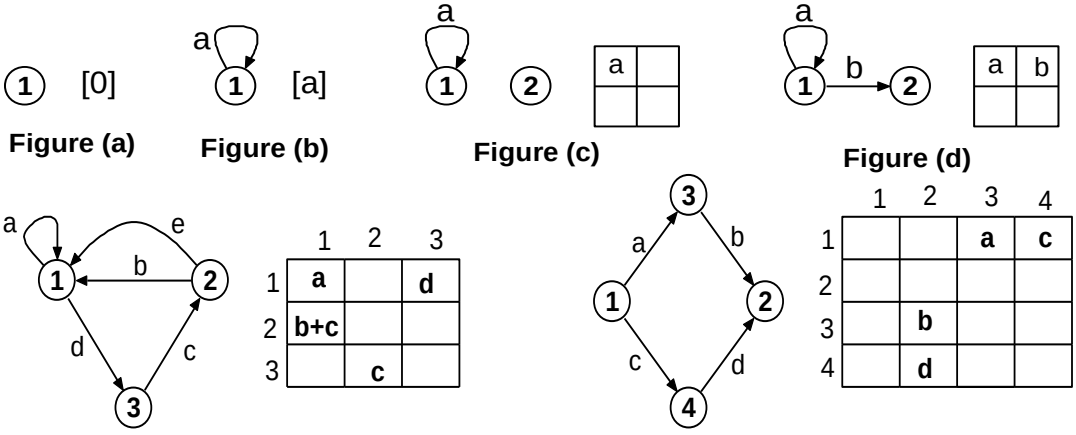
1. Graph matrix representation for two dimensional arrays is useful only for small graphs with simple link weights, however with large graphs; this matrix representation gives inconvenience.
2. Matrix representation requires a large storage space.
3. An additional weight matrix is also needed.
4. Since many entries of the graph matrices are null, the time taken to process such entries is a waste of time.

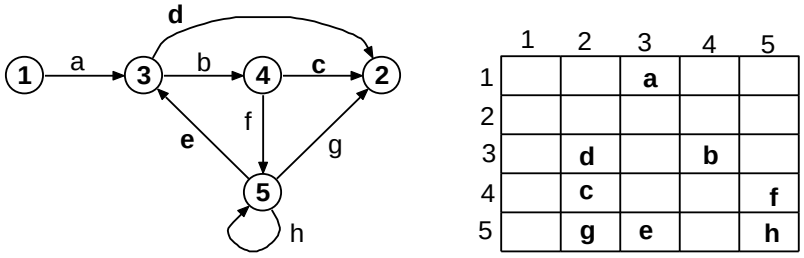
(2) The Matrix of a graph:

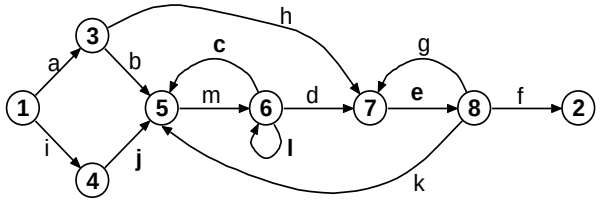
(1) Explain about the matrix of a graph?

(i) Basic Principles:

- A graph matrix is array representation of nodes. In a graph matrix each row and each column intersection represents, the relationship between the respective row nodes and column nodes.
- Some examples of graphs and their associated matrices are given by.







	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Figure (h)

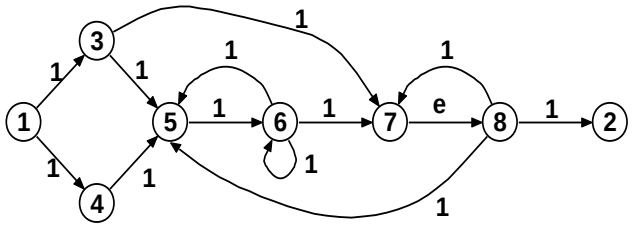
- Now observe the following
  1. The size of the matrix is equal to the number of nodes.
  2. There is a place to put every link weight between any node and any other node. i.e. The entry at a row and column intersection is the link weight of the link.
  3. A connection from node i to node j does not same that a connection from node j to node i. For example in figure (h) the (5,6) entry is m but the (6,5) entry is c..

(ii) A simple weight:

- Let '1' means that there is a connection and '0' means that there is no connection.
- The different arithmetic rules are

  1+1=1 1+0=1 0+0=0

  1x1=1 1x0=0 0x0=0
- A matrix with link weights defined with 1 or 0 is called a connection matrix.
- Consider the following flowgraph and its matrix representation.



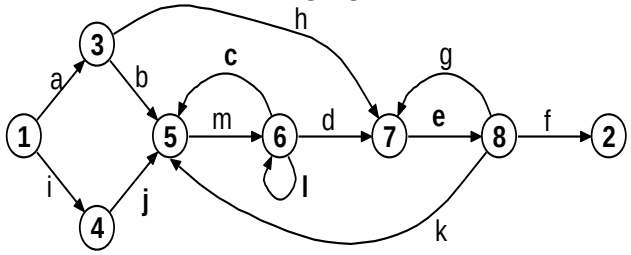
	1	2	3	4	5	6	7	8	
1			1	1					2-1=1
2									
3					1		1		2-1=1
4					1				1-1=0
5						1			1-1=0
6					1	1	1		3-1=2
7								1	1-1=0
8		1			1		1		3-1=2
									6+1=7

- Each row of a matrix denotes the outlinks corresponding to that node and each column denotes the inlinks corresponding to that node.
- A branch node is a node with more than one non zero entry in its row. For example rows 1,3,6, and 8 of the above figure have more than one entry, so these nodes are branch nodes.
- A junction node is a node with more than one non zero entry in its column. For example columns 5,6 and 7 of the above figure have more than one entry, so these nodes are junction nodes
- By subtracting 1 from the total number of entries in each row and ignoring rows with no entries we obtain the number of decisions for each row. Adding these decision values and then adding 1 to the sum gives the graph's cyclomatic complexity.
- In the above figure the graph's cyclomatic complexity is 7.

(iii) Further notation:

- The link weight between node i and node j, is denoted by a_{ij}.

- A self loop at node i is denoted by a_{ii} The link weight for the link between nodes j and i is denoted by a_{ji}.
- Consider the following figure.



