As per R15, R19 CSE Syllabus

Specially Prepared for Engineering Students

Software Testing

By

H. Ateeq Ahmed, M.Tech., (Ph.D)

Assistant Professor of CSE,

Kurnool.

As per JNTUA R15 & R19 CSE Syllabus

NOT FOR SALE

ABOUT AUTHOR

H. ATEEQ AHMED is currently Pursuing **Ph.D** in CSE at JNTUA, Anantapur. Completed **M.Tech** from Samskruti College of Engineering & Technology, JNTUH, Hyderabad. **B.Tech** from Safa College of Engineering & Technology, JNTUA, Anantapur, Currently he is working as an Asst. Professor in the Department of CSE at Dr. KVSR Institute of Technology, Kurnool, A.P. He has more than **13 years** of teaching experience. He has vast subject knowledge in the field of Computer networks & organization, Network security, Software design & testing, Operating Systems, Programming in C, Java & Python as well as Data mining.

His **dynamic** and **illustrative** approach towards the subjects helps the students to enrich their skills in their academics which can help them to achieve success in job career.

He always believes in **real time approach** to solve various problems and expect the same with his students which can help them to understand their subject in an easy and efficient manner.

He has prepared many **materials** of different subjects for engineering students which can help them to prepare for exams in an **easy** method.

Finally, we expect our students to acquire as much knowledge as possible and convert his **dynamic teaching** into their **individual success**.

ACKNOWLEDGEMENT

First I thank Almighty God for giving me the knowledge to learn and teach various students.

It's my privilege to thanks my parents as without their right guidance and support, this book will be a dream for me.

Finally, I thank my colleagues and friends for helping me during tough period of time.

"The Goal of Software Testing is to achieve Quality in Software which results in Customer Satisfaction"

H. Ateeq Ahmed, M.Tech.,

Mobile no: 994 8 37 8 994,

E-mail ID: ateeqh25@gmail.com.

Website: *engineeringdrive.blogspot.in*

My YouTube Channel: *ENGINEERING DRIVE*

SYLLABUS & CONTENTS

UNIT-IV	
Paths, Path products and Regular expressions: path products path expression, reduction Procedure. applications, regular expressions & flow anomaly detection. **Logic Based Testing:** overview, decision tables. path expressions, kv charts, specifications. Examples.	72 – 99
UNIT-V	
State, State Graphs and Transition testing: Introduction state graphs. Good & Bad state graphs, state testing, importance Testability tips. **Graph Matrices and Application:** Motivational overview, matrix of graph, relations. power of a matrix, node reduction algorithm. Building Tools. (Student should be given an exposure to a tool like JMeter or Winrunner).	100 – 115

UNIT - I

Definition of Software Testing

Software testing is more than just error detection;

Testing software is operating the software under controlled conditions, to (1) *verify* that it behaves "as specified"; (2) to *detect errors*, and (3) to *validate* that what has been specified is what the user actually wanted.

In general, **Software Testing** is a combination of the following activities which are inter-related with one another.

- Error Detection
- Verification
- Validation

Let us now discuss each of the above activities briefly…

Error Detection:
It is a process in which the presence of errors is detected.
Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should.

Verification:
Verification is the checking or testing of items, including software, for confirmation and consistency by evaluating the results against pre-specified requirements.

Customer Requirements = Developed Software

Validation:
Validation looks at the system correctness – i.e. is the process of checking that what has been specified is what the user actually wanted.

Customer Requirements = Working Software

"The purpose of testing is verification, validation and error detection in order to find problems – and the purpose of finding those problems is to get them fixed."

THE PURPOSE OF TESTING

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 logical: 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.

Productivity and Quality in Software

The relation between productivity and quality for software is very different from that for manufactured goods. 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.

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.

There are two major Goals for Testing.

(1) Bug Prevention
(2) Bug Discovery.

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*.

Phases in a Tester's Mental Life

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.

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 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. 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.

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. The trouble with phase 2 thinking is that we don't know when to stop.

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.

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.

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.

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."

Testing Isn't Everything

Testing, I believe, is still our most potent weapon, but there's evidence 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.

The other major methods in decreasing order of effectiveness are as follows:

Inspection Methods—In this category I include walkthroughs, desk checking, formal inspections, 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.

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.

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.

SOME DICHOTOMIES

(1) 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 same; 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; Debugging starts from possibly unknown initial conditions, and the end cannot be predicted.
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. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.
6. Testing can often be done by an outsider. Debugging must be done by an insider.
7. Much of test execution and design can be automated. Automated debugging is still a dream.

(2) 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.

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.

(3) 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.

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.

(4) 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.

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.

(5) Small Versus Large

I often write small analytical programs of a few hundred lines that, once used, are discarded. 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.

You can extrapolate from there or draw on your experiences. **Programming in the large** 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.

(6) 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.

A MODEL FOR TESTING

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.

A model Project consists of the following entities.

Application—The specifics of the application are unimportant. It is a real-time system that must provide timely responses to user requests for services.

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.

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.

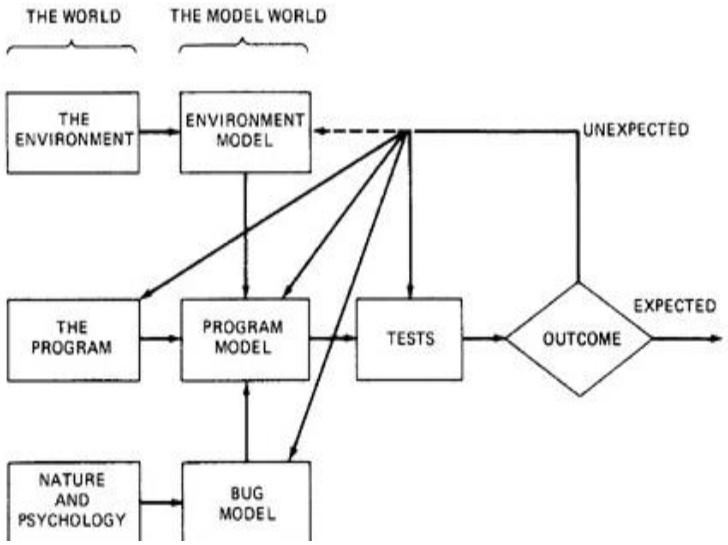
Standards—Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards. Documentation is good. There is an internal, semiformal, quality-assurance function. The database is centrally developed and administered.

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. 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.



A Model of Testing.

Overview

The above figure is a model of the testing process. 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.

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. 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.

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.

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.

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

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.

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 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 Testing:

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

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.

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.

THE CONSEQUENCES OF BUGS

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?

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)

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.

A TAXONOMY FOR BUGS

(1) 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.

(2) Requirements and Feature Bugs

Requirements Bugs

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.

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.

(3) 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

(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

(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.

(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.

(4) Data Bugs

Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values. Data bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all. 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.

(5) 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.

(6) Interface, Integration and System Bugs

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.

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.

(7) Test and Test Design Bugs

Testing

Testers have no immunity to bugs. Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently they can have bugs.

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.

Test Bug 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.*

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.

★★★★★★★

Basic concepts of path Testing :-

A path Testing is the name given to a family of Test techniques based on selecting a set of test paths through the program.

Control flowgraphs :

A control flowgraph is a graphical representation of a programs control structure.

* A control flowgraph is a form of flow chart which does not deal with the internal structure of the process rather it shows the data flow and control flow between the processes.

Every control flow graph consists of the following elements.

① process blocks

② Decision and case statements

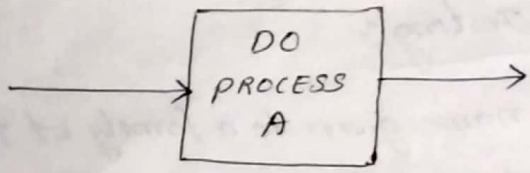
③ Junctions.

Let us discuss each of them briefly.

① process blocks :

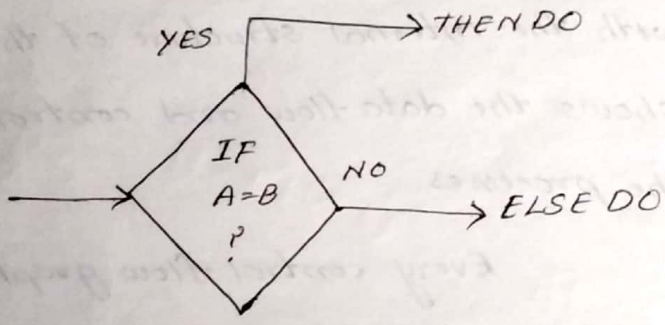
* A process block is a sequence of program statements uninterrupted by either decisions or junctions.
* Formally, it is a sequence of statements such that if any one statement is executed of that block, then all statements thereof are executed.
* A process has one entry and one exit.
* It can consist of one source statement or thousands.

It is represented as follows

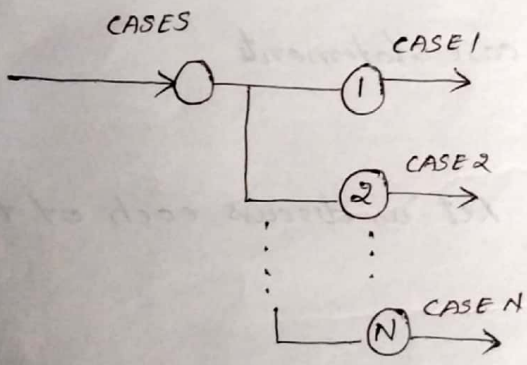


② Decisions and case statements:

* A Decision is a point in a program at which the control flow can diverge or split.
* The flow gets diverted in one of the many options available.

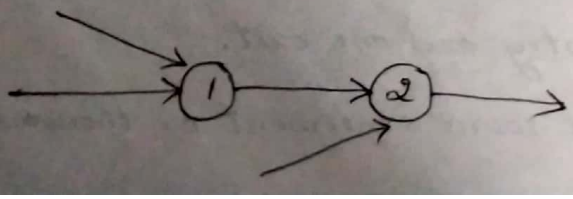


- A case statement is a multiway branch or decision.



③ Junctions:

* A junction is a point in the program where the control flow can merge.
* It is just an opposite to Decisions.



Differences between control flowgraphs and flow charts:

* In control flowgraphs, we don't show the details of what is in the process block whereas in flowchart provides the details.
* Flow chart shows the internal flows of each process whereas control flow graphs shows only the control and data flow between the processes.
* Flowcharts can be easily drawn manually, semi automatically or automatically using flow chart generators whereas no such good generator is available in the case of control flow graph.

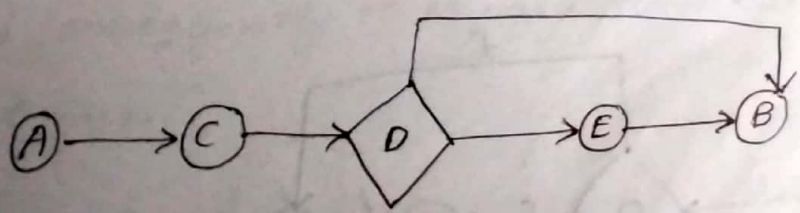
path Testing:

paths:

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."

eg:



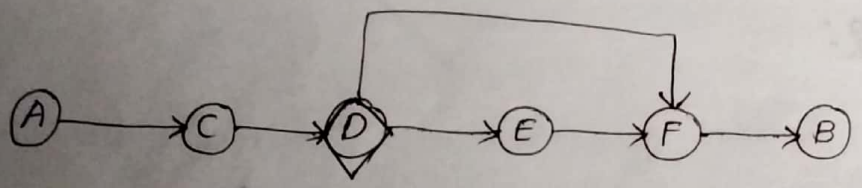
There are two different paths from on entry A to on exit B. They are ACDEB and ACDB respectively.

The simple and shortest path is "ACDB."

Nodes:

* Nodes are the graphical representation of real world entities.
* Nodes are mainly denoted by small circles.
* A node with more than one input link is known as junction and a node which has more than one output link is a Decision.
* Nodes can be labelled as alphabets or numbers.

eg:

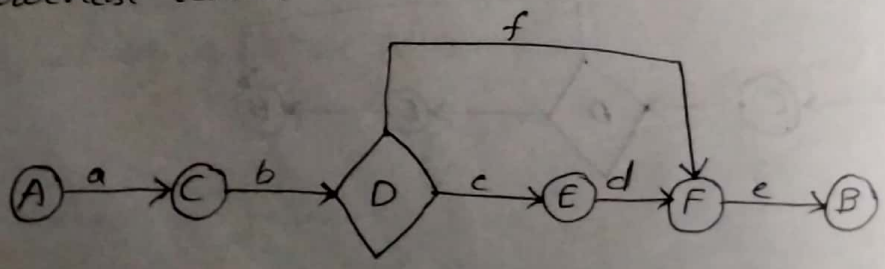


In the above fig, Nodes are (A,B,C,D,E,F) where D is the Decision which has two output links. F is the junction ~~whos~~ which has two input links.

Links:

* Every path consists of a set of processes known as links.
* A link is the mediator for any two nodes.
* It is denoted by '⟶' and can be represented by lowercase letter.

eg:



* In the above, a, b, c, d, e, f are the available links.

Multi-Entry / Multi-Exit Routines:

* As the name implies multi entry means, multiple entry points and multi exit refers to multiple exit points.
* Although there are circumstances in which it is proper to jump out of a routine and bypass the normal control structure and exit method.
* A user might want to jump out of a routine when an illogical condition has been detected for which it is clear that any further processing along that path could damage the system's operation of data.
* Under such circumstances, the normal return path must be bypassed. In such cases, there is only one place to go - to the system's recovery processing software.
* The correct way to design the routine is to provide an entry parameter within the routine (say, by a case statement) directs the control flow to the proper point.
* Similarly, if a routine can have several different kinds of outcomes, then an exit parameter should be used.

The trouble with multi entry and multi exit routines is that it can be very difficult to determine what the interprocess control flow is, and consequently it is easy to miss important test cases.

Fundamental path selection Criteria :

* Every routine may have several no. of different paths available from its entry to its exit.
* 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 no. of paths.
* The no. of paths become double on each decision and gets multiplied with the no. of iterations on each loop.
* Testing all the available paths will be an expensive approach as there may be some bugs in the routine which results in unnecessary paths.

The alternative is to follow the complete Testing which includes testing the following.

(I) Excercise every path from entry to exit
(II) Excercise every statement or instruction atleast once.
(III) Excercise every branch and case statement, in each direction atleast once.

path Testing Criteria :

Any testing strategy based on paths must atleast both exercise every instruction and take branches in all directions.

There are 3 different testing criteria or strategies as follows.

① path Testing (P_∞) :

* It deals with the execution of paths, if we have tested

all the available control flowgraphs, we have achieved 100% path coverage which is mostly impossible.

② Statement Testing (P_1):

* Statement testing deals with the execution of all the statements in a program atleast once.
* If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
* In other words, we can say 100% node coverage.
* we denote statement coverage as C_1.

③ Branch Testing (P_2):

* Branch Testing deals with the execution of all the branches in a program.
* If we do enough tests to achieve this, then we have achieved 100% branch coverage.
* In other words, we can say 100% link coverage.
* Branch coverage is denoted by C_2.

Common sense and strategies:

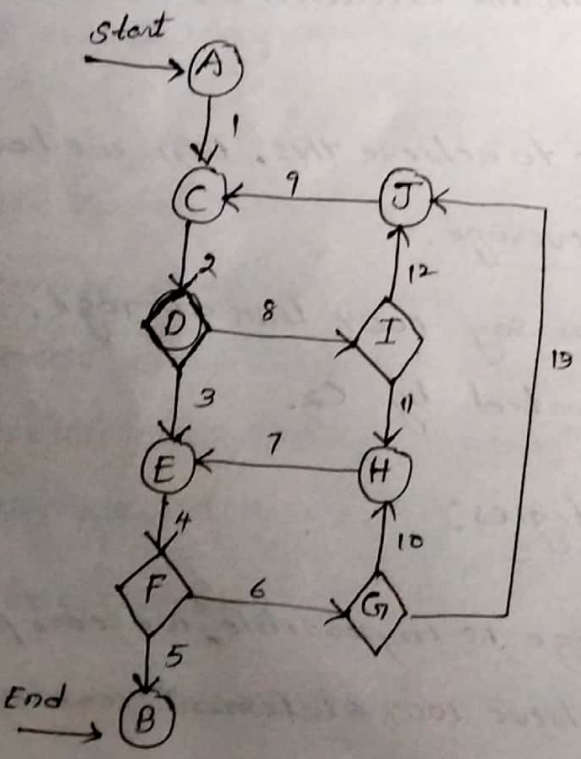
* Though the path coverage is impossible, we can put in some efforts to achieve 100% statement coverage and branch coverage which is the minimum requirement for perp performing unit Testing.
* With the common sense, we can classify code with much probability of finding bugs and code with less bugs separetely.