- From the above figure
  abmd=$a_{13}\, a_{35}\, a_{56}\, a_{67}$
  degef=$a_{67}\, a_{78}\, a_{87}\, a_{78}\, a_{82}$
  ahekmlld=$a_{13}\, a_{37}\, a_{78}\, a_{85}\, a_{56}\, a_{66}\, a_{66}\, a_{67}$
- The expression $a_{ij}\, a_{jj}\, a_{jm}$ denotes that a path from node i to j, with a self loop at j and then a link from node j to m.
- The transpose of a matrix is the matrix with rows and columns interchanged.
- It is denoted by A^T.
- If $C=A^T$ then $c_{ij}=a_{ji}$
- The intersection of two matrices is denoted by A#B. If C=A # B then $c_{ij}=a_{ij}$ # b_{ij}.

(3) Node Reduction Algorithm:

Write the steps involved in Node Reduction Algorithm. Illustrate with an example?

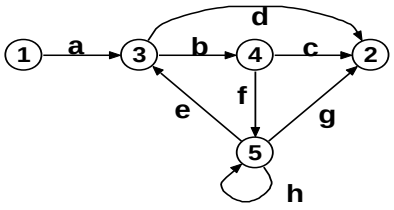
Node Reduction Algorithm:

Steps:

1. The reduction is done one node at a time by combining the elements in the last column with the elements in the last row and putting the result into the entry at the corresponding intersection. This step is called cross-term reduction. After cross term reduction the matrix size is reduced by one.
2. If there is one entry in one position and we want to enter another entry in that same position then add that two entries. This step is called parallel reduction.
3. If there is entry in principle diagonal then it represents a self loop. To remove that self loop, multiply every term in that row by the loop term. This step is called loop reduction.
4. By using the above three steps a 2x2 size matrix is obtained with the path expression. This path expression is the required path expression from node 1 to node 2.

Example:

- Consider the following flow graph.



- Specify the above flowgraph in the matrix format.

	1	2	3	4	5
1	.		a		
2		.			
3		d	.	b	
4		c		.	f
5		g	e		h

- Remove the self loop at node 5 by applying the loop reduction step.

	1	2	3	4	5
1	·		a		
2		·			
3		d	·	b	
4		c		·	f
5		h*g	h*e		·

- Combine the elements in the last column with the elements in the last row by applying cross-term reduction and parallel reduction steps.

	1	2	3	4
1	·		a	
2		·		
3		d	·	b
4		c+fh*g	fh*e	·

- Combine the elements in the last column with the elements in the last row by applying cross-term reduction and parallel reduction steps.

	1	2	3
1	·		a
2		·	
3		d+b(c+fh*g)	bfh*e

- Again a self loop occurred at node 3. So Remove the self loop by applying loop reduction step.

	1	2	3
1	·		a
2		·	
3		(bfh*e)*(d+b(c+fh*g))	

- Combine the elements in the last column with the elements in the last row by applying cross-term reduction.

	1	2
1	·	a(bfh*e)*(d+b(c+fh*g))
2		·

Note: Refer other four examples from class notes

(4) Applications:

(1) Illustrate the applications of Node Reduction Algorithm:

(i) Maximum Path Count Arithmetic:

- For theory refer unit-5 material.
- For example refer unit-8 notes.

(ii) Probability of path expressions:

- For theory refer unit-5 material.
- For example refer unit-8 notes.

(5) Relations:

(1) What is a Relation? What are the different properties of Relations?

Relation:

- The property by which two nodes are interconnected is called a relation.
- A relation can be represented by a link with connecting nodes.
- A link represented with link weight.
- This link weight can be numerical, logical, illogical, or whatever.
- The graph matrix which consists of unweighted simple links is called a connection matrix
- The graph matrix which consists of weighted simple links is called a relation matrix.

Different properties of relations:

- The different properties of relations are.

(i) Transitive Relations:

- A Relation R is transitive if *a*R*b* and *b*R*c* then *a*R*c*.
- Examples of transitive relations are: is connected to, is greater than or equal to, is less than or equal to, is a relative of, etc.
- Examples of intransitive relations are: is a friend of, is a neighbor of, etc.

(ii) Reflexive Relations:

- A relation R is reflexive if for every *a*, *a*R*a*. This relation represents a self-loop at every node.
- Examples of reflexive relations are: equals, is a relative of, etc.
- Examples of irreflexive relations include: not equals, is a friend of, etc.

(iii) Symmetric Relations:

- A relation R is symmetric if *a*R*b* then *b*R*a*. This relation represents if a link from a to b then there is also a link from b to a.
- A graph whose relations are symmetric is called an undirected graph and a graph whose relations are not symmetric is called a directed graph
- Examples of symmetric relations are: a relative of, is brother of, etc.
- Examples of asymmetric relations are: is the boss of, is greater than, etc.

(iv) Antisymmetric Relations:

- A relation R is antisymmetric, if *a*R*b* and *b*R*a*, then *a* = *b.*
- Examples of antisymmetric relations are: is greater than or equal to, is a subset of, etc.
- Examples of nonantisymmetric relations are: is connected to, is a relative of, etc.

(2) What are Equivalence Relations and Partial Ordering Relations?

(i)Equivalence Relations:

- A relation is said to be an equivalence relation if it satisfies transitive, reflexive, and symmetric properties. If a set of objects satisfy an equivalence relation, then it forms an equivalence class.
- The idea behind a partition-testing is that we can partition the input space into equivalence classes.

(ii)Partial Ordering Relations:

- A partial ordering relation satisfies the reflexive, transitive, and antisymmetric properties.
- A graph which shows partial ordering relation between its nodes is said to be partial ordered graph. Partial ordered graphs have different properties. They are
  i. loop free,
  ii. There is at least one maximum element.
  iii. There is at least one minimum element.

iv. If you reverse all the arrows the resultant graph is also partial ordered.

- The maximum element '*a'* represents the relation *x*R*a.* Similarly, the minimum element '*a'*, represents a relation *a*R*x*. Examples are Trees and loop-free graphs.

(6) The Powers of a Matrix:

(i) Explain about Matrix Powers and Products?

Matrix Powers and Products:

- A graph matrix is array representation of nodes. In a graph matrix each row and each column intersection represents, the relationship between the respective row nodes and column nodes.
- The square of the matrix represents all path segments with two links long. Similarly the third power represents all path segments with three links long and so on.
- Let A be a matrix whose entries are a_{ij}. The set of all paths between any node i and any node j is given by

$$a_{ij} + \sum_{k=1}^{n} a_{ik} a_{kj} + \sum_{k=1}^{n} \sum_{m=1}^{n} a_{ik} a_{km} a_{mj} + \sum_{k=1}^{n} \sum_{m=1}^{n} \sum_{l=1}^{n} a_{ik} a_{km} a_{ml} a_{lj}$$

$$+ \cdots \sum_{k=1}^{n} \sum_{m=1}^{n} \sum_{l=1}^{n} \cdots \sum_{p=1}^{n} a_{ik} a_{km} a_{ml} \cdots a_{qp} a_{pj}$$

- Given a matrix whose entities are a_{ij} the square of that matrix is given by

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

- When given two matrices A,B with entries a_{ik} and b_{kj} respectively. The product of AB is a new matrix C whose entries are c_{ij}.

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

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} X \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42}$$

$$c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43}$$

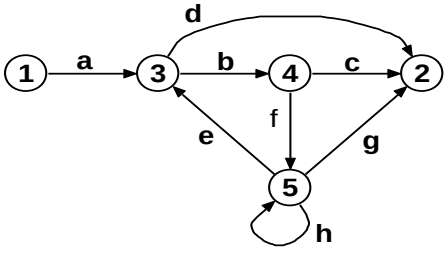
$$c_{32} = a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42}$$

$$c_{44} = a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}$$

- The c_{32} entry is obtained by combining, the entries in the third row of the A matrix, with the corresponding elements in the second column of the B matrix.

Example:

- Consider the following flowgraph and its graph matrix.



Let A =

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

$A^2 = A * A$

$A^2 =$

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

x

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

=

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed+hg	he	eb	h^2

$A^3 = A^2 * A$

$A^3 =$

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed+hg	he	eb	h^2

x

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

=

	1	2	3	4	5
1		abc			abf
2					
3		bfg	bfe		bfh
4		fed+fhg	fhe	feb	fh^2
5		hed+ebc+h^2g	h^2e	heb	ebf+h^3

(ii) Explain about the set of all paths and the algorithm for finding set of all paths?

(a) The set of all paths:

- The set of all paths is given by the following infinite series of matrix powers.

$$\sum_{i=1}^{\infty} A^i = A^1+A^2+A^3+\ldots+A^\infty$$

- Let I be an n x n diagonal matrix where n is the number of nodes, then the above expression becomes

$$\sum_{i=1}^{\infty} A^i = A(I+A^1+A^2+A^3+\ldots+A^\infty)$$

- We know that (A+A) = A

  So $(A+I)^2 = A^2 + 2AI + I^2 = A^2 + 2A + I^2 = A^2 + A+A + I^2 = A^2+A+I$. (Since A + A = A)

- Similarly

  $(A+I)^n = I+A^1+A^2+A^3+\ldots A^n$

- Now the original expression becomes

$$\sum_{i=1}^{\infty} A^i = A(I+A^1+A^2+A^3+\ldots+A^\infty) = A(A+I)^\infty$$

- If the paths of length n-1, where n is the number of nodes, the set of all such paths is

$$\sum_{i=1}^{n-1} A^i = A(A+I)^{n-2}$$

(b) The algorithm for finding set of all paths:

- The algorithm for finding set of all paths
  1. Express n-2 as a binary number.
  2. Calculate the successive squares of (A+I) matrix, that is $(A+I)^2$, $(A+I)^4$, $(A+I)^8$, $(A+I)^{16}$ and so on.
  3. Select only the binary powers of (A+I) matrix that correspond to a value 1 in the binary representation of (n-2).
  4. The set of all paths of length less than or equal to (n-1) is obtained from the original matrix as a result of step 3.

- For example the set of all paths for 16 nodes is given by

$$\sum_{i=1}^{15} A^i = A(A+I)^8(A+I)^4(A+I)^2$$

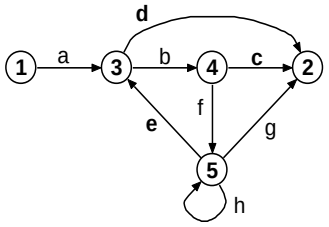
- A matrix for which $A^2=A$ is said to be idempotent matrix. A matrix whose successive power gives an idempotent matrix is called idempotent generator. The n^{th} power of a matrix A + I over a transitive relation is called the transitive closure of the matrix.

(iii) What are the loops? How to convert graphs with loops into loop-free graphs:

- Loops are infinite sum of matrix powers.
- The way to handle loops is similar like handling loops in regular expressions.
- Loop terms are displayed on the principle diagonal of the graph matrix. A loop can be removed from a graph by using loop reduction step of Node Reduction Algorithm.

Example:

- Consider the following flowgraph and its graph matrix



Let A =

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

A + I =

	1	2	3	4	5
1	1		a		
2		1			
3		d	1	b	
4		c		1	f
5		g	e		h+1

- In (A+I) matrix there is a self loop at node 5. Now we can obtain (A+I)* after removing the self loop at node 5 by applying loop reduction step.
- i.e. To remove self loop at node 5 multiply loop term h* with all row elements of row 5.

(A+I)* =

	1	2	3	4	5
1	1		a		
2		1			
3		d	1	b	
4		c		1	f
5		h*g	h*e		1

$(A+I)^{2*} = (A+I)^* \ (A+I)^*$

=

	1	2	3	4	5
1	1		a		
2		1			
3		d	1	b	
4		c		1	f
5		h*g	h*e		1

x

	1	2	3	4	5
1	1		a		
2		1			
3		d	1	b	
4		c		1	f
5		h*g	h*e		1

=

	1	2	3	4	5
1	1	ad	a	ab	
2		1			
3		d+bc	1	b	bf
4		c+fh*g	fh*e	1	f
5		h*g+h*ed	h*e	h*eb	1

$(A+I)^{3*} = (A+I)^{2*} \ (A+I)^*$

=

	1	2	3	4	5
1	1	ad	a	ab	
2		1			
3		d+bc	1	b	bf
4		c+fh*g	fh*e	1	f
5		h*g+h*ed	h*e	h*eb	1

x

	1	2	3	4	5
1	1		a		
2		1			
3		d	1	b	
4		c		1	f
5		h*g	h*e		1

=

	1	2	3	4	5
1	1	ad+abc	a	ab	abf
2		1			
3		d+bc+ bfh*g	1+ bfh*e	b	bf
4		c+fh*g+ fh*ed	fh*e	1+ fh*eb	f
5		h*g+h*ed +h*e(d+bc)	h*e	h*eb	1+ h*ebf

- No new loops were found for the second power matrix

- But the third power matrix has a loop term bfh*e at node 3. So all other entries in that row are multiplied by (bfh*e)*. Similarly there is a loop term (fh*eb) at node 4.
- So all other entries in that row are multiplied by (fh*eb)*. Also a loop term (h*ebf)* at node 5.
- So all other entries in the fifth row is multiplied by (h*ebf)*

(iv) Explain about Partitioning Algorithm in detail?