* Keeping the code with lower probability of bugs untested may not be wrong as it contains less or no bugs.
* Hence the code with higher probability of bugs is tested thoroughly to remove all the bugs.

Which paths to be selected:

* We must pick enough paths to achieve C1 + C2
* During the process of path selection, always prefer simple and easily achievable paths. i.e, complicated paths even if they are few in number must be given the least priority while selection.

eg:



The most obvious path from A to B is (A, C, D, E, F, B). as it is the shortest and simple path.

Issues related to Multi-entry and multi exit routines:

The main drawback of this routine is, control flows are not easily determined which leads to test coverage problem.

The other various issues are as follows

(a) Weak approach:

* To test the program with multi entry and multi exit routines, we need to follow certain procedures.

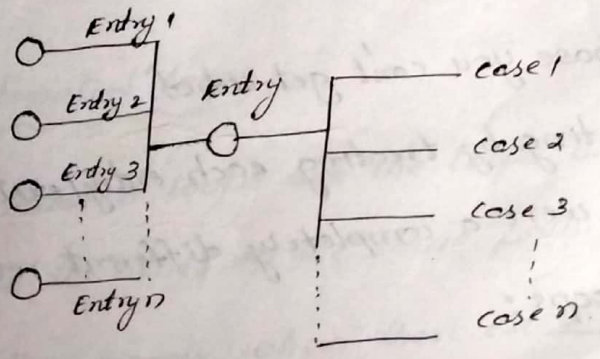
(i) First, build the imaginary single entry routine and imaginary exit routine with pseudo case statements and processes respectively.

(ii) Secondly, concentrate on hypothetical common junction.

* This overall structure will enable us to create test case design for multi entry / multi exit routines.

eg:

Multi entry routine is converted to single entry with case statements as follows



(b) Integration Testing issue:

* Weak approach does not concentrate on interfaces and paths of called components, which are the major aspects of integration testing.

* Unit testing considers each pair of entry and exit as a different routine and tests for all the routines.
* Integration testing tests each pair of entry/exit for every input/output. This testing will be so vast that, it hardly be completed for such routines.

(c) Theory and Tools issue:

* A well formed software is a software, which has a single entry and single exit with a rigid structure.
* Multi entry and multi exit routines comes 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/multi exit) software does not have any structure.

Strategy summary:

Some of the proper ways to test multi entry or multi exit routines is

(i) Get rid of them
(ii) Completely control those you can't get rid of.
(iii) Do stronger unit testing by treating each entry/exit combination as if it were a completely different routine.

Different Kinds of Loops:

There are three different types of loops. They are as follows.

1. Nested Loops
2. concatenated Loops
3. Horrible Loops.

① 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.

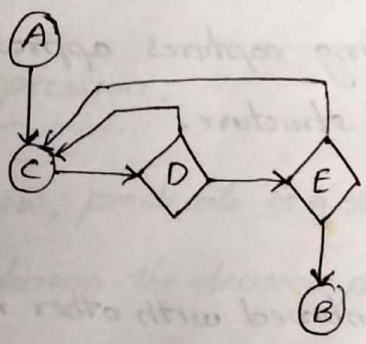
To overcome this complexity, the following steps must be followed.

(I) Begin your test from the most internal loop.

(II) After testing the innermost loop, come out of it and conduct a test on the immediate outer loop.

(III) Once all the loops have been tested move to the other nested loop on the same path.

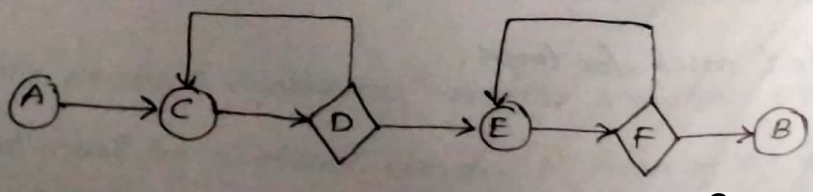
eg:



② Concatenated loops:

* Concatenated loops are the loops which reside one beside the other on the same path.
* In other words, when there exists 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.

eg:



③ Horrible loops:

* Horrible loops are the complexed of all the three loops and may involve nested loops, concatenated loops etc.
* The complext structure of horrible loops makes it very difficult to be tested.
* Hence, horrible loops must be avoided.

Effectivenes of path Testing:

* Approxumately 65% of all the bugs can be caught in unit testing, which is dominated by path testing methods.
* path Testing captures 50% of the bugs that unit testing captures.
* This implies that path testing captures approxumately 35% of bugs in the overall structure.

Limitations of path Testing:

(I) path Testing has to be combined with other methods to improve the overall performance.

(II) path Testing may not reveal all the bugs.

(III) It may not reveal totally wrong or missing functions

(IV) Specification errors can not be caught.

path Testing involves a lot of work i.e

(1) Development of control-flowgraph.

(2) choosing a route that covers all elements in a flowgraph.

(3) writing test cases for loops.

PREDICATES, PATH PREDICATES and ACHIEVABLE PATHS:

predicates:

* Decision is a point in a program at which the control flow gets diverge.
* The direction taken at a decision depends on the value of decision variable
* predicate is a function which is logically executed during the decision processing, hence the result of this function decides the direction of flow.

eg: 'X is greater than zero' is TRUE

* A predicate associated with a path is called path predicates.

predicate Expression:

* As we know, predicate is a function which is logically executed during the decision processing.

eg: consider the predicate

$X + Z > 0$

Now let the value of Z be given using another predicate as $Z = Y + 5$

The substitution of 'Z' value in the first predicate gives you another predicate "$X + Y + 5 > 0$."

This process is known as "predicate interpretation"

* "path predicate expressions" are the collection of expressions that must be fulfilled inorder to achieve the desired path.

* This collection of expressions are satisfied based on the input values provided. These input values must meet all the expressions.
* If all the expressions are met, then that path is choosen else the path is rejected.

eg:

path predicate expression conditions are

A = 18

B+5C+2>0

D-B>=10C

Let the input values of B, C, D be 2, 1, 12 respectively.

Substituting the values in the above predicates, we get

A = 18

B+5C+2>0 ⇒ 2+5(1)+2>0 ⇒ 9>0

D-B>=10C ⇒ 12-2>=10(1) ⇒ 10>=10

∴ all the conditions appear to be correct as per the values, so this path can be choosen.

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.

Achievable paths:

* A desired path can be achieved if the path predicate expressions are satisfied.
* It depends on path predicate expression.

predicate blindness:

* Testing blindness is a situation in which the desired path is achieved for the wrong reason.
* Blindness is a situation which results in the correct path via wrong route unintentionally.

There are three types of predicate blindness

1. Assignment blindness
2. Equality blindness
3. Self blindness.

Let us discuss each of them briefly.

1. Assignment blindness:

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

eg:

correct	Buggy
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.

2. Equality blindness:

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

eg:

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.

③ Self blindness:

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

eg:

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.

PATH SENSITIZING:-

* 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 said to be achievable path. else the path is not achievable.
* "The process of attempting to find set of solutions to the path predicate expressions is called path sensitization."

Heuristic procedures for sensitizing paths:

The first preference for selecting a path must be given to the paths which can be easily sensitized thereby delaying the paths whose solution to path predicate expression is difficult to obtain.

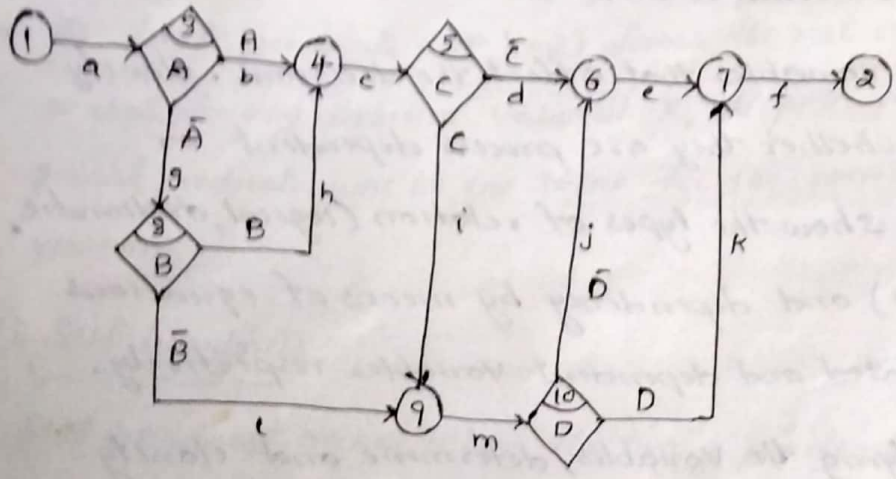
* Identify all variables that affect the decisions. classify them as to whether they are process dependent or independent. show the types of relation (logical, arithmetic or functional) and dependency by means of equations for the correlated and dependent variables respectively.
* After classifying the variables, determine and classify the predicates depending on the input variables into dependent, independent or correlated predicates.
* Consider the uncorrelated and independent predicates for selection of path.
* Now, consider the correlated and independent predicates if they are not covered then start considering the dependent and uncorrelated predicates.
* Display all the input variables, its values, relationship among the variables.
* Every path will produce some set of inequalities, which must be met inorder to select that path.

Eg:

Simple, independent, Uncorrelated predicates

* The upper case letters in the decision boxes of the given figure represents the predicates.

* The capital letters on the links indicate whether the predicate is true (unbarred) or false (barred) for that link.
* There are 4 decisions and 4 predicates in our example.



The following are some of the set of covering paths.

path	predicate values
abcdef	$A\bar{C}$
aghcimkf	ABCD
aglmjef	ABD

From the above table, it is clear that every decisions and links have covered thoroughly.

PATH INSTRUMENTATION:

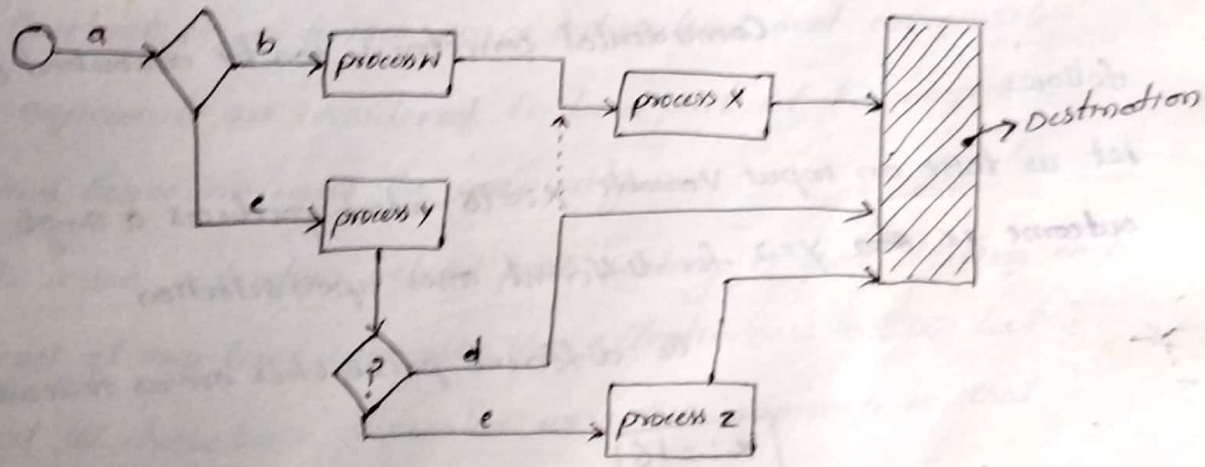
* path instrumentation is a technique used for identifying whether the outcome of a test is achieved through the desired path or wrong path.
* The basic goal of path instrumentation is to produce the desired outcome with the assistance of intended path only.
* It is a form of interpretive trace program.

* The only drawback of the trace programs is its additional large information content, which is of no use.
* To overcome this drawback, many instrumentation methods have developed.

eg: link marker method of path instrumentation

Link Markers or Traversal marker:

* Link marker is also known as traversal marker which uses small case letters for naming every link.
* Whenever a link is passed, its name is recorded in the marker.



In the example, the preferred route is through "acd" but, due to an error in the structure, we reached the correct destination via process X.

* To avoid this scenario, we need to use two markers on each link, one at the start of the link and the other at the end of link. Now path name includes both the markers of the link.
* It is shown in the following fig.

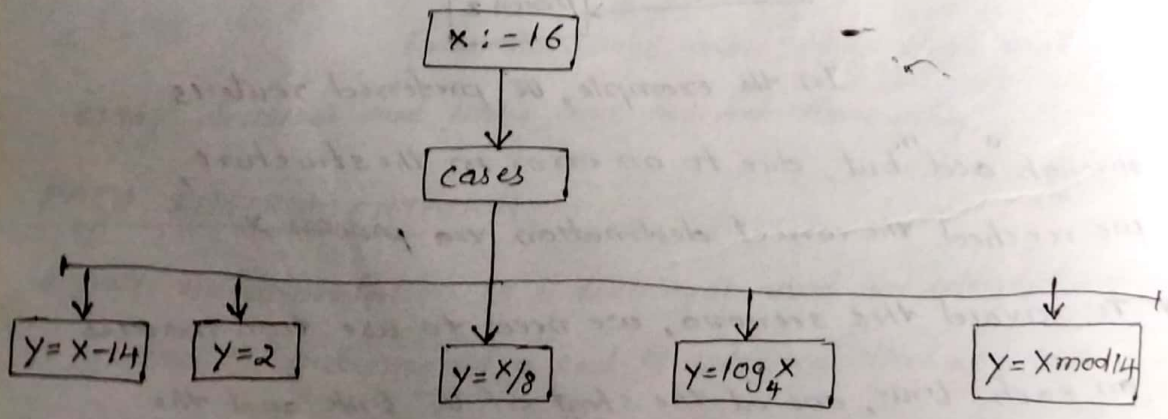
Coincidental correctness:

* Since the test outcome is considered as a part of test 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 enough, this type of condition is known as coincidental correctness.
* Simply, it can be defined as a condition in which we check whether the expected outcome of a test is generated or not.

Coincidental correctness can be described as follows.

Let us take an input variable X:=16 which produces a single outcome i.e Y=2 for different cases upon selection.

The different possibilities are as follows



* The selection of test cases carried out in this manner does not help to achieve path coverage, because the desired outcome could be achieved by the wrong path.
* If we jumble the test cases, the outcome will be the same, but the right path is not assured.

* The basic goal of path instrumentation is to achieve the outcome of the test by the desired path only.

Applications of path Testing:

path testing is process of testing all the available paths in a program from an entry to an exit in such a way that all entire path is thoroughly tested.

path testing implementation and applications can be categorized as follows.

① Integration, coverage and paths in called components:

* The concept of path testing is mainly employed in unit testing.
* Coverage issues arises, since subroutines and corequisite components are considered to be a part of the component and hence increases the complexity.
* The main intention behind path testing is that, testing each level at any time increases the effectiveness of the test but the draw back associated with this approach is that it results in predicate coverage and blindness.

② New code:

* The new code (components) has to be given higher priority for testing than the old trusted components.

③ Maintenance:

* path testing will be carried out mostly on modified components.

④ Rehosting:

* Rehosting is a process of transforming the old software environment into a new more friendly environment in which

rebasted software can run more effectively.

* path testing along with the structural test generators will produce a powerful, efficient and an effective rebasting process.

* This approach might be even more costlier than building the new software, but it provides us with an environment which suits the requirements of a software.

UNIT-II

Transaction Flow and Data Flow Testing :-

Transaction Flows :-

Transaction :-

A Transaction is defined as a unit of work handled by the system user.

* Each transaction is usually associated with an entry point and an exit point.
* The execution of the transaction starts at an entry point and end at an exit point.
* After getting executed, the transaction doesn't remain in the system.
* All the results of the transactions are stored in the form of records (i.e history) in the system.

Every Transaction related to an online information system consists of 12 steps.