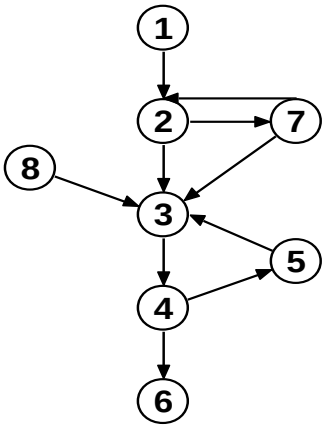
Partitioning Algorithm:

- It is an algorithm which is used to transform the graphs with loops into loop free graphs.
- There are certain points to remember here. They are
  1. A graph is loop free at the top level.
  2. Many graphs with loops are easy to analyze, if you know where to break the loops.
  3. This algorithm is used to develop a tool which can identify the loops.
- The partition algorithm represents.

  $(A+I)^n \# (A+I)^{nT}$

Example:

- Consider the following flowgraph and its graph matrix (A + I).



A+I =

	1	2	3	4	5	6	7	8
1	**1**	**1**						
2		**1**	**1**				**1**	
3			**1**	**1**				
4				**1**	**1**	**1**		
5			**1**		**1**			
6						**1**		
7		**1**	**1**				**1**	
8			**1**					**1**

- The transitive closure matrix $(A+I)^n$ can be obtained by using the following steps.

  Step:1: Mark all diagonal entries by 1

  Step:2: The flow from node 1 to node 6 is 1-2-7-2-3-4-5-3-4-6
  So mark nodes 1,2,3,4,5,6,7 by 1 in the first row

  Step:3: The flow from node 2 to node 6 is 2-7-2-3-4-5-3-4-6
  So mark nodes 2,3,4,5,6 by 1 in the second row.

  Step:4: The flow from node 3 to node 6 is 3-4-5-3-4-6.
  So mark nodes 3,4,5,6 by 1 in the third row

  Step:5: The flow from node 4 to node 6 is 4-5-3-4-6.
  So mark nodes 3,4,5,6 by 1 in the fourth row

. Step:6: The flow from node 5 to node 6 is 5-3-4-6.
  So mark nodes 3,4,5,6 by 1 in the fifth row.

  Step:7: The flow from node 6 is only 6. So mark node 6 by 1 in the sixth row.

  Step:8: The flow from node 7 to node 6 is 7-2-3-4-5-3-4-6
  So mark nodes 2,3,4,5,6,7 by 1 in the seventh row

. Step:9: The flow from node 8 to node 6 is 8-3-4-5-3-4-6
  So mark nodes 3,4,5,6,8 by 1 in the eighth row

- Therefore the transitive closure matrix is.

$(A+I)^n$ =

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	
2		1	1	1	1	1	1	
3			1	1	1	1		
4			1	1	1	1		
5			1	1	1	1		
6						1		
7		1	1	1	1	1	1	
8			1	1	1	1		1

- The transpose of $(A+I)^n$ is.

$(A+I)^{nT}$ =

	1	2	3	4	5	6	7	8
1	1							
2	1	1					1	
3	1	1	1	1	1		1	1
4	1	1	1	1	1		1	1
5	1	1	1	1	1		1	1
6	1	1	1	1	1	1	1	1
7	1	1					1	
8								1

- The intersection of transitive closure matrix $(A+I)^n$ and transpose matrix $(A+I)^{nT}$ is given by, identifying similar rows and column entries from $(A+I)^n$ and $(A+I)^{nT}$

$(A+I)^n \# (A+I)^{nT}$ =

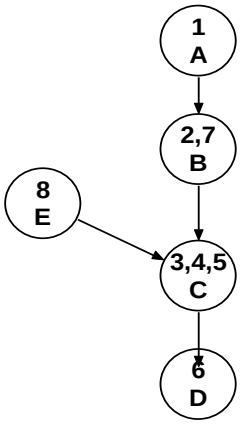
	1	2	3	4	5	6	7	8
1	1							
2		1					1	
3			1	1	1			
4			1	1	1			
5			1	1	1			
6						1		
7		1					1	
8								1

- From the above matrix, by comparing a row/column with other rows/columns, the equivalent nodes to be grouped.
- After grouping

Let A=[1] B=[2,7] C=[3,4,5] D=[6] E=[8]

- The graph and graph matrix representation to the above values is given by

FlowGraph



Graph Matrix

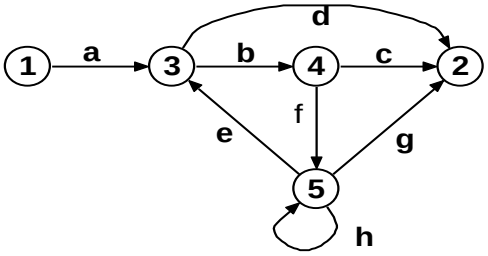
	A	B	C	D	E
A	1	1			
B		1	1		
C			1	1	
D				1	
E			1		1

(v) Explain about Breaking Loops And Applications:

- Consider the matrix format of a flowgraph.
- If there are any entries on the principal diagonal, then break the loop for that link.
- Here we use successive powers of the matrix.
- Another way to break the loop is applying the node reduction algorithm.
- Here we break the loop or we remove the self loop at any node is, by multiplying loop term with all other entries in that corresponding row.

(vi) Explain about Some matrix properties?

- To interchange the node names in the flowgraph, we must interchange both the corresponding rows and the corresponding columns of the node names in the graph matrix.
- Consider the following flowgraph and its Graph matrix.



	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

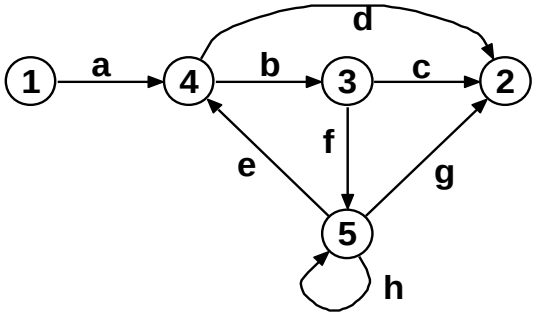
- Interchange rows 3 and 4 to the above graph matrix.

	1	2	3	4	5
1			a		
2					
3		c			f
4		d		b	
5		g	e		h

- Interchange columns 3 and 4 to the above graph matrix.

	1	2	3	4	5
1				a	
2					
3		c			f
4		d	b		
5		g		e	h

- The flowgraph to the above graph matrix is given by.