1. It accepts the input
2. Validates the input
3. After validating, an acknowledgement is sent to the user.
4. Input processing is performed
5. Searches a file, if necessary
6. User processes the request
7. Accepts the input
8. Validates the input
9. Requests are further processed.
10. Updates the file.

11. Output is transmitted.

12. The output is stored in the form of records and the transaction is cleaned up from the system.

All the above steps are processed by the user as a single Transaction.

Transaction Flow:-

The Transaction flow is a test design model where testing conceptual tools are considered.

* The Transaction flow occurs to test the structural and behavioural model of a system.
* The components involved in a transaction flowgraph are links and nodes.
* These links and nodes represent the entire flow of a transaction.

Thus conceptual model is a powerful tool for testing a system.

Eg:

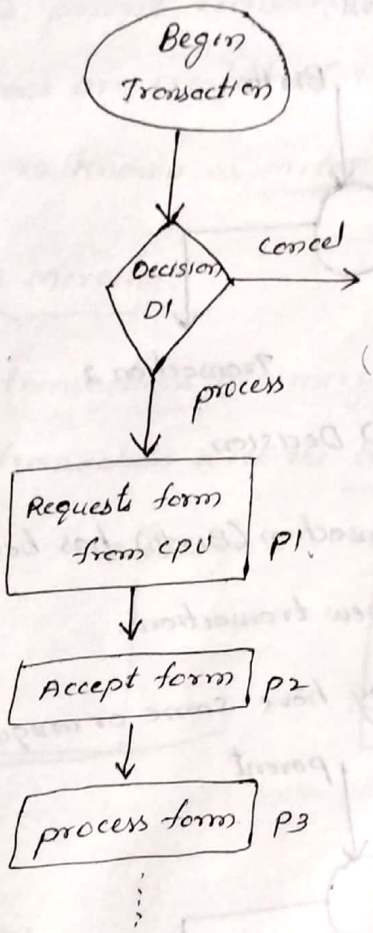
* A Transaction flow is processed in forms. Each form consists of several pages with records and files in it.
* A system is taken as the "terminal controller" to process these forms.

Only those forms which are located on a central computer are requested for processing.

* 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 computer.
* The terminal controller in turn transmits the code to the user to check the input.



Drawbacks of Transaction Flow Model:

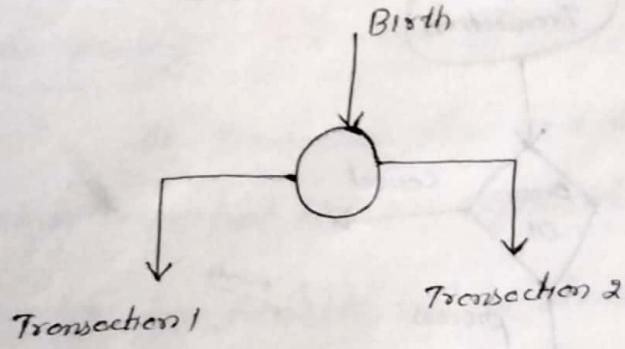
* Transaction flows don't have a good structural design for code.
* Transactions are interactions between modules.
* A good system design indicates that there is no implementation of new transaction or changing an existing transaction.
* The decision nodes of a transaction flowgraph can be complicated.

Eg:

A transaction can give birth to others and can also merge with others in many of the systems.

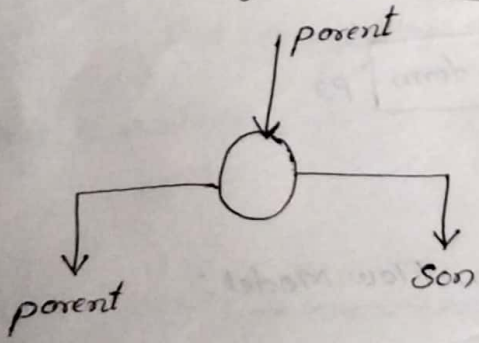
* From its creation to its death, a Transaction will have unique name.

(1) Birth of New Transactions:



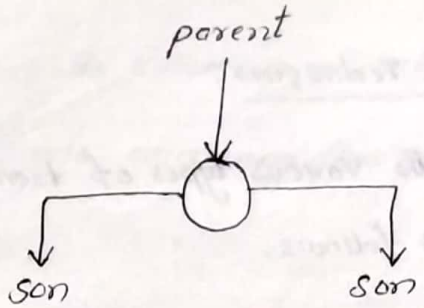
(a) Decision

* In the above fig (a), a transaction (Birth) has been created which gives birth to two new transactions.
* Both these transactions may have same or unique identity.



(b) Biosis

* In the above fig (b), the parent transaction gives birth to two new transactions.
* One transaction has the same identity as the "parent" and the other transaction results in different identity "Son".
* This situation is called Biosis.

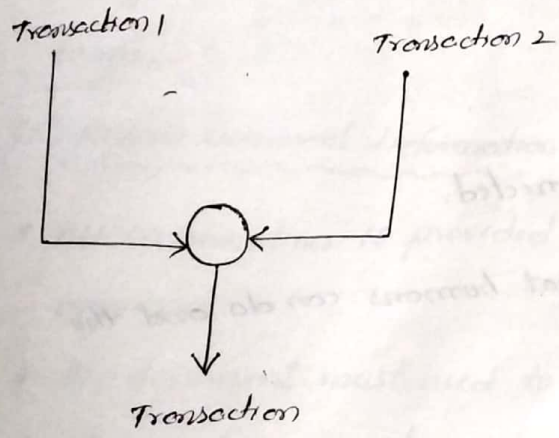


(c) Mitosis

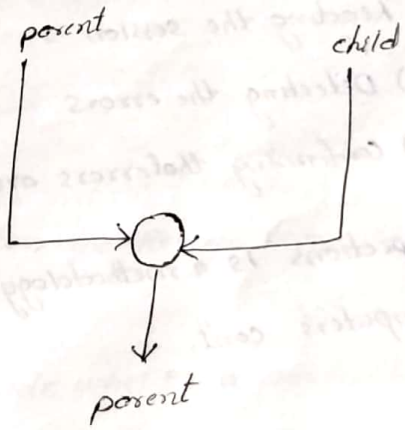
* In fig (c), the parent transaction is destroyed and two new transactions are created of the same identity.
* This situation is known as Mitosis.

(II) Transactions Merging:

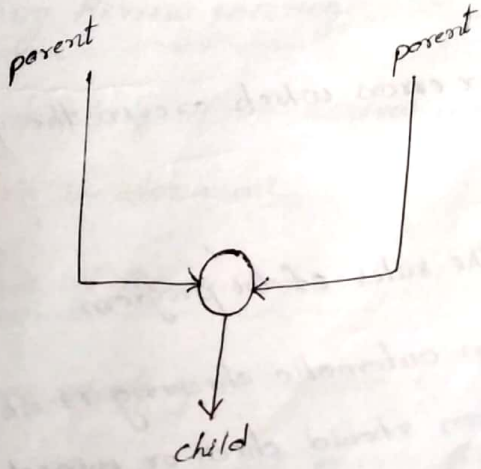
In this, two transactions are merged at decision node giving a new transaction with the same or different identity.



(a) Junction



(b) Absorption



* This Merging is as troublesome as transaction flow births.

Transaction Flow Testing Techniques:

The various types of transaction flow testing strategies are as follows.

(1) **Inspections:**

* An inspection is a technique used to detect errors by reading a group of code.
* This is often done by the developers or the programmers.

The following are the duties of the programmer.

(i) Distributing the requirements, scheduling and inspecting module.

(ii) Leading the session

(iii) Detecting the errors.

(iv) Confirming that errors are corrected.

* Inspections is a methodology that humans can do and the computers can't.

eg:

(i) Checking syntax errors:

Inspecting those syntax errors which are in the program

(ii) Violating Rules:

* It means breaking the rules of the program
* If the facilities for an automatic checking is not available then the person should check or inspect the code manually.

The following are some other examples comparison of code, comparing flow graph etc.

2. Reviews :-

* Reviews are used to check the semantics of a document.
* The quality of the documents can be examined and assured by eliminating the defects and mistakes from the documents.

The following steps are involved in a Review.

(I) Review planning :

* A Review has to be planned.
* Every individual review has a review leader and a review team.

(II) Review Document Information :

* All information is provided when the Review team is organized.
* The document must used to decide whether a particular statement is correct or wrong.

(III) preparing Review meeting :

* All the members of the Review meeting Team analyse the defects in the document.

(IV) objective of Review meeting :

* The review meeting is handled by a ~~new~~ review leader.
* The Review leader must ensure that the requirements of the ~~a~~ all the reviewers has met to find the defects.

(v) Review scheduling:

* If the result of the first review was not acceptable, then another review has been assigned by the Review Manager. to correct the defects.

3. Walkthroughs:

* A walkthrough is an informal review method.
* It is a way of finding defects or problems in the written documentation.
* Walkthroughs is a technique or a set of procedures to detect errors by reading a group of code.

Walkthrough has a team similar to that of a reviewer team and consist of three people

(1) Secretary, to record errors
(2) Leader who handles the team
(3) Tester to clear defects.

4. path Selection:

* The path selection for system testing that is based on transaction flows from the other unit tests that are based on control flow graphs.
* In this a longest path is taken to find the bugs
* The path is reviewed and bugs are removed.

5. Sensitization:

* The simple paths in transaction flows are easily to sensitize

when compared to bug related difficult bugs.

6. Instrumentation:

* Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
* The information of the path taken for a given transaction must be kept with that transaction.

7. Test Databases:

* About 30%-40% of the effort of transaction flow test design is the design and maintenance of the test databases.
* They contain the information which are related in the design of test cases.
* The design of these databases is no less important than the design of the system data structures.

8. Execution:

* It means, all the test cases must be executed automatically.

Data Flowgraphs:

The data flow graph is a graph consisting of nodes and directed links.

Data object state and usage:

* Data objects can be created, killed and/or used.
* They can be used in two distinct ways
  (I) in calculation
  (II) as a part of control flow predicate.

d – defined, created, initialized, etc

k – killed, undefined, released

u – used for something

c – used in a calculation

p – used in a predicate

① Defined:

An object is defined manually when it appears in a data declaration or automatically when it appears on the left hand side of an assignment statement.

② Killed or Undefined:

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

eg:

If A had been previously defined and we do a new assignment such as A:=17, we have killed A's previous and redefined A.

③ Usage:

* A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
* It is used in a predicate (p) when it appears directly in a predicate (for eg. IF A>B).

Data Flow Anomalies:

* An Anomaly is a term that leads to inconsistency in data flow analysis.

* During data flow analysis, every variable is inspected. The following are the various types of variables used in Data flow Analysis.

Variables		Definition
Defined (d)	–	Value assigned to a variable
Referenced (r)	–	Value read or used by a variable
Undefined (u)	–	Variable that has no defined value

Depending on these variables, three types of Anomalies can be classified.

① ur Anomaly

② du anomaly

③ dd anomaly.

Let us discuss each of them briefly.

① ur anomaly:

During data flow analysis, if the undefined value of a variable (u) is read (r) then it is known as 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.

eg:

A:=10, A:=15

therefore, it accepts A:=15 value.

Let us take the following example to detect the anomalies

```
void swap(int a, int b)
{
 int get;
   if(a>b)
   {
    b=get;
    b=a;
    get=a;
    }
}
```

Depending on the usage of variables, the anomalies can be detected.

Detection of Anomalies:

1. ur anomaly:

* In the above example, the variable get is used on the right side of assignment i.e (b=get)
* the variable get has undefined value as it is not initialized.
* this undefined variable is being read and hence it results in ur anomaly.

2. du anomaly:

* In the above eg, the variable 'get' has a defined value in the last assignment (i.e get=a)

* The defined variable cannot be used anywhere in the function because only those variables are valid which are inside the function.

3 dd anomaly:

* The variable 'b' is used twice on the left side of an assignment.
* The first assignment becomes invalid or unused whereas the second value is taken.

Data Flow Anomaly State Graph:

The data flow anomaly model prescribes that an object can be in one of four distinct states.

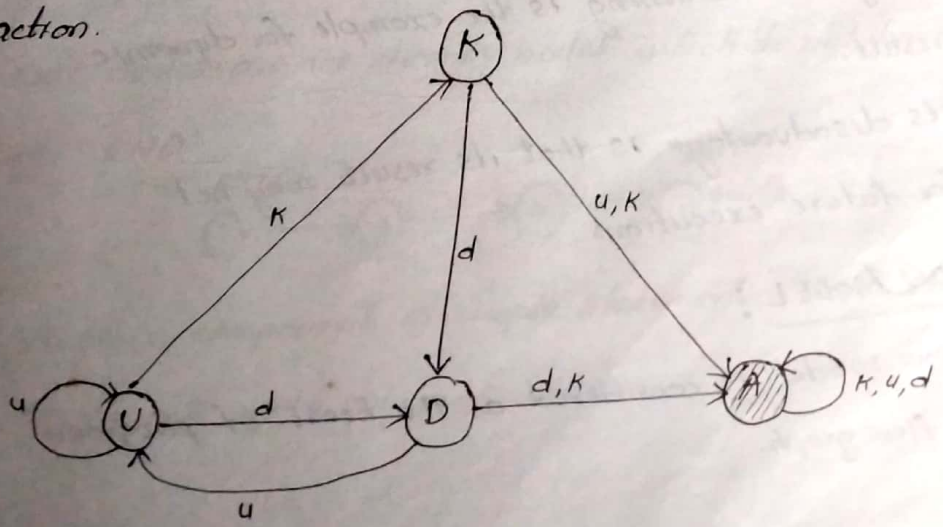
K — undefined, previously killed, does not exist

D — defined but not yet used for anything.

U — has been used for computation or in predicate

A — Anomalous.

* The capital letters (K, D, U, A) which denote the state of the variable and the lower case letters (k, d, u) denotes the program action.



Static Versus Dynamic Anomaly Detection:

Static Analysis for Data flow Anomaly Detection:

* Static Analysis is done at compile time in data flow anomaly detection
* It is based on the source code before execution.
* Static Analysis for data flow anomaly detection is more compact and faster than dynamic analysis.
* Source code syntax error detection is the example for static analysis result.
* The disadvantage of static analysis is that its results are not accurate.

Dynamic Analysis for Data flow Anomaly detection:

* Dynamic Analysis is done at run time in data flow anomaly detection.
* It is based on the intermediate values that result from the program's execution.
* Dynamic Analysis can be as fast as program execution.
* A division by zero warning is the example for dynamic analysis result.
* One of its disadvantage is that its results may not correct for future executions.

DATA FLOW MODEL:

* Data flow model is considered as the heart of programs control flow graph.

* It consists of links which are denoted by symbols d, K, u, c, p or sequence of symbols like dd, du etc.
* These symbols are called link weights

The symbols are defined as

d = Defined object

K = Killed object

u = used object

c = used in calculation

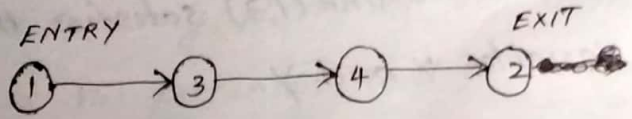
p = used in predicate

Components of Data flow Model:

* There exists a unique node for each and every statement
* A node has both inlinks and outlinks but the condition is that each and every node must have atleast one outlink and one inlink
* There are two types of Nodes
  (i) Entry nodes
  (ii) Exit nodes
* Entry nodes are the dummy nodes which do not have inlinks

  eg: BEGIN
* Exit nodes are the dummy nodes which do not have outlinks.

  eg: END



* Another components is simple statements. These are the components with only one outlink.

Strategies in Data Flow Testing :

Data flow testing strategies are based on the program's control flow graph.

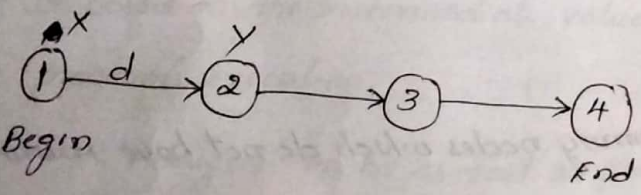
The following are the various types of data flow testing strategies.

1. All du paths (ADUP)
2. All uses (AU)
3. All P uses or some C uses (APU+C)
4. All C uses or some P uses (ACU+P)
5. All P uses (APU)
6. All definition (AD) strategy.

Let us discuss each of them briefly.

① All du paths (ADUP) :

A du path is a path which has a defined variable and has a computational use or a predicate use of that variable.



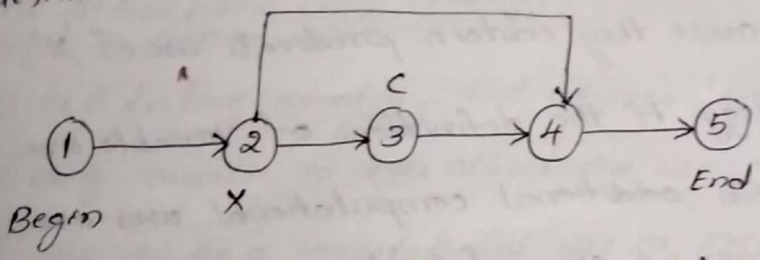
In the above fig, variables X and Y are used on the data flow link (1,3).

* Any test that starts from the link (1,3) satisfies this strategy only for variables X and Y.

② All uses (AU) :

* All uses (AU) strategy is a data flow testing strategy that requires 'every definition from atleast one path segment to use every variable that can be reached by

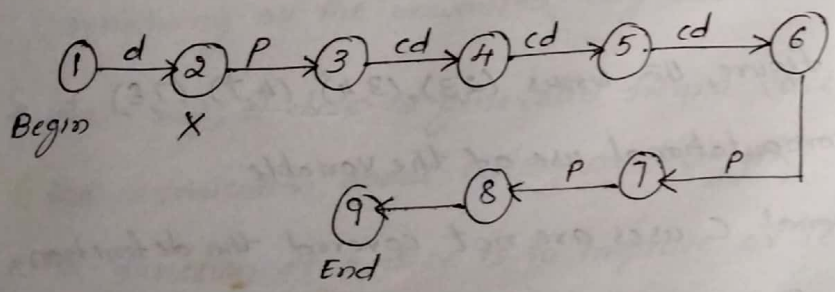
that definition.



* In AU strategy either (2,3,4) or (2,4) can be included, but we don't have to use both to begin a path.
* The link (2,3,4) is included as it is the only way to reach the C (computational) use in that link.

③ All P uses or some C uses (APU+C) strategies:

* If a variable has a predicate use then there is no need to select a computational use.
* APU+C is a strategy that requires 'every definition of every variable include atleast one path from that definition to every predicate use.'

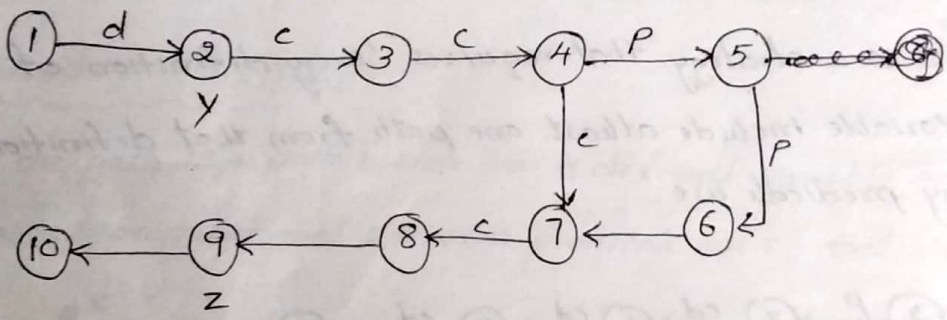


* In APU+C strategy, every definition of every variable has a path to every P-use of that definition. If there is no P-use of that definition, then a C-use is considered.
* In the above fig, links (1,2), (3,4), (4,5), (5,6) must be included in test cases as they contain definitions for variable 'X'.

* The links (2,3), (6,7), (7,8) must be included in test cases because they contain predicate use of 'x'.
* In this strategy, if the definitions of variable are not covered then additional computational uses are required to cover up every definition.

④ All C uses or some P uses (ACU+P) strategy:

* In this testing strategy, first ensure that every definition has a computational use of that definition and if any definition is not covered, predicate use cases are added to cover up the definition.



* In the above figure, the links (2,3), (3,4), (4,7), (7,8) involves the computational use of the variable.
* If the ~~additional~~ C uses are not covered the definition, then additional p uses are covered.

⑤ All p uses (APU) strategy:

* In this testing strategy, 'every definition of every variable has a path to every ~~use~~ p-use of that definition'. If there is no p use ~~of~~ in that definition, then it is dropped from contention.

⑥ All definitions (AD) strategy:

* AD is a testing strategy that requires 'every definition of every variable to cover atleast one use of that variable'. The use can be a computational use or predicate use.

Slicing, Dicing, Data flow and Debugging:

* Dicing and debugging are the concepts related to removal of unwanted bugs.
* Slice is a part of the program on which testing is performed.
* The term "dicing" is an element of slicing, which actually is a part of a program with some selected set of statements.
* Dicing is defined as the process of refining slice by removing all the unwanted bug statements.
* Basically, a dice is generated from a slice which contains the information about testing or debugging.
* The function of a dice is to improve or refine a slice by removing unwanted statements in a program.
* Data flow is defined as the process of reading the variables
* Slicing is divided into two types
  (i) static slicing
  (ii) Dynamic slicing

* Static slicing is a part of program defined with respect to a given variable z and a statement x.
* Dynamic slicing is a refinement of static slicing.

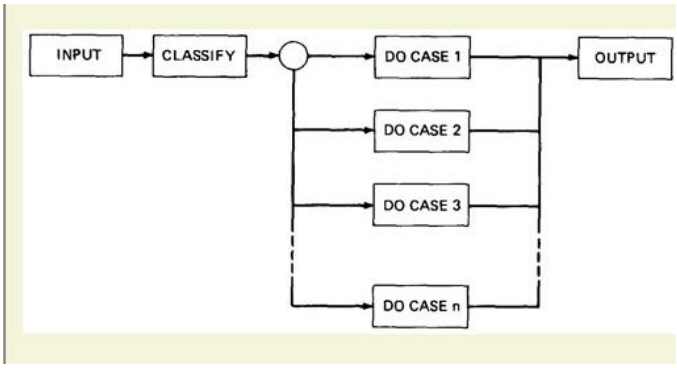
Applications of Data Flow Testing :

* Data flow testing is used to detect the different abnormalities that may arise due to data flow anomalies
* It shows the relationship between the data objects that represents data.
* Data flow testing strategies are used in determining the usage of variables that are included in the test set.

UNIT- III

DOMAIN TESTING

- Domain testing, as practiced, is usually applied to one input variable or to simple combinations of two variables, based on specifications.
- For example, you're doing domain testing when you check extreme values of an input variable.
- Domain testing as a theory, however, has been primarily structural.



Schematic Representation of Domain Testing.

Domain Closure

- Figure 6.2 shows three situations for a one-dimensional domain—i.e., a domain defined over one input variable; call it *x*.
- As in set theory, a domain boundary is closed with respect to a domain if the points on the boundary belong to the domain. If the boundary points belong to some other domain, the boundary is said to be open.
- Figure 6.2 shows three domains called D1, D2, and D3.
- In Figure 6.2a, D2's boundaries are closed both at the minimum and maximum values; that is, both points belong to D2. If D2 is closed with respect to *x*, then the adjacent domains (D1 and D3) must both be open with respect to *x*.
- Similarly, in Figure 6.2b, D2 is closed on the minimum side and open on the maximum side, meaning that D1 is open at the minimum (of D2) and D3 is closed at D2's maximum.
- Figure 6.2c shows D2 open on both sides, which means that those boundaries are closed for D1 and D2. The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, x >= 0 when x > 0 was intended.

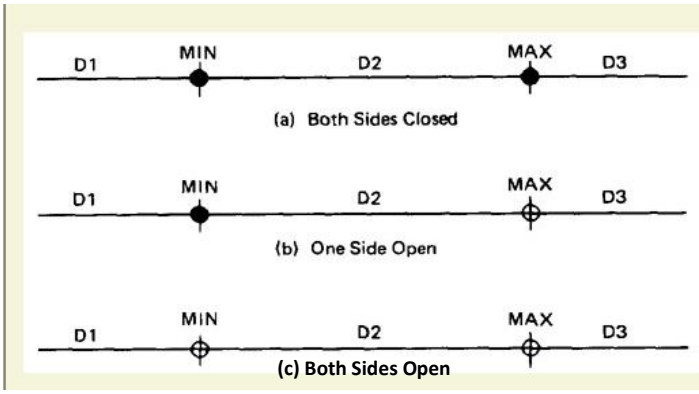


Figure 6.2. Open and Closed Domains.

Domain Dimensionality

Every input variable adds one dimension to the domain. One variable defines domains on a number line, two variables define planar domains, three variables define solid domains, and so on.

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 may in turn mean that control-flow predicates are wrong.

1. ***Double-Zero Representation*****—**In computers or languages that (unfortunately) have a distinct positive and negative zero, boundary errors for negative zero are common.

2. ***Floating-Point Zero Check*****—**A floating-point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself, multiplied by zero, or created by some operation that forces a zero value.

3. ***Contradictory Domains*****—**An implemented domain can never be ambiguous or contradictory, but a specified domain can. A contradictory domain specification means that at least two supposedly distinct domains overlap. Programmers resolve contradictions by assigning overlapped regions to one or the other domain for a 50-50 chance of error.

4. ***Ambiguous Domains*****—**Ambiguous domains means that the union of the specified domains is incomplete; that is, there are either missing domains or holes in the specified domains.

5. ***Overspecified Domains*****—**The domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.

6. ***Boundary Errors*****—**Domain boundary bugs are discussed later section, but here's a few: boundary closure bug, shifted, tilted, missing, extra boundary.

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

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

Restrictions

General

Domain testing has restrictions, as do other testing techniques. They aren't restrictions in the sense that we can't use domain testing if they're violated—but in the sense that if you apply domain testing to such cases, tests are unlikely to be productive because they may not reveal bugs.

Coincidental Correctness

Coincidental correctness is assumed not to occur. Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. Although we're not focusing on outcome in domain testing, we still have to look at outcomes to confirm that we're in the domain we think we're in. If we're plagued by coincidental correctness we may misjudge an incorrect boundary.

Representative Outcome

Domain testing is an example of **partition testing. Partition-testing** strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain. If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct.

NICE DOMAINS AND UGLY DOMAINS

Nice Domains

- The below figure shows some nice, typical, two-dimensional domains.
- The boundaries have several important properties discussed below.
- They are linear, complete, systematic, orthogonal, closure consistency, convex and simply connected.
- To the extent that domains have these properties, domain testing is as easy as testing gets.
- To the extent that these properties don't apply, testing is as tough as it gets.
- What's more important is that bug frequencies are much lower for nice domains than for ugly domains.

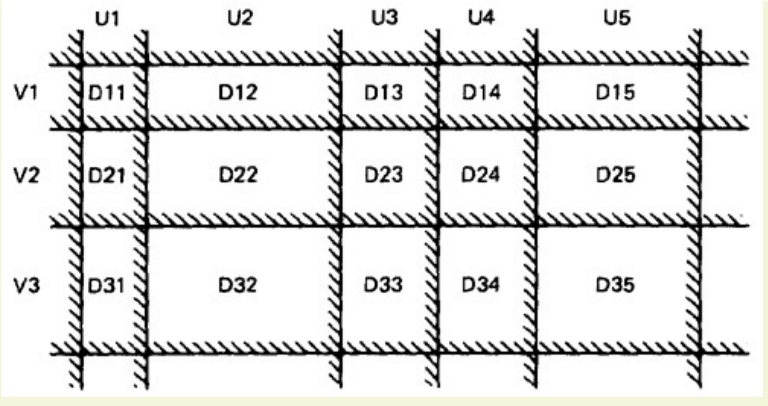


Figure 6.3. Nice Two-Dimensional Domains.

1. **Linear Boundaries**

Nice domain boundaries are defined by linear inequalities or equations—*after interpretation in terms of input variables*. My guess is that, in practice, more than 99.99% of all boundary predicates are either directly linear or can be linearized by simple variable transformations.

2. Complete Boundaries

Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions. Figure 6.4 shows some incomplete boundaries. Boundaries A and E have gaps. Such boundaries can come about because the path that hypothetically corresponds to them is unachievable. The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds. If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

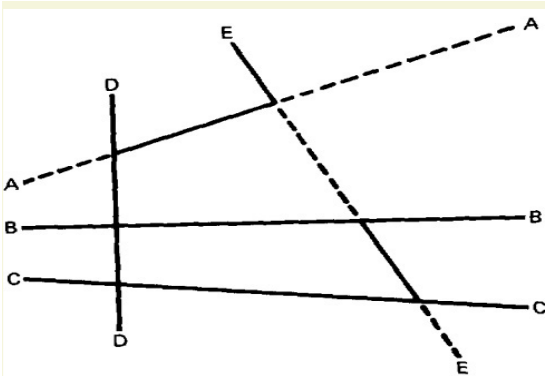


Figure 6.4. Incomplete Domain Boundaries

3. Systematic Boundaries

By **systematic boundaries,** it means boundary inequalities related by a Simple function such as a constant. In Figure 6.3 for example, the domain boundaries for u and v differ only by a constant. We want relations such as

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

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 over i and c that yields a constant.

4. Orthogonal Boundaries

The U and V boundary sets in Figure 6.3 are **orthogonal;** that is, every inequality in V is perpendicular to every inequality in U. The importance of this property cannot be minimized. If two boundary sets are orthogonal, then they can be tested independently. In Figure 6.3 we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6 + 4 = 10 (O(n)).

5. Closure Consistency

Figure 6.6 shows another desirable domain property: boundary closures are consistent and systematic. The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies—e.g., the boundary lines belong to the domains on the right.

6. Convex

A geometric figure (in any number of dimensions) is **convex** if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within the figure. Nice domains are convex; dirty domains aren't.

7. Simply Connected

Nice domains are **simply connected;** that is, they are in one piece rather than pieces all over the place interspersed with other domains. Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

Ugly Domains

Ugly domains have to be **simplified** by the programmers and Testers.

1. Nonlinear Boundaries

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

2. Ambiguities and Contradictions

Figure 6.7 shows several domain ambiguities and contradictions. Domain ambiguities are holes in the input space. The holes may lie within domains or in cracks between domains. A hole in a one-variable input space is easy to see. An ambiguity for two variables can be difficult to spot, especially if boundaries are lines going every which way. Ambiguities in three or more variables are almost impossible to spot without formal analysis—which means tools.

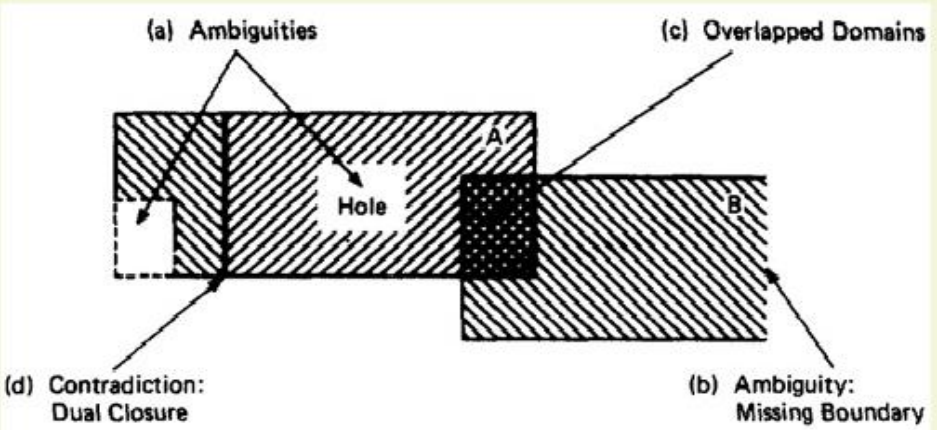


Figure 6.7. Domain Ambiguities and Contradictions

3. Simplifying the Topology

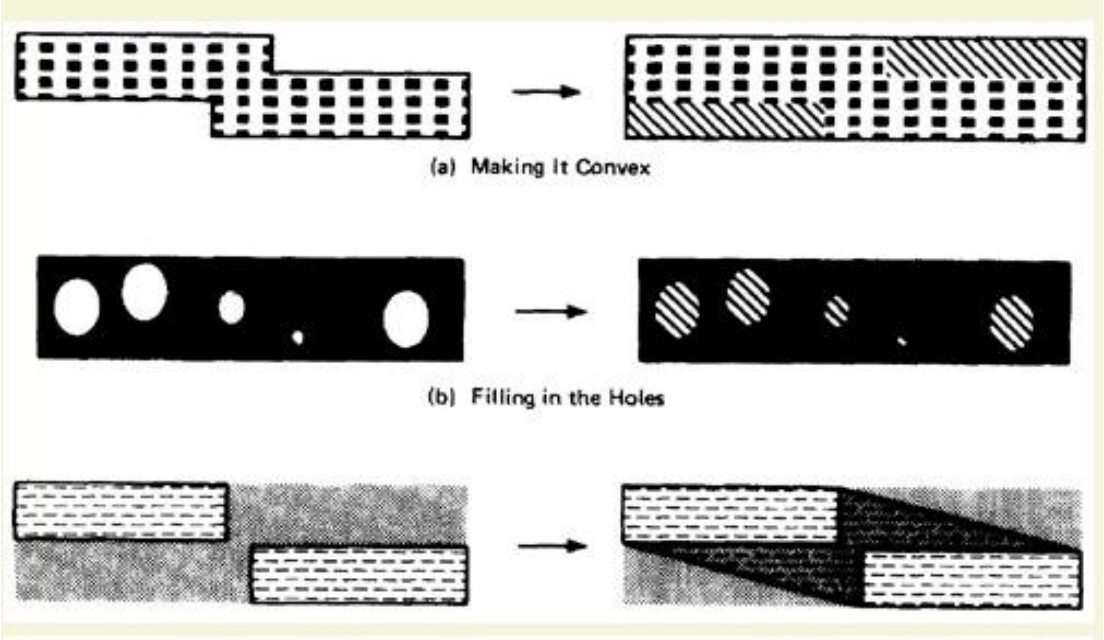
The programmer's and tester's reaction to complex domains is the same—simplify. There are three generic cases: concavities, holes, and disconnected pieces.