- By comparing the above flowgraph with the given flowgraph, it is proved that nodes 3,4 are interchanged

(7) Building Tools:

Explain about building tools of graph matrices?

i. Matrix Representation in software:

- A graph is an abstract representation of the software structure.
- Theoretically speaking, graphs are the simple structures but when used in theorem proving we have to apply graph matrices.
- It consists of

a) Overview:

- We prove theorems and develop graph algorithms by using graph matrices. When we want to process graphs in a computer we represent them as linked lists.

b) Node degree and graph density:

- Each intermediate node may have any number of inner links and outer links.
- The inner links of a node represents the in degree of the node.
- Similarly the outer links of a node represents the out degree of a node.
- The sum of in degree and out degree of a particular node is the degree of that node.
- The average degree of a node for a graph is between 3 and 4.
- The degree of a simple branch and simple junction is 3.
- The degree of a loop in a graph is 4.
- Any graph with average degree of 5 or 6 is said to be a very busy flowgraph.

c) What is wrong with arrays:

- We can represent the matrix as a two-dimensional array for small graphs with simple weights, but this is not convenient for larger graphs because

(i) Space:

- Matrix representation requires a large storage space.
- Hence a linked list representation is used which requires less storage space.

(ii) Weights:

- An additional weight matrix is required for complicated link weights.

(iii) Variable-Length Weights:

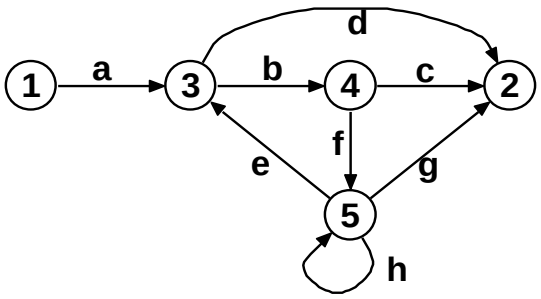
- The link weights in a flow graphs are represented in a two dimensional string array (matrix format), in which most of entries are null.

(iv) Processing time:

- Since many entries of the graph matrices are null, the time taken to process such entries are more.

d) Linked-list Representation:

- Consider following the flowgraph.



- The linked list for the above flowgraph is

1,3;a
2,exit
3,2;d
3,4;b
4,2;c
4,5;f
5,2;g
5,3;e
5,5;h

- List entries are usually placed in lexicographic (dictionary) order.
- The link weight expressions are stored in a string array to which link names act as pointers.
- If the link weights are values then, they are stored in an array of fixed length.

List Entry	Content
1	node 1,3;a
2	node 2,exit
3	node 3,2;d ,4;b
4	node 4,2;c ,5;f
5	node 5,2;g ,3;e ,5;h

- The node names appear only once, at the first link entry. A node name i.e starting node is used for the list entry.
- And there are back pointers for the inlinks. So we get

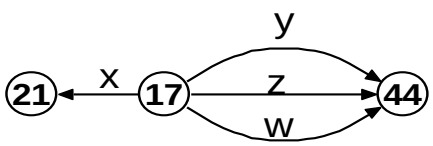
List Entry	Content	List Entry	Content
1	node 1,3;a	4	node 4,2;c ,5;f 3,
2	node 2,exit 3, 4, 5,	5	node 5,2;g ,3;e ,5;h 4, 5,
3	node 3,2;d ,4;b 1, 5,		

- It is important to keep the lists sorted in lexicographic order with the following priorities: node names or pointer outlink names, inlink names.

2. Matrix Operations:

a) Parallel Reduction:

- Parallel links after sorting are adjacent entries with the same pair of node names. Ex:



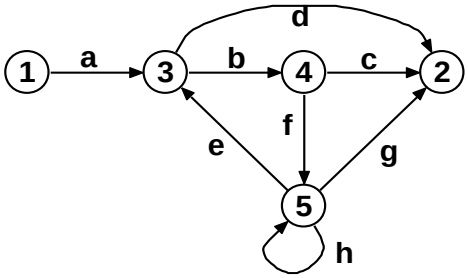
node 17,21;x
,44;y
,44;z
,44;w

- We have three parallel links from node 17 to 44. So

Node 17,21;x
,44;y(where y=y+z+w)

b)Loop Reduction:

- The loop reduction step is used to remove the loop. Here self link represents the loop.
- For removing a loop, the loop term is multiplied with all the outer links of the node at which the loop presents. Consider the following example.



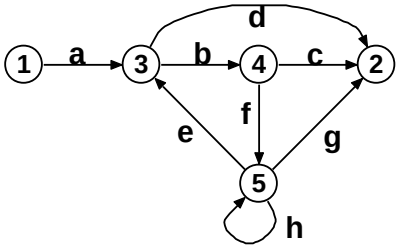
List Entry	Content Before	Content After
5	node 5,2;g ,3;e ,5;h 4, 5,	node 5,2;h*g ,3;h*e 4,

c) Cross term reduction:

- Select a node for reduction.
- The cross term reduction represents that we combine every inlink to the node with every outlink from that node after removing that node.
- The links created by node removal are stored in a separate list which is then sorted and thereafter merged into the master list.

d) Addition, Multiplication and other operations:

- Here addition of two matrices is simple. Multiplication is more complicated.
- Transposition is done by reversing the pointer directions, resulting in a list that is not correctly sorted. Sorting that list provides the transpose. All other matrix operations can be easily implemented by sorting, merging, and combining parallels.



List Entry	Content Before	Content After
2	node 2,exit 3, 4, 5,	node 2,exit 3, 4, 5,
3	node 3,2;d ,4;b 1, 5,	node 3,2;d ,2;bc ,5;bf 1, 5,
4	node 4,2;c ,5;f 3,	
5	node 5,2;h*g ,3;h*e 4,	node 5,2;h*g ,3;h*e

3. Node Reduction Optimization:

- The optimum order for node reduction is to do lowest-degree nodes first. The idea is to get the lists as short as possible as quickly as possible. Nodes of degree 3 reduce the total link count by one link when removed. A degree-4 node keeps the link count the same, and all higher-degree nodes increase the link count.
- For large graphs with 500 or more nodes and an average degree of 6 or 7, the difference between not optimizing the node-reduction order and optimizing it was about 50: 1 in processing time.

UNIT –IV
SOFTWARE TESTING TOOLS

(1) Introduction to Testing-Automated Testing:

- Automated Testing means using an automation tool executing the test case suite. Manual Testing is performed by a human sitting in front of a computer carefully executing the test steps.
- The automation software can also enter test data into the System. Now compare expected and actual results and generate detailed test reports. Test Automation demands considerable investments of money and resources.
- Successive development cycles will require execution of same test suite repeatedly. Using a test automation tool, it's possible to record this test suite and re-play it as required. Once the test suite is automated, no human test is required.
- This improve ROI (Return On Investment) of Test Automation. The goal of Automation is to reduce the number of test cases to be run manually and not to eliminate Manual Testing.

(2)Concepts of Test Automation:

(i)Test Automation

- Automated Software Testing or Test Automation is the process of automating the manual test cases. This also involves comparing the run time data with the test data provided, and producing useful Test Results.

(ii)Test Automation tool – What we can expect

- To test a software, a manual testing engineer needs to do the following basic actions:
  1. Click a button, link, image
  2. Select radio button
  3. Check / Uncheck a checkbox
  4. Type into an Edit box or controls which resemble edit box
- But also he needs to get the run time data from the application and compare with the test data provided (i.e. checkpoints). For this actually, we will be extracting the values from an edit box, button, link, image, table, drop down list, etc.
- All of these basic features are provided by Test Automation tools such as QTP(Quick Test Professional). There are many test automation tools which allow users to record user actions and replay them any number of iterations and compare the actual results with the test data and generate results.

(iii)Why Test Automation?

- Since the very large quantity of software development tools available in the market the productivity of the developer is dramatically increased over the past years.
- To manage the quality of the product developed it is essential to thoroughly test all the features of the application.
- If the application is a large software product having lot of features, then it is very much difficult to test by manually executing the test cases.
- There will be a lot of pressure on testing team to test more and more functionalities in less time. *Hence, the Automation Testing has become an integral part of software development*.

(iv)Benefits of Test Automation:

(a)Execution is Less Time Consuming:

- Automated tools execute the test cases (test script) faster than manual test case execution.
- Hence, in less time, more end-to-end test cases can be covered during test execution.

(b)Un-attended mode execution:

- There is no need of human involvement for the execution of entire Test Automation Suite for regression testing. But, to achieve the unattended mode execution, the automation framework has to be properly designed.

(c)Repeatable:

- The test suite can be executed multiple times on the application under test.
- For example, if there is a need to test the application on different browsers and environments, we need to just change the configurations of the test automation suite and execute.
- In case of manual execution, we would need one more resource to execute the same set of test cases on different environment / browser.

(d)Reusability:

- The test suite can be built in such a way that the functions or methods written are highly reusable across the framework. Also, the entire test suite built with a proper framework can also be utilized for different versions of the application under test.

(e)Consistency of Test Execution:

- There is a chance of manual tester making errors during execution of test cases. But, the test suite being automated we can expect no or zero errors during execution.
- For example, if there is a need to enter a value in an edit box such as 7693178.87651, a manual tester might make mistakes (as this is a big number with five decimal values) but the automation tool will not make any mistakes. It will enter the same value even if the test is run for many times.

(f)Better Coverage:

- As the time required executing automated test suite will be less compared to manual test case execution, more number of test scenarios can be covered during the execution.
- Hence, we can expect better coverage.

(g)Cost Effective:

- Once the test suite is completely ready for regression testing, the resources required will be less compared to manual test execution. This reduces the cost of testing.

(v)Key factors to be considered in Test Automation:

(a)Test automation is expensive

- It involves testing tools as well as skilled professionals.

***(b)Size of the Regression Test Suite*:**

- Generally, a recommendation is made for test automation if there is a long regression cycle.

(c)Tool compatibility:

- Test Automation tool has to be compatible with the application under test.
- A comprehensive tool evaluation has to be conducted and a best suited tool should be recommended for automation testing.

(d)No. of Regression Cycles:

- If there is a need to execute the automated test suite against many builds / releases then the test automation becomes cost effective and beneficial.

(3)Introduction to list of tools like Win runner, Load Runner & JMeter:

(i)Win Runner:

- Win Runner is the most used Automated Software Testing Tool.
- Main Features of Win Runner are
  - •Developed by Mercury Interactive
  - •Functionality testing tool
  - •Supports o/s and web technologies
  - •To Support .net, xml, SAP, Multimedia etc we can use QTP.
  - •Winrunner run on Windows only.
  - •Xrunner run only UNIX and linux.
  - •Tool developed in C on VC++ environment.

•To automate our manual test win runner used TSL (Test Script language like c)

The main Testing Process in Win Runner is

1) Learning

❖ Recognization of objects and windows in our application by winrunner is called learning.

2) Recording

❖ Winrunner records over manual business operation in TSL

3) Edit Script

❖ Depends on corresponding manual test, test engineer inserts check points in to that record script.

4) Run Script:

❖ During test script execution, winrunner compare tester given expected values and application actual values and returns results.

5) Analyze Results

❖ Tester analyzes the tool given results to concentrate on defect tracking if required.

(ii)Load Runner:

❖ LoadRunner is a software testing tool from Micro Focus.
❖ It is used to test applications, measuring system behaviour and performance under load.
❖ LoadRunner can simulate thousands of users concurrently using application software, recording and later analyzing the performance of key components of the application.
❖ The components of LoadRunner are The Virtual User Generator, Controller, and the Agent process, LoadRunner Analysis and Monitoring, LoadRunner Books Online.
❖ The Component of LoadRunner would you use to record a Script is the Virtual User Generator (VuGen) component .
❖ LoadRunner copy hundreds or thousands of human users with Virtual Users (Vusers) to apply measurable & repeatable production workloads and stresses an Application end-to-end.
❖ Vusers copy the actions of human users by performing typical Business Processes.
❖ For each user action, the tool submits an input request to server and receives the response.
❖ Increase number of Vusers, to increase the load on the Server.

(iii)Jmeter:

❖ The Apache JMeter is pure Java open source software, which was first developed by Stefano Mazzocchi of the Apache Software Foundation, designed to load test functional behavior and measure performance. You can use JMeter to analyze and measure the performance of web application or a variety of services.
❖ Apache JMeter is a testing tool used for analyzing and measuring the performance of different software services and products. It is a pure Java open source software used for testing Web Application or FTP application. It is used to execute performance testing, load testing and functional testing of web applications.
❖ First JMeter can be integrated into selenium using the plugins, So Selenium you can use as an automation tool for your testing needs. The Apache JMeter is pure Java open source software, designed to load test functional behavior and measure performance.
❖ JMeter is an open source desktop Java application that is designed to load test and measure performance. It can be used to simulate loads of various scenarios and output performance data in several ways, including CSV and XML files, and graphs.
❖ Apache JMeter is one of the most popular open source systems for testing applications, and also a 100% Java scripted desktop application.