Programmers introduce bugs and testers design test cases by:

(a) smoothing out concavities

(b) filling in holes and

(c) joining disconnected pieces



4. Rectifying Boundary Closures

If domain boundaries are parallel but have closures that go every which way (left, right, left, . . .) the natural reaction is to make closures go the same way (see Figure 6.9). If the main processing concerns one or two domains and the spaces between them are to be rejected, the likely treatment of closures is to make all boundaries point the same way.

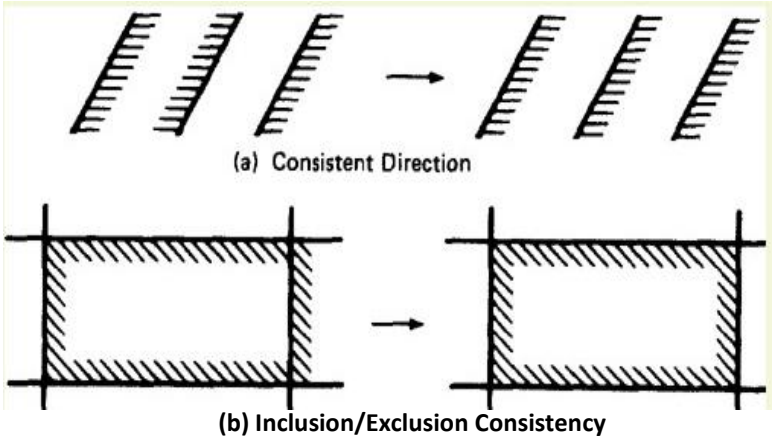


Figure 6.9. Forcing Closure Consistency

DOMAIN TESTING

Overview

The domain-testing strategy is simple, albeit possibly tedious.

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

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

3. Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant test points.

4. Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so, how.

5. Run enough tests to verify every boundary of every domain.

An **interior point** (Figure 6.10) is a point in the domain such that all points within an arbitrarily small distance (called an **epsilon neighborhood**) are also in the domain. A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in the domain. An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain

An **on point** is a point on the boundary. If the domain boundary is closed, an **off point** is a point near the boundary but in the adjacent domain. If the boundary is open, an off point is a point near the boundary but in the domain being tested.

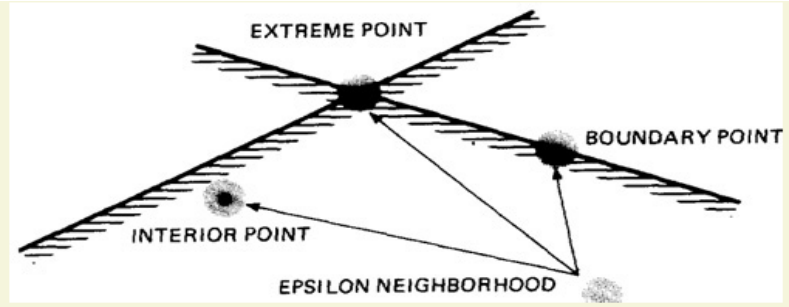


Figure 6.10. Interior, Boundary, and Extreme Points.

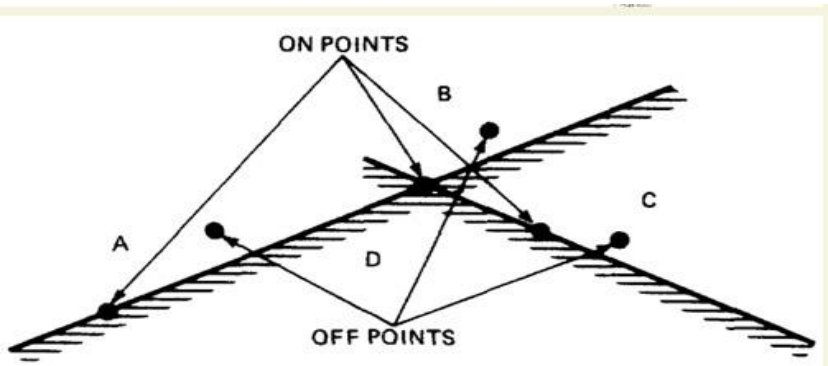
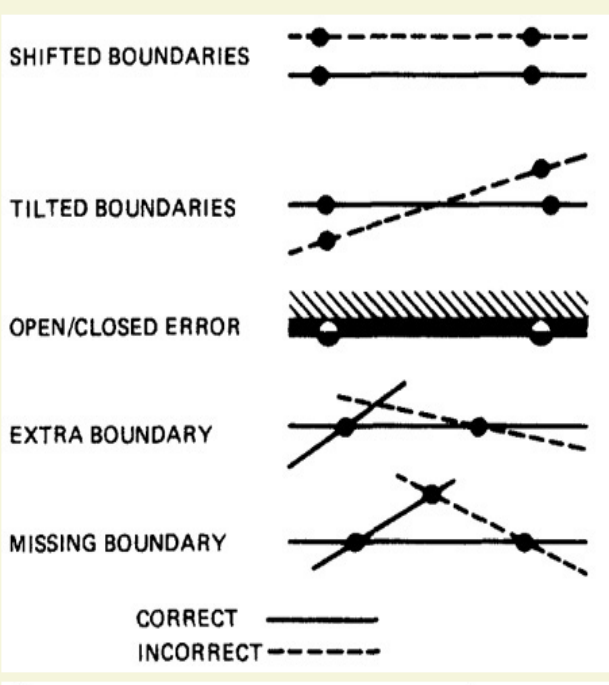


Figure 6.11. On Points and Off Points.

The below figure shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.



Testing One-Dimensional Domains

Figure 6.13 shows possible domain bugs for a one-dimensional open domain boundary. The closure can be wrong (i.e., assigned to the wrong domain) or the boundary (a point in this case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

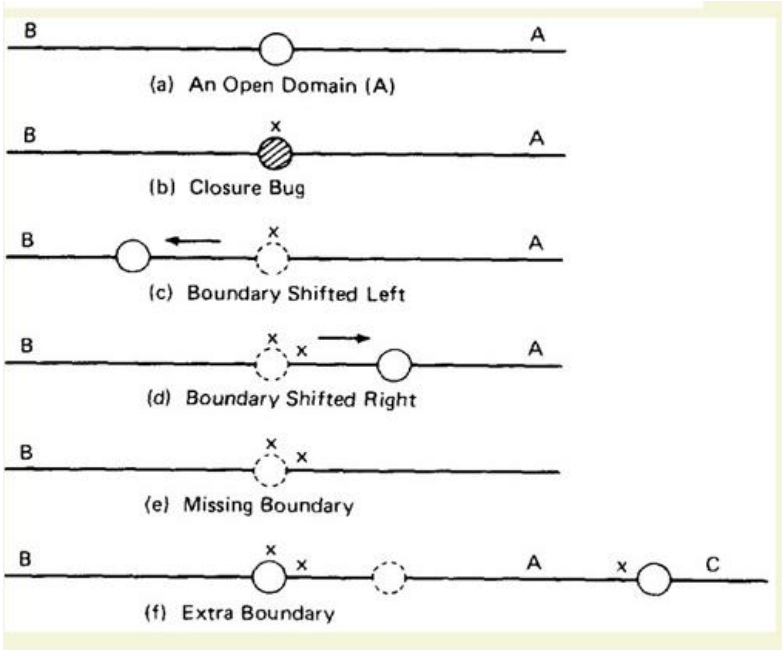


Figure 6.13. One-Dimensional Domain Bugs, Open Boundaries.

In Figure 6.13a we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts > to >= or < to <= (Figure 6.13b). One test (marked x) on the boundary point detects this bug because processing for that point will go to domain A rather than B.

In Figure 6.13c we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing. Note that we can't distinguish between a shift and a closure error, but we do know that we have a bug.

Figure 6.13d shows a shift the other way. The on point doesn't tell us anything because the boundary shift doesn't change the fact that the test point will be processed in B.

For closed domains look at Figure 6.14. As for the open boundary, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside the domain.

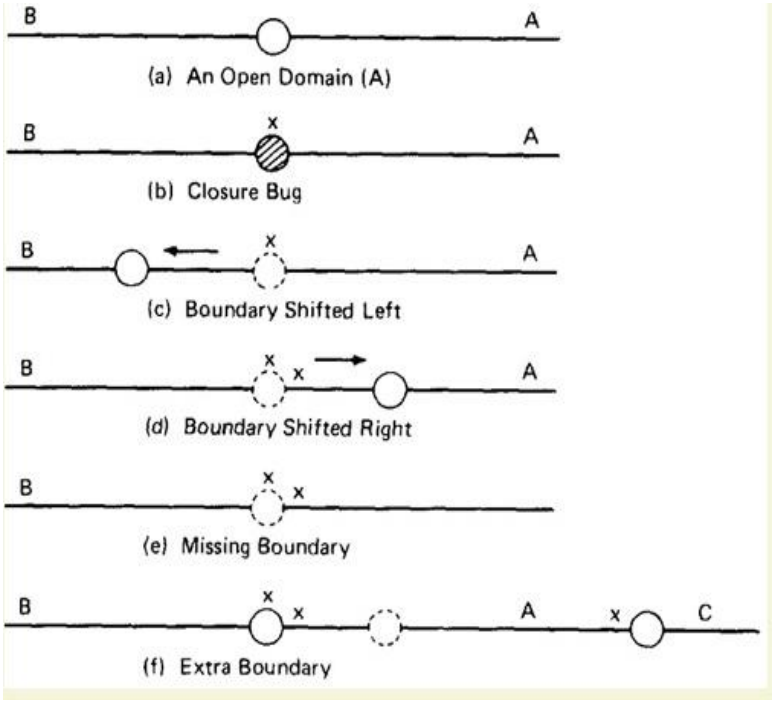


Figure 6.14. One-Dimensional Domain Bugs, Closed Boundaries

Testing Two-Dimensional Domains

Figure 6.15 shows domain boundary bugs for two-dimensional domains. A and B are adjacent domains and the boundary is closed with respect to A, which means that it is open with respect to B. We'll first discuss cases for closed boundaries; turn the figure upside down to see open boundary cases.

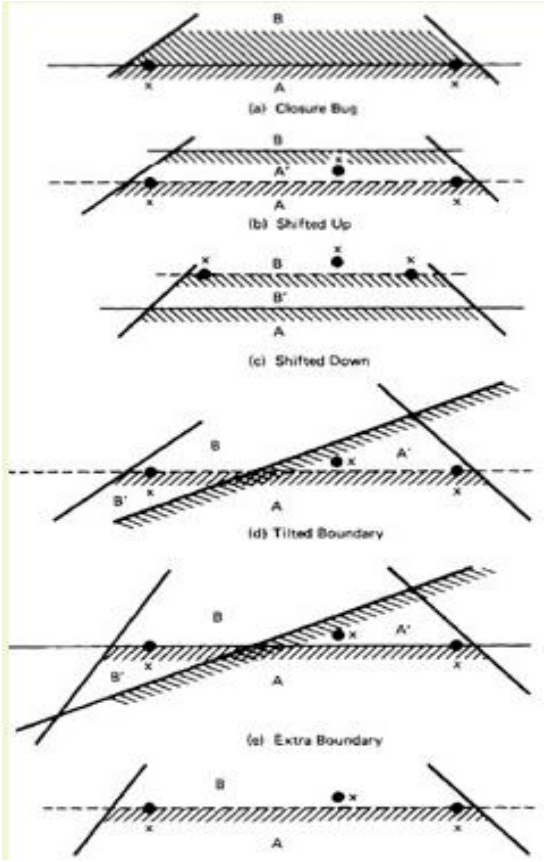


Figure 6.15. Two-Dimensional Domain Bugs

1. ***Closure Bug***—Figure 6.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, $x >= k$ when $x > k$ was intended, or vice versa). The two on points detect this bug because those values will get B rather than A processing.

2. ***Shifted Boundary***—In Figure 6.15b the bug is a shift up, which converts part of domain B into A processing, denoted by A'. This result is caused by an incorrect constant in a predicate, such as $x + y >= 17$ when $x + y >= 7$ was intended. The off point (closed off outside) catches this bug. Figure 6.15c shows a shift down that is caught by the two on points.

3. ***Tilted Boundary***—A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, $3x + 7y > 17$ when $7x + 3y > 17$ was intended. Figure 6.15d has a tilted boundary, which creates erroneous domain segments A' and B'. In this example the bug is caught by the left on point.

4. ***Extra Boundary***—An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug. The extra boundary in Figure 6.15e is caught by two on points

5. ***Missing Boundary***—A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and, as the extra boundary can, will cause many test failures although there is only one bug. A missing boundary, shown in Figure 6.15f, is caught by the two on points because the processing for A and B is the same—either A or B processing.

Equality and Inequality Predicates

Equality predicates such as $x + y = 17$ define lower-dimensional domains. For example, if there are two input variables, a two-dimensional space, an equality predicate defines a line—a one-dimensional domain. Similarly, an equality predicate in three dimensions defines a planar domain. We get test points for equality predicates by considering adjacent domains (Figure 6.18). There are three domains. A and B are planar while C, defined by the equality boundary predicate between A and B, is a line. Applying the two-on, one-off rule to domains A and B and remembering **O**pen **O**ff **I**nside, we need test point *b* for B and point *a* for A and two other points on C (*c* and *c*') for the on points. This is equivalent to testing C with two on and two off points. There is a pathological situation in which the bug causes the boundary to go through the two selected off points: it can be detected by another on point at the intersection between the correct domain line and the two off points (marked *d*).

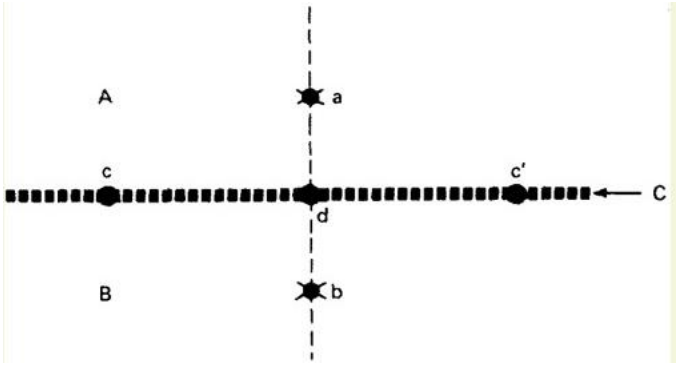


Figure 6.18. Equality Predicates.

DOMAINS AND INTERFACE TESTING

Don't be disappointed by domain testing because it's difficult to apply to two dimensions and humanly impossible for more than two. Don't reject it because it seems that all a person can do is handle one dimension at a time and you can't (yet) buy tools to handle more than one dimension.

Domain span is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least): compatible domain spans and compatible closures.

Domains and Range

The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function is defined. In most of testing we are only minimally concerned with output values.

Closure Compatibility

Assume that the caller's range and the called domain spans the same numbers—say, 0 to 17. Figure 6.19 shows the four ways in which the caller's range closure and the called's domain closure can agree. I've turned the number line on its side. The thick line means closed and the thin line means open. Figure 6.19 shows the four cases consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom.

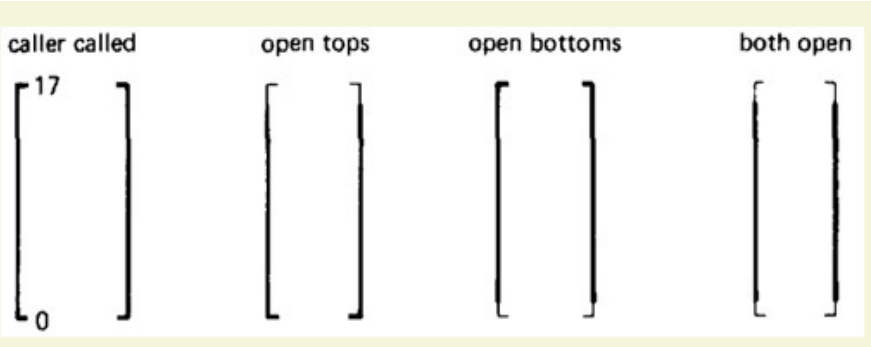


Figure 6.19. Range/Domain Closure Compatibility.

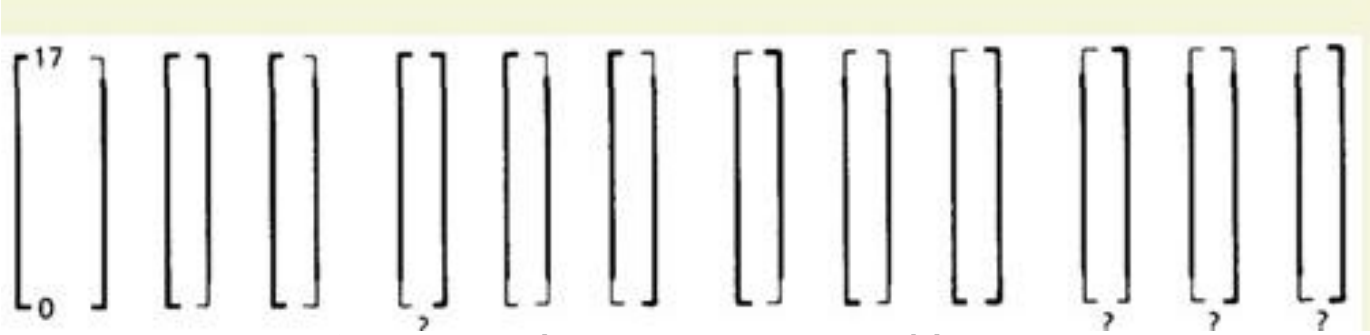


Figure 6.20. Equal-Span Range/Domain Compatibility Bugs.

Figure 6.20 shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs. The four cases in which a caller boundary is open and the called is closed (marked with a "?") are probably not buggy. It means that the caller will not supply such values but the called can accept them.

Span Compatibility

Figure 6.21 shows three possibly harmless span incompatibilities. I've eliminated closure incompatibilities to simplify things. In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug.

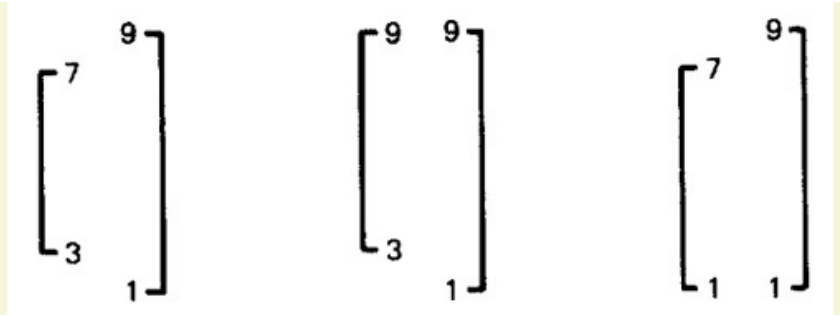


Figure 6.21. Harmless Range/Domain Span Incompatibility Bug. (Caller Span Is Smaller Than Called.)

Figure 6.22a shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples are buggy. In Figure 6.22b the ranges and domains don't line up; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes. Figure 6.22c combines these notions to show various ways we can have holes in the domain: these are all probably buggy.

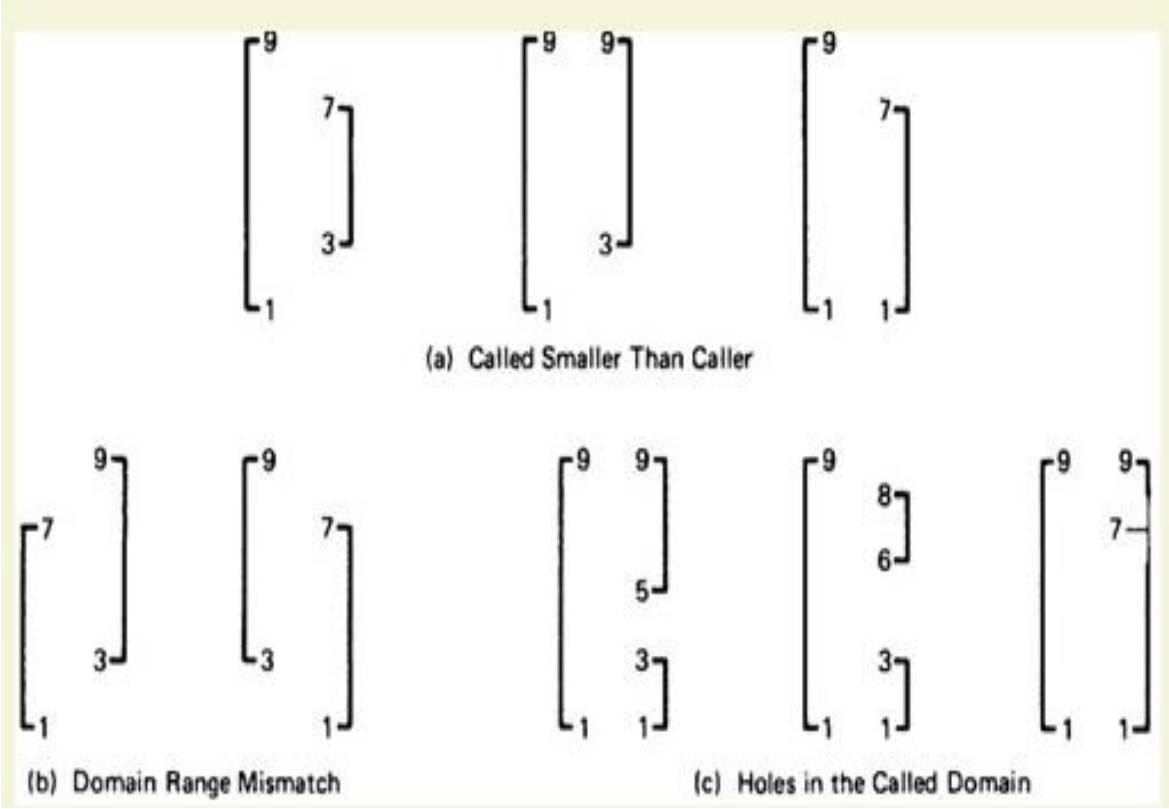


Figure 6.22. Buggy Range/Domain Mismatches.

Interface Range/Domain Compatibility Testing

To generalize caller/called range/domain incompatibilities by first considering combinations of closure and span errors for one variable. That led to a vast combination of cases that were no more revealing than those shown above. It curbed our ambition and tried to picture all possible range/domain disagreements for two variables simultaneously.

Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable. For subroutines that classify inputs as "valid/invalid" the called routine's domain span and closure (for valid cases) is usually broader than the caller's span and closure.

DOMAINS AND TESTABILITY

The best way to do domain testing is to avoid it by making things so simple that it isn't needed.

Linearizing Transformations

In the unlikely event that we're faced with essential nonlinearity we can often convert nonlinear boundaries to equivalent linear boundaries. This is done by applying **linearizing transformations**. The methods are centuries old. Here is a sample.

1. ***Polynomials*****—**A boundary is specified by a polynomial or multinomial in several variables. For a polynomial, each term (for example, x, x^2, x^3, . . .) can be replaced by a new variable: $y_1 = x$, $y_2 = x^2$, $y_3 = x^3$, ... For multinomials you add more new variables for terms such as xy, x^2y, xy^2, ... You're trading the nonlinearity of the polynomial for more dimensions. The advantage is that you transform the problem from one we can't solve to one for which there are many available methods (e.g., linear programming for finding the set of on and off points.)

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

3. More General Transforms**—**Other linearizable forms include $x/(ax + b)$ and ax^b. You can also linearize (approximately) by using the Taylor series expansion of nonlinear functions, which yields an infinite polynomial. Lop off the low-order terms to get an approximate polynomial and then linearize as for polynomials. There's a rich literature on even better methods for finding approximating polynomials for functions.

Coordinate Transformations

- Nice boundaries come in parallel sets. Parallel boundary sets are sets of linearly related boundaries: that is, they differ only by the value of a constant.
- Finding such sets is straightforward. Its an $O(n^2)$ procedure for n boundary equations. Pick a variable, say x. It has a coefficient a_i in inequality i.
- Divide each inequality by it's x coefficient so that the coefficients of the transformed set of inequalities is unity for variable x.
- If two inequalities are parallel, then all the coefficients will be the same and they will differ only by a constant.
- A systematic comparison will find the parallel sets, if any. There are more efficient algorithms, and any serious tool builder would do well to research the issue.
- We now have parallel sets of inequalities.

★★★★★★★

UNIT-IV

PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

path expression:

* path expression is defined as an algebraic representation of sets of paths in a graph.
* By using suitable arithmetic laws and weights, path expressions are converted into algebraic functions or regular expressions that can be used to examine structural properties of flow graphs, such as the number of paths, processing time or whether a data flow anomaly can occur.
* These expressions are then applied to problems in test design and debugging.

Overview:

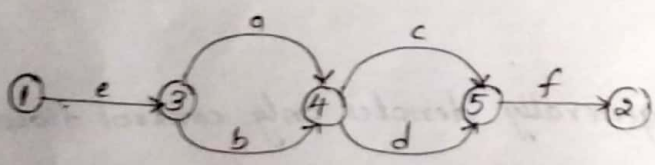
* Flowgraphs generally denotes only control flow, i.e links had no property other than they simply connect nodes.
* In Data-flow testing, we have added the concept of link weights i.e every link was associated with a letter.
* The simplest weight we can give to a link is a name.
* Using link names as weight, we then convert the

graphical flow graph into an equivalent algebraic like expression, which denotes the set of all possible paths from entry to exit for that flowgraph.

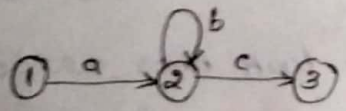
Basic Concepts :

* Every link of a graph can be given a name, the link name will be denoted by lowercase letters.
* Travelling a path includes travelling a succession of link names.
* If we traverse links a, b, c and d along some path, the name for that path segment is abcd.
* This path name is also called a path product.

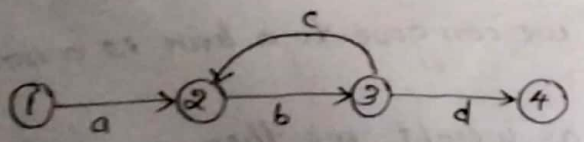
Examples :



In the above flow graph, the available paths are eacf, eadf, ebcf, ebdf



ac, abc, abbc, ...



path products :

The name of a path that consists of two successive path segments is expressed by the concatenation.

Example:

Let X and Y are defined as

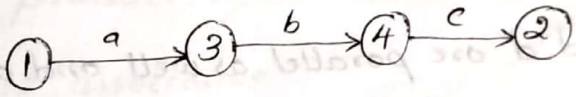
$X = abcde$

$Y = fghij$

then the path product of X and Y is

$XY = abcdefghij$

where X and Y represents set of paths.



path ~~expression~~ = abc

path product = abc

If a link or segment name is repeated, the fact is denoted by an exponent.

* The exponent's value denotes the no. of repetitions.

$a^1 = a$, $a^2 = aa$, $a^3 = aaa$, $a^n = aaa....$ n times

Example:

if $X = abcde$ then

$X^1 = abcde$

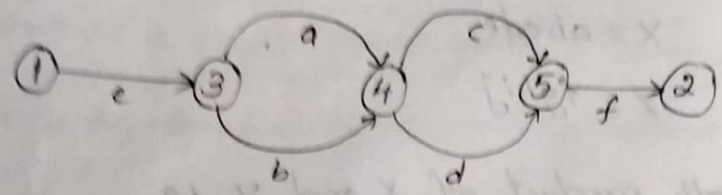
$X^2 = abcdeabcde = (abcde)^2$

$X^3 = abcdeabcdeabcde = (abcde)^3$.

Path Sum:

* The path sum denotes paths in parallel between two nodes.
* The '+' sign was used to denote the fact that path names were part of some set of paths.

Eg:



In the above flowgraph, the links a and b are parallel paths and are denoted by a+b.

Similarly, c and d are parallel as well and are denoted by c+d.

Determining available paths:

From the above flow graph, we have

$e(a+b)(c+d)f$

$\Rightarrow e(ac+ad+bc+bd)f$

$\Rightarrow eacf+eadf+ebcf+ebdf$ are all the set of available paths.

Loops:

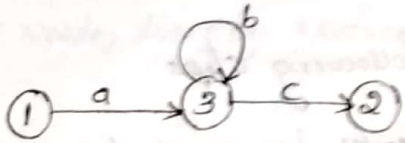
* Loops can be understood as an infinite set of parallel paths.

If a loop consist of a single bnt b then the set of all paths through that loop point is

$b^0 + b^1 + b^2 + b^3 + \cdots$

* The notation b^* is used to denoted infinite sum.

Example:



The path expressions for the above flow graph are $ab^*c = ac + abc + abbc + \cdots$

* The path product is not commutative (i.e $XY \neq YX$) but it is associative

Rule 1 : $A(BC) = (AB)C = ABC$

where A, B, C are path names.

* The path sum is both commutative and associative.

Rule 2 : $A + B = B + A$

Rule 3 : $A + (B + C) = (A + B) + C = A + B + C$

* The product and sum operations are distributive

Rule 4 : $A(B+C) = AB + AC$ and $(B+C)D = BD + CD$.

* Absorption rule says

Rule 5 : $A + A = A$

A REDUCTION PROCEDURE:

A reduction procedure is used for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph.

* It consist of the following steps

(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) Select any node for removal other than the initial or final node.

(5) Combine any remaining serial links by multiplying their path expressions.

(6) Combine all parallel links by adding their path expressions.

(7) Remove all self loops as in step-3.

(8) Does the graph consist of a single link between the entry node and the exit?
If yes, then the path expression for that link is a path expression for the original flowgraph otherwise return to step4.

Finally, the appearance of the

path expression depends on the order in which nodes are removed.

cross term step:

* It is the fundamental step of the reduction algorithm.
* It removes a node, thereby reducing the number of nodes by one.
* Successive applications of this step eventually get you down to one entry and one exit node.

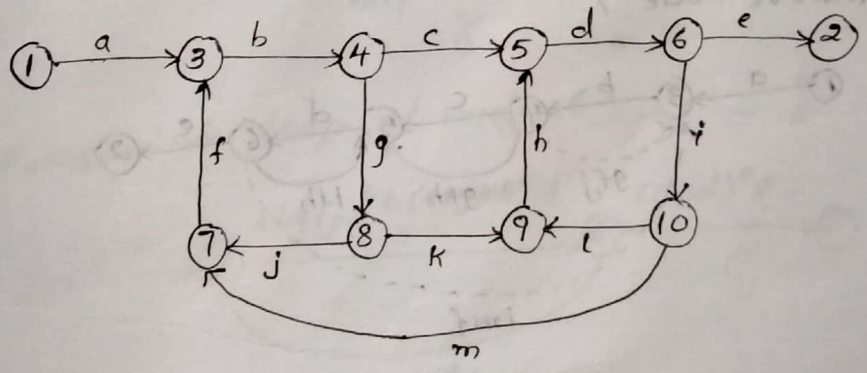
parallel term:

* It leads to addition of parallel links.