(4)About Win Runner:

❖ WinRunner is a test automation tool, designed to help customers save testing time and effort by automating the manual testing process. Automated testing with WinRunner addresses the problems by manual testing, speeding up the testing process

- ❖ WinRunner uses the GUI Map file to recognize objects on the application. When WinRunner runs a test, it uses the GUI map to locate objects. It reads an object's description in the GUI map and then looks for an object with the same properties in the application being tested

(5)Using Win runner:

- ❖ After installing the WinRunner on your computer, invoke the WinRunner application:
  Start -> Programs ->WinRunner ->WinRunner
- ❖ The opening screen of the WinRunner application is displayed, prompting you to select one of the three options:
  New Test: To create a new test script
  Open Test: To open an existing test script
  Quick Preview: To view the quick preview of WinRunner

(6)Recording Test Cases

- ❖ To test any application, first you can run the application and understand its operation. Then, you can invoke WinRunner, again run the application and record the GUI operations.
- ❖ During the recording mode, WinRunner will capture all your actions, which button you pressed, where you clicked the mouse etc. You need to work with the application as usual and perform all the actions to be tested.
- ❖ Once the recording is completed, WinRunner generates a script in TSL (Test Script Language). You can run this test script generated by WinRunner to view the results. The test results will show whether the test has passed or failed.
- ❖ There are two modes of recording:

(i)Context Sensitive mode:

- ❖ This mode of recording is used when the location of the GUI controls (i.e. X and Y coordinates) or the mouse positions are not necessary.

(ii)Analog mode:

- This mode of recording is used when the mouse positions, the location of the controls in the application, also play an important role in testing the application. This mode of recording has to be used to validate bitmaps, testing the signature etc.
- The procedure for recording a test case is as follows:
  Step 1: Open a new document: File -> New (or) Select "New Test" from the WinRunner's Welcome screen.
  Step 2: Open (run) the application to be tested.
  Step 3: Start recording a test case.
  Create ->Record - Context Sensitive (or) click on the toolbar's "Record" button once, to record in Context Sensitive mode.
  Step 4: Select the application to be tested by clicking on the application's title bar.
  Step 5: Perform all the actions to be recorded.
  Step 6: Once all required actions are recorded, stop the recording.
  Create -> Stop (or) Click on the toolbar's "Stop" button to stop the recording WinRunner generates the script for the recoded actions.

Mapping the GUI,

- When WinRunner runs a test, it uses the GUI map to locate objects. It reads an object's description in the GUI map and then looks for an object with the same properties in the application being tested. Each of these objects in the GUI Map file will be having a logical name and a physical description.
- There are two views in the GUI Map Editor, which enable you to display the contents of either:
  (i)the entire GUI map
  (ii)an individual GUI map file

(i)the entire GUI map

- When viewing the contents of specific GUI map files, you can expand the GUI Map Editor to view two GUI map files simultaneously. This enables you to easily copy or move descriptions between files.
- To view the contents of individual GUI map files, choose ***View GUI Files***.

(ii)an individual GUI map file

- In the GUI Map Editor, objects are displayed in a tree under the icon of the window in which they appear. When you double-click a window name or icon in the tree, you can view all the objects it contains. To concurrently view all the objects in the tree, choose ***View Expand Objects Tree***. To viewwindows only, choose ***ViewCollapse Objects Tree***.
- When you view the entire GUI map, you can select the ***Show Physical Description*** check box to display the physical description of any object you select in the ***Windows/Objects*** list.
- When you view the contents of a single GUI map file, the GUI Map Editor automatically displays the physical description.
- Suppose the WordPad window is in your GUI map file. If you select ***Show Physical Description*** and click the WordPad window name or icon in the window list, the following physical description is displayed in the middle pane of the GUI Map Editor:

(7)Working with Test:

- WinRunner facilitates easy test creation by recording how you work on your application. As you point and click GUI (Graphical User Interface) objects in your application, WinRunner generates a test script in the C-like Test Script Language (TSL).
- You can further enhance your test scripts with manual programming.
- User can create tests using recording or programming or by both. While recording, each operation performed by the user generates a statement in the Test Script Language (TSL).

- These statements are displayed as a test script in a test window. User can then enhance the recorded test script, either by typing in additional TSL functions and programming elements or by using WinRunner's visual programming tool, the Function Generator, or using the Function Viewer.

(8)Checkpoints:

- Checkpoints allow user to compare the current behavior of the application being tested to its behavior in an earlier version. If any mismatches are found, WinRunner captures them as actual results.
- User can add four types of checkpoints to test scripts they are
  - (i)GUI Checkpoints
  - (ii)Bitmap Checkpoints
  - (iii)Text checkpoints
  - (iv)Database checkpoints

(i)GUI checkpoints

- It verifies information about GUI objects. For example, user can check that a button is enabled or see which item is selected in a list. There are three types of GUI Check points they are
  - (a)For Single Property
  - (b)For Object/Window
  - (c)For Multiple Objects

(a)GUI checkpoint for single property:-

- User can check a single property of a GUI object. For example, user can check whether a button is enabled or disabled or whether an item in a list is selected.

(b)GUI checkpoint for object/window:-

- User can create a GUI checkpoint to check a single object in the application being tested.
- User can either check the object with its default properties or user can specify multiple properties to check.

(c)GUI checkpoint for multiple objects:-

- User can create a GUI checkpoint to check multiple objects in the application being tested. User can either check the object with its default properties or user can specify multiple properties of multiple objects to check.

(ii)Bitmap Checkpoint :

- It checks an object, a window, or an area of a screen in the application as a bitmap. While creating a test, user can indicate what user want to check. WinRunner captures the specified bitmap, stores it in the expected results folder (exp) of the test, and inserts a checkpoint in the test script. While running the test, WinRunner compares the bitmap currently displayed in the application being tested with the expected bitmap stored earlier.
- In the event of a mismatch, WinRunner captures the current actual bitmap and generates a difference bitmap. By comparing the three bitmaps (expected, actual, and difference), user can identify the nature of the discrepancy there are two types of bitmap check points they are

(a)Bitmap Checkpoint for Object/Window: -

- User can capture a bitmap of any window or object in the application by pointing to it.

(b)Bitmap Checkpointfor Screen Area:-

- User defines any rectangular area of the screen and captures it as a bitmap for comparison.