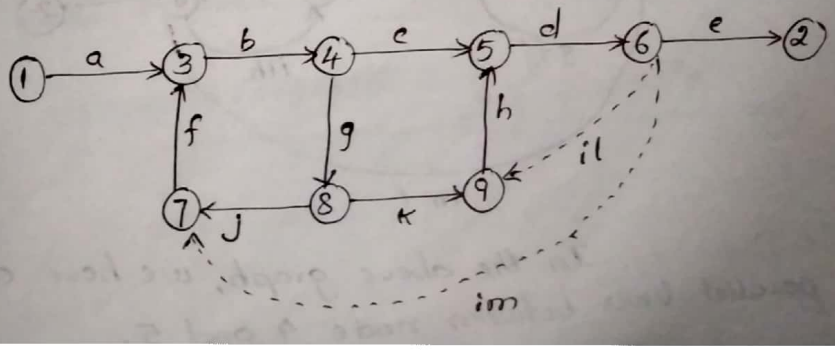
loop term:

* It leads to the removal of loops.

Example:

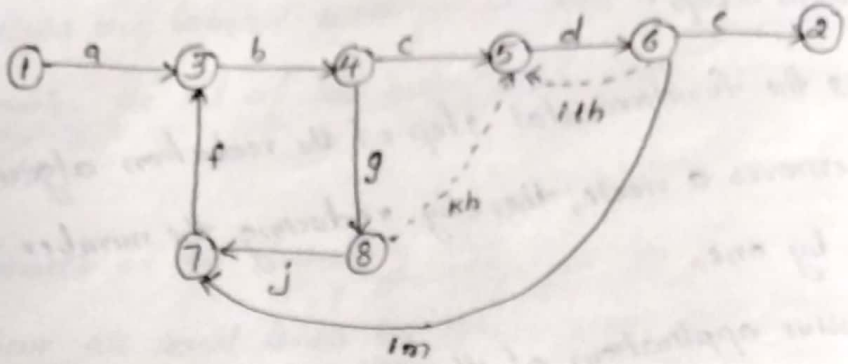


step-1 Let us first remove node 10



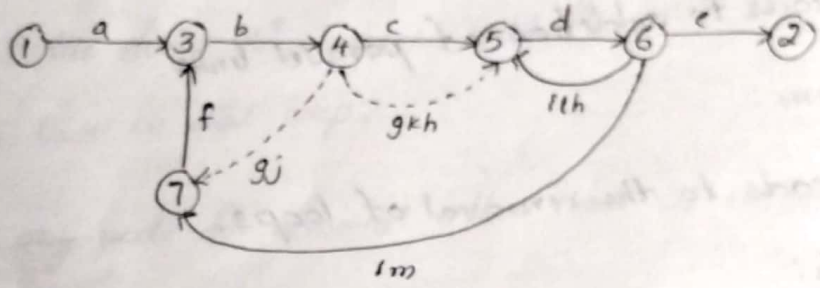
step-2:

Now Remove node 9

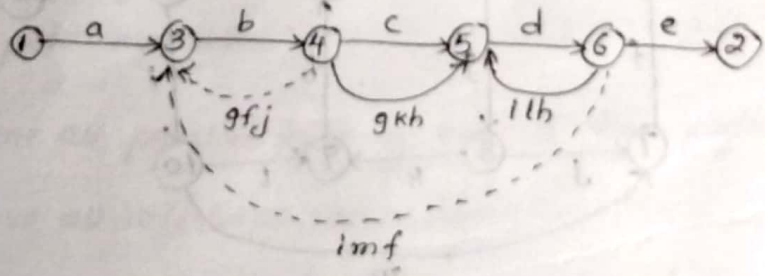


step-3:

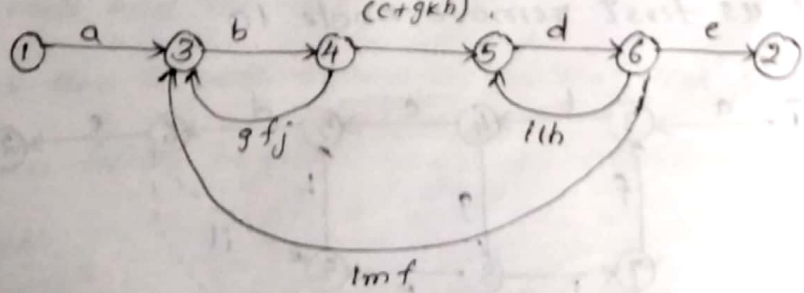
Remove node 8



step-4: Remove node 7



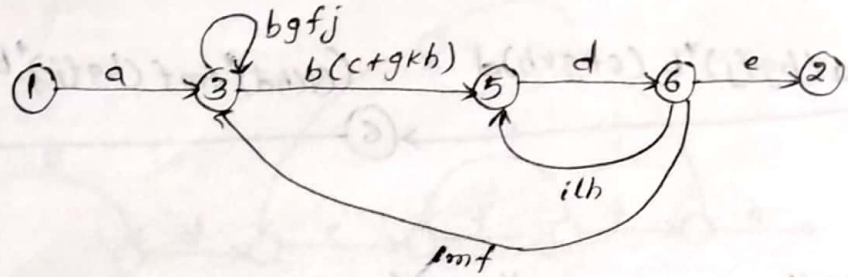
step-5:



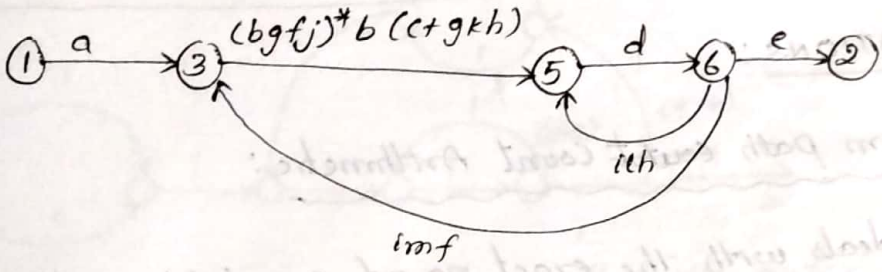
In the above graph, we have combined the paralled links between node 4 and 5.

step-6:

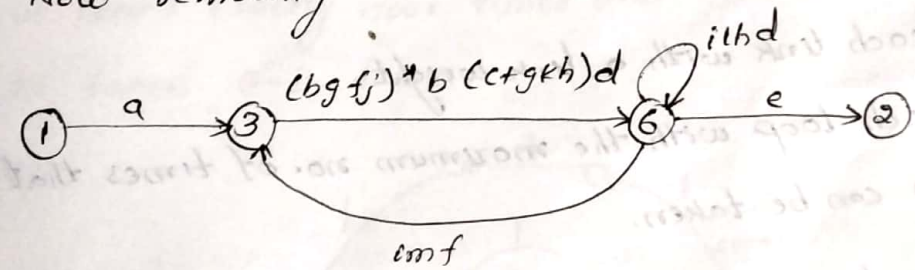
Now remove node 4



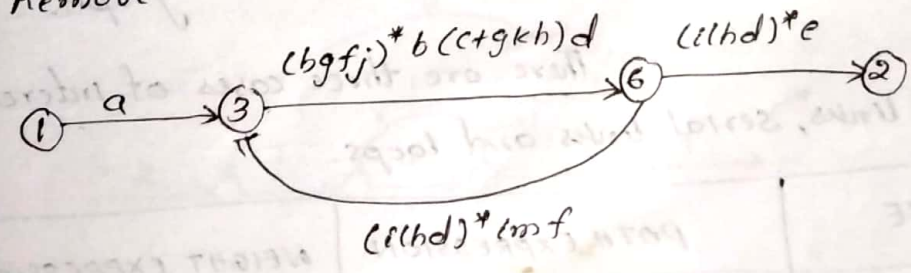
step-7: Remove the loop at node 3



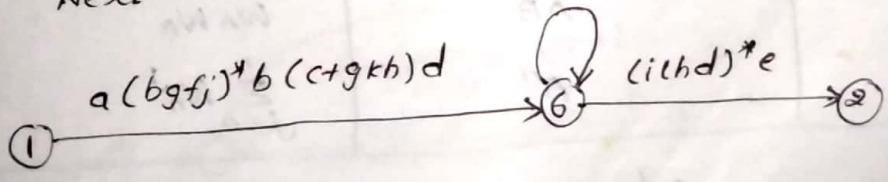
step-8: Now removing node 5



step-9: Remove the loop at node 6

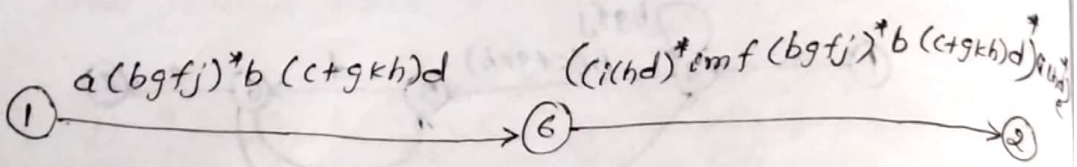


step-10: Next remove node 3



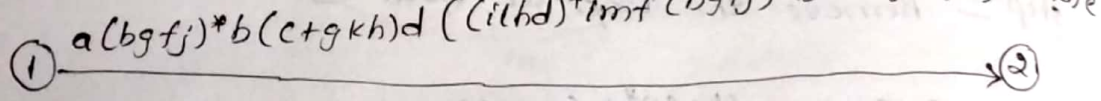
step-11:

Remove the loop at node 6



step-12:

Finally removing the node 6



APPLICATIONS:

Maximum path ~~count~~ Count Arithmetic:

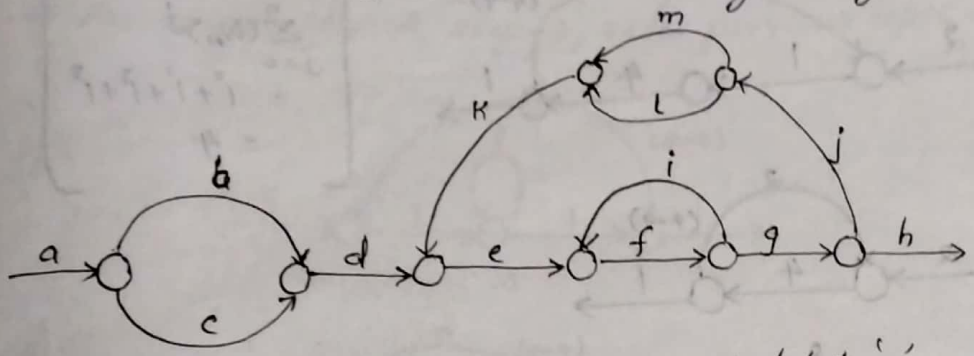
* It deals with the exact no. of available paths in a flowgraph.
* Label each link with a link weight.
* Mark each loop with the maximum no. of times that the loop can be taken.
* If the loop is infinite, then it will be clear that infinite no. of paths available in the flowgraph.

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

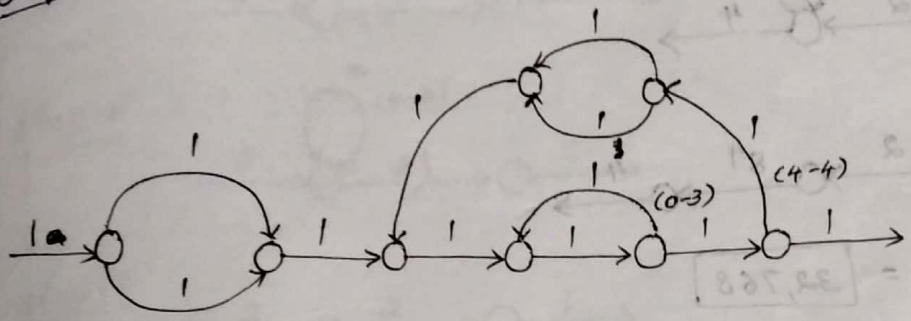
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
parallel	A+B	$W_A + W_B$
series	AB	$W_A W_B$
Loop	A^n	$\sum_{j=0}^{n} W_A^j$

where A and B are path expressions and W_A and W_B are algebraic representations in the weights.

: Consider the following flowgraph.

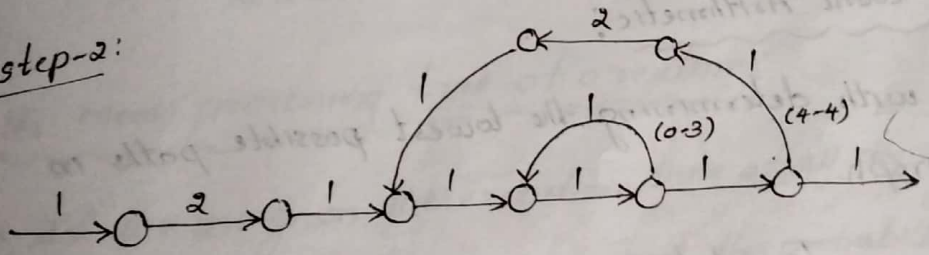


step-1: Let us now substitute 'i' in the above

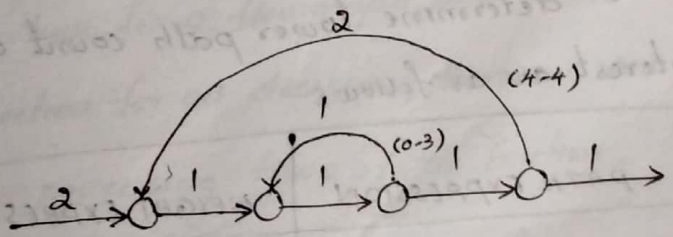


Also assume that the outer loop will be taken exactly four times and the inner loop can be taken 0-3 times.

step-2:

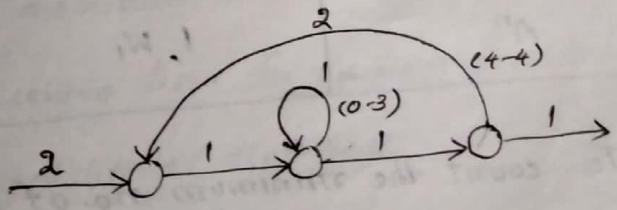


step-3:

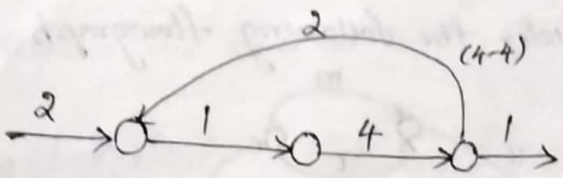


step-4:

For the inner loop



step-5:



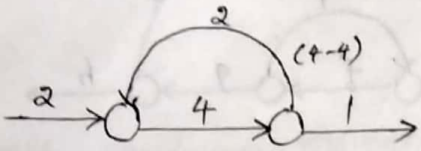
$\because j=0, n=3$

$\sum_{j=0}^{n} (W_A)^j$

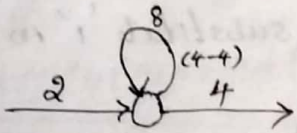
$= 1^0+1^1+1^2+1^3$

$= 4$

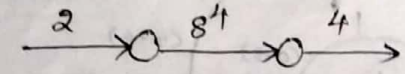
step-6:



step-7:



step-8:



$= \boxed{32,768}$

∴ The total or maximum no. of paths are 32,768.

And also the total no. of different paths are 8192.

Lower path count Arithmetic:

* It deals with determining the lowest possible paths in the flowgraph.

To determine lower path count arithmetic, the cases of interest are as follows

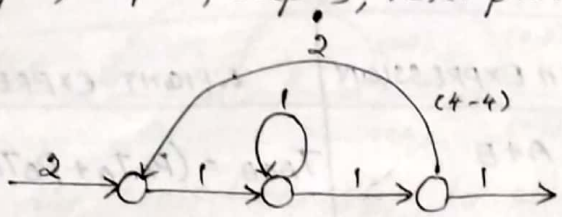
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
parallel	A+B	$W_A + W_B$
Series	AB	$MAX(W_A, W_B)$
Loop	A^n	$1, W_1$

To count the minimum no. of paths, it's more likely that the minimum is to be taken under

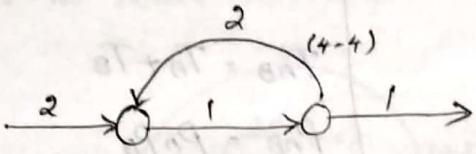
the assumption that the routine will loop once.