(iii)Text checkpoints

- It reads and check text in GUI objects and in areas of the screen. While creating a test user has to point to an object or a window containing text. WinRunner reads the text and writes a TSL statement to the test script.
- Later user can add simple programming elements to test scripts to verify the contents of the text. User should use a text checkpoint on a GUI object only when a GUI checkpoint cannot be used

to check the text property. There are two types of Text checkpoints they are

From Object/Window

From Screen Area

(iv)Database checkpoints

❖ It checks the contents and the number of rows and columns of a result set, which is based on a query user, create on database. There are three types of database check points they are

(a)Default Check:-

❖ Used to check the entire contents of a result set, Default Check are useful when the expected results can be established before the test run.

(b)Custom Check:-

❖ Used to check the partial contents, the number of rows, and the number of columns of a result set.

(c)Runtime Record Check:-

❖ User can create runtime database record checkpoints in order to compare the values displayed in the application during the test run with the corresponding values in the database.

(9)Test Script Language & Enhancing Test

❖ TSL stands for "Test Scripting Language". The test scripts are written in Test Scripting Language in winrunner. TSL is an enhanced, C-like programming language designed for testing.

❖ The advantages of TSL are:

It is easy to use.

It is similar to other **programming languages**. ...

It is a high level **language**.

❖ When you record a test, a test script is generated in Mercury Interactive's Test Script Language (TSL). Each line WinRunner generates in the test script is a statement.

❖ A statement is any expression that is followed by a semicolon. Each TSL statement in the test script represents keyboard and/or mouse input to the application being tested. A single statement may be longer than one line in the test script.

❖ TSL is a C-like programming language designed for creating test scripts. It combines functions developed specifically for testing with general purpose programming language features such as variables, control-flow statements, arrays, and user-defined functions. You enhance a recorded test script simply by typing programming elements into the test window, If you program a test script by typing directly into the test window, remember to include a semicolon at the end of each statement.

❖ TSL is easy to use because you do not have to compile. You simply record or type in the test script and immediately execute the test.

❖ TSL includes four types of functions:

- ✓ Context Sensitive functions perform specific tasks on GUI objects, such as clicking a button or selecting an item from a list. Function names, such as button_press andlist_select_item, reflect the function's purpose.
- ✓ Analog functions depict mouse clicks, keyboard input, and the exact coordinates traveled by the mouse.
- ✓ Standard functions perform general purpose programming tasks, such as sending messages to a report or performing calculations.
- ✓ Customization functions allow you to adapt WinRunner to your testing environment.

❖ WinRunner includes a visual programming tool which helps you to quickly and easily add TSL functions to your tests. For more information, see "Generating Functions."

❖ This chapter introduces some basic programming concepts and shows you how to use a few simple programming techniques in order to create more powerful tests. For more information, refer to the following documentation:

(10)Running and Debugging Tests

(i)Create Tests

- Next, you create test scripts by recording, programming, or a combination of both. While recording tests, insert checkpoints where you want to check the response of the application being tested. You can insert checkpoints that check GUI objects, bitmaps, and databases.
- During this process, WinRunner captures data and saves it as expected results—the expected response of the application being tested.

(ii)Debug Tests

- You run tests in Debug mode to make sure they run smoothly. You can set breakpoints, monitor variables, and control how tests are run to identify and isolate defects.
- Test results are saved in the debug folder, which you can discard once you've finished debugging the test.
- When WinRunner runs a test, it checks each script line for basic syntax errors, like incorrect syntax or missing elements in *If*, *While*, *Switch*, and *For* statements.
- You can use the Syntax Check options (*Tools Syntax Check) to check for these types of syntax errors before running your test.*

(iii)Run Tests

- You run tests in Verify mode to test your application. Each time WinRunner encounters a checkpoint in the test script, it compares the current data of the application being tested to the expected data captured earlier.
- If any mismatches are found, WinRunner captures them as actual results.

(11)Analyzing Results:

- After you run a test, you can view the results in the Test Results window. The appearance of this window depends on the Report View option you select in the **Run**category of the General Options dialog box.
- There are two types of Test Results Views:

(i)WinRunner report view

- Displays the test results in a Windows-style viewer.
- If you run a test that includes a call to a QuickTest test, the WinRunner report view displays only basic information about the results of the QuickTest test.
- When running tests that call QuickTest tests, it is recommended to use the Unified report view.

(ii)Unified report view

- Displays the results in an HTML-style viewer.
- The unified TestFusion report viewer is identical to the style used for QuickTest Professional test results.
- If you run a test that includes a call to a QuickTest test (version 6.5 or later), the unified report view enables you to view detailed results of each step in the called QuickTest test.
- Regardless of the selected report view, the test results window always contains a description of the major events that occurred during the test run, such as GUI, bitmap, or database checkpoints, file comparisons, and error messages. It also includes tables and pictures to help you analyze the results.

(12)Batch Tests,

- A batch test is a test script that calls other tests.
- You program a batch test by typing call statements directly into the test window and selecting the Run in batch mode option in the Run category of the General Options dialog box before you execute the test.

- A batch test is a test script that calls other tests. You program a batch test by typing call statements directly into the test window and selecting the **Run in batch mode** option in the **Run** category of the General Options dialog box before you execute the test.
- A batch test may include programming elements such as loops and decision making statements. Loops enable a batch test to run called tests a specified number of times.
- Decision-making statements such as if/else and switch condition test execution on the results of a test called previously by the same batch script.
- See "Enhancing Your Test Scripts with Programming," for more information.
- You execute a batch test in the same way that you execute a regular test.
- Choose a mode (Verify, Update, or Debug) from the list on the toolbar and choose **Test > Run from Top**. See "Understanding Test Runs," for more information.
- When you run a batch test, WinRunner opens and executes each called test. All messages are suppressed so that the tests are run without interruption.
- If you run the batch test in **Verify** mode, the current test results are compared to the expected test results saved earlier.
- If you are running the batch test in order to update expected results, new expected results are created in the expected results folder for each test.
- See "Storing Batch Test Results" below for more information. When the batch test run is completed, you can view the test results in the Test Results window.
- Note that if your tests contain TSL **texit** statements, WinRunner interprets these statements differently for a batch test run than for a regular test run. During a regular test run, **texit** terminates test execution.
- During a batch test run, **texit** halts execution of the current test only and control is returned to the batch test.

(13)Rapid Test Script Wizard

- The Rapid test script wizard is the fastest way of performing the test process. It systematically opens up all the windows in the application, stores the learnt information in the GUI Map file and generates the test cases based on the information learnt from the application.
- It is possible to apply these tests only on those applications, which open windows upon performing some task (like clicking a button, selecting the menu item etc). Let us apply this wizard on the Calculator application whose GUI is shown in Figure.