→ For step-1, step-2, step-3, refer previous topic - page No: 11

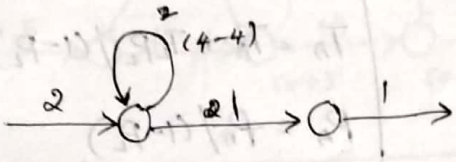
step-4:



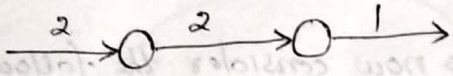
step-5:



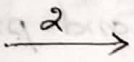
step-6:



step-7:



step-8:



∴ Lowest no. of paths = 2

The Mean processing time of a routine:

* If we are given the execution time of all the statements for every link in a flowgraph and the probability of each direction for all decisions, then we can calculate the mean processing time of the routine.

Our flowgraph consist of two weights associated with every link

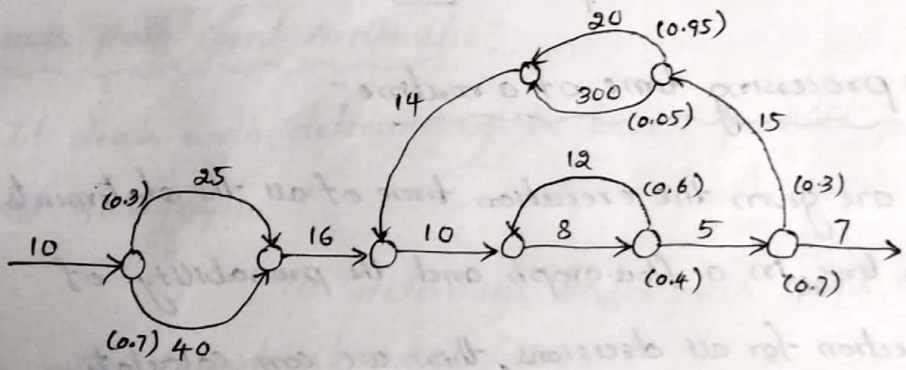
* The processing time for the link denoted by T.
* The probability of that link.

The rules for the mean processing times are

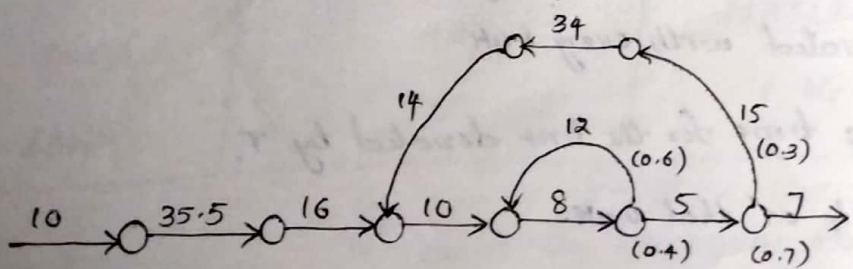
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
parallel	A+B	$T_{A+B} = (P_A T_A + P_B T_B)/(P_A + P_B)$ $P_{A+B} = P_A + P_B$
Series	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
Loop	A^*	$T_A = T_A + T_L P_L / (1 - P_L)$ $P_A = P_A / (1 - P_L)$

Example:

step-1: Let us now consider the following flowgraph annotated with probabilities and processing time in microseconds or in instructions.



step-2: Combine all the parallel links



eg:

$$T_{A+B} = \frac{(P_A T_A + P_B T_B)}{(P_A + P_B)}$$

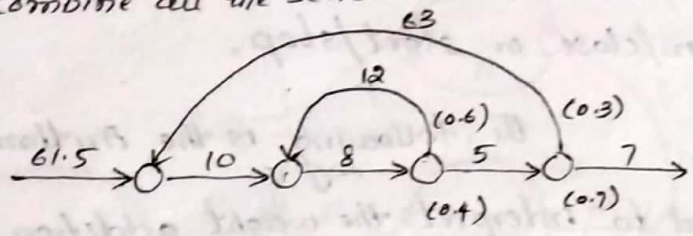
$$= \frac{(0.3 \times 25) + (0.7 \times 40)}{0.3 + 0.7}$$

$$= 35.5$$

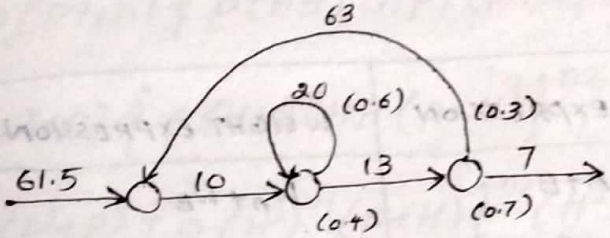
Since $P_A = 0.3$, $P_B = 0.7$

$T_A = 25$, $T_B = 40$

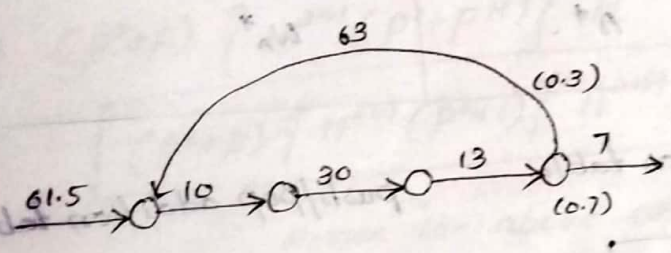
step-3: combine all the serial links



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

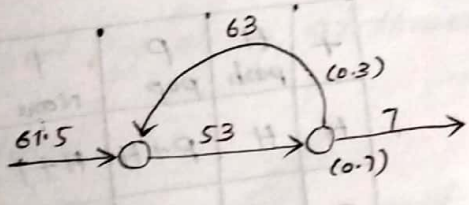


step-5:

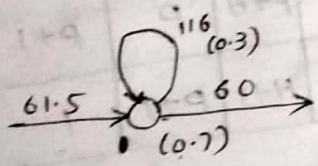


$$\frac{0.6}{0.4} \times 20 = 30$$

step-6:

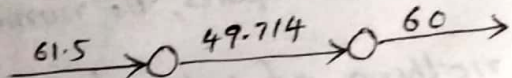


step-7:



$$\frac{0.3}{0.7} \times 116 = 49.714$$

step-8:



$= \boxed{171.24}$

push/pop, Get/Return:

- In this application, two complementary operations such as push and pop are taken.

Some more examples are

Get/Return, open/close or start/stop.

The following is the Arithmetic table which is needed to interpret the weight addition and multiplication operations. as show in fig① and fig②.

push/pop Arithmetic :

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
parallel	A+B	$W_A + W_B$
series	AB	$W_A W_B$
Loops	A^*	W_A^*

push/pop Multiplication table

×	H push	P pop	I None
H	H^2	I	H
P	I	P^2	P
I	H	P	I

push/pop addition table

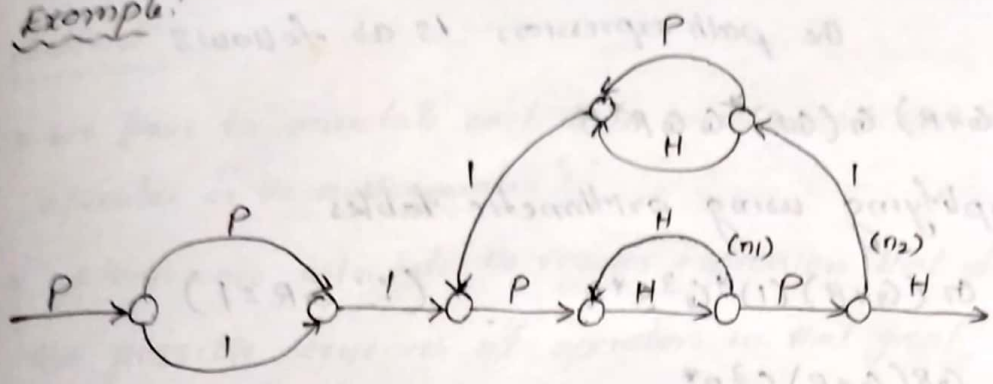
+	H push	P pop	I None
H	H	P+H	H+I
P	P+H	P	P+I
I	H+I	P+I	I

In the above tables, the numeral 'I' is used to indicate nothing of interest i.e neither push nor pop.

* H denotes PUSH
* P denotes pop.

All the operations are commutative, associative and distributive.

Example:



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

$$\Rightarrow (P^2+P)\{P(HH)^{n1}.HP(P+H)\}^{n2}PHPH(HH)^{n1}$$

$$\Rightarrow (P^2+P)\{P(H^2)^{n1}(P+H)\}^{n2}(H^2)^{n1} \quad [\because PH=1]$$

$$\Rightarrow (P^2+P)\{H^{2n1}(P^2+PH)\}^{n2}H^{2n1}$$

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

Hence the above expression is simplified by using arithmetic tables.

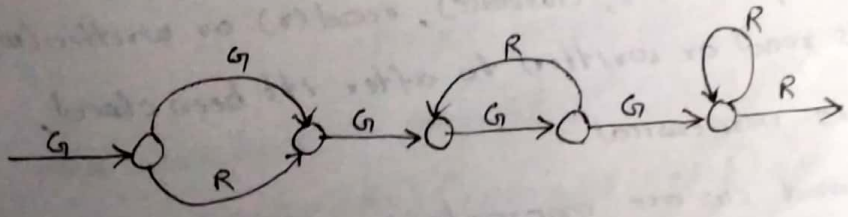
Get/Return:

* We can use the same arithmetic tables for Get/Return.

The arithmetic tables are as follows.

X	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

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



The path expression is as follows

$G(G+R)\,G(GR)^*\,G\,G\,R^*R$

Now simplifying using arithmetic tables

$G(G+R)(1)^*G^3R^*R \quad (\because GR=1)$

$\Rightarrow GR(G+R)G^3R^*$

$\Rightarrow 1(G+R)G^3R^*$

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

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

$\boxed{(G^4+G^2)R^*}$

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

* In generic flow anomaly detection, we will look for a specific sequence of operations by considering all possible paths through a routine.
* Let's say that 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)
* In this concept, we are interested in knowing whether a specific sequence occured, not what the net effect of the routine is.

Example:

* A file can be opened(o), closed(c), read(r) or written(w).
* If the file is read or written to after it's been closed, the sequence is nonsensical.
* Therefore cr and cw are anomalous.

Method:

* We have to annotate each link in the graph with the appropriate operator or the null operator 'ε'.
* We can now calculate the regular expression that denotes all the possible sequences of operators in that graph.
* Finally we can examine the regular expression for the sequences of interest.

Huang's Theorem:

* Let A, B, C be non empty 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.

Consider an example, let

$$A = PP$$
$$B = SR$$
$$C = RP$$
$$T = SS$$

The theorem states that SS will appear in $PP(SR)^nRP$ if it appears in $PP(SR)^2RP$.

Now let $A = P + PP + PS$

$$B = PSR + PS(R + PS)$$
$$C = RP$$
$$T = P^4$$

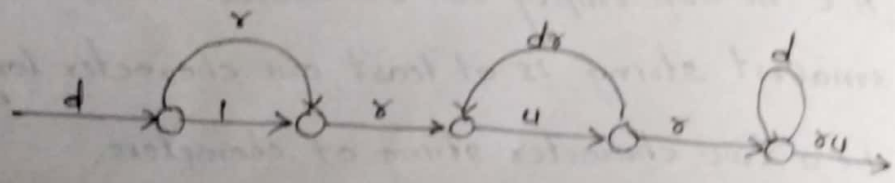
Now we have to look at

$$(P + PP + PS)\left[(PSR + PS(R + PS)\right]^2 RP$$

* Another point of Huang's theorem is that if you substitute $1+x^2$ for every expression of the form x^n, the paths that result from this substitution are sufficient to determine whether a given two character sequence exists or not.

Data flow testing example:

Consider the following flowgraph annotated with operators at the links that corresponds to the variables of interest.



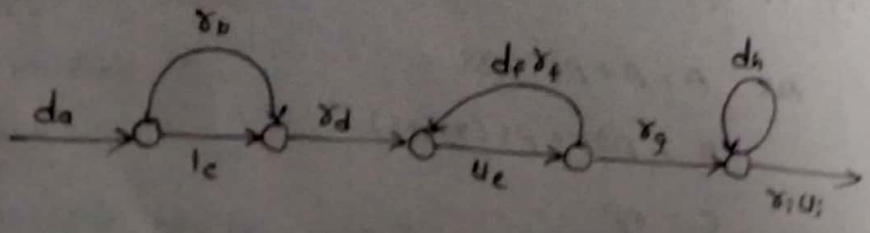
Simplifying...

$$d(\gamma+1).\gamma(ud\gamma)^*u\gamma d^*\gamma u$$

$$d(\gamma+1)\gamma(1+(ud\gamma)^2)u\gamma(1+d^2)\gamma u \quad (\because x^* = 1+x^2)$$

$$(d\gamma\gamma+d\gamma)(1+ud\gamma ud\gamma)(u\gamma\gamma u+u\gamma d^2\gamma u)$$

Another way to do this is to subscript the operator with the link name, so that we will not loose the path information.



The path expression is

$$a(b+c)d(ef)^*egh^*;$$

The regular expression is

$$d_a(\gamma_b + l_c)\gamma_d(u_e d_f \gamma_f)^* u_e \gamma_g d_h^* \gamma_i u_i$$

Applying Huang's theorem

$$d_a(\gamma_b + l_c)\gamma_d\left(\varepsilon l + (u_e d_f \gamma_f)^2\right) u_e \gamma_g (1 + d_h^2)\gamma_i u_i$$

— Best of Luck —

LOGIC-BASED TESTING

The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design. Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**.

OVERVIEW

Programmers and Logic

"Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques. This chapter concerns logic in its simplest form, **boolean algebra**, and its application to program and specification test and design. Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.

Hardware Logic Testing

Logic has been, for several decades, the primary tool of hardware logic designers. Today, hardware logic design and more important in the context of this book, hardware logic *test* design, are intensely automated. Many test methods developed for hardware logic can be adapted to software logic testing.
Hardware testing is ahead of software testing not because hardware testers are smarter than software testers but because hardware testing is easier.

Specification Systems and Languages

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

Knowledge-Based Systems

The **knowledge-based system** (also **expert system**, or **"artificial intelligence" system**) has become the programming construct of choice for many applications that were once considered very difficult. Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain. One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data. The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**. From the point of view of testing, there's nothing special about the inference engine—it's just another piece of software to be tested, and the methods discussed in this book apply. When we talk about testing knowledge-based systems, it's not the inference engine that concerns us, but testing the validity of the expert's knowledge and the correctness of the transcription (i.e., coding) of that knowledge into a rule base.

Overview

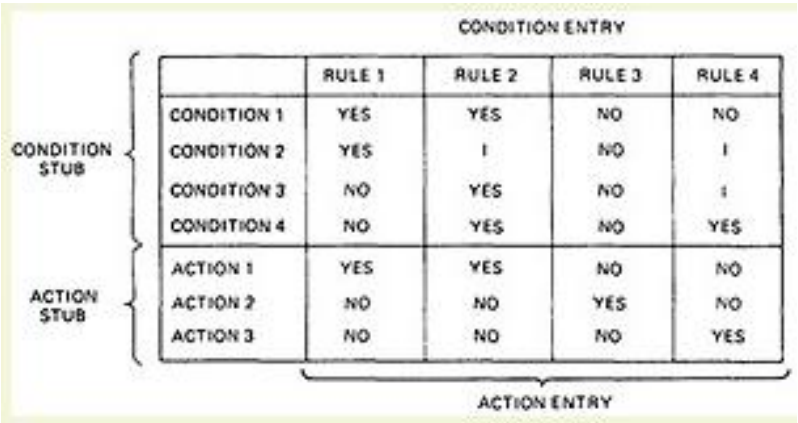
We start with **decision tables** because: they are extensively used in business data processing; decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors. The next step is a review of boolean algebra (included to make this book self-contained). I included decision tables because the engineering/mathematically trained programmer may not be familiar with them, and boolean algebra because the business data processing programmer may have had only cursory exposure to it. That gets both kinds of readers to a common base, which is the use of decision tables and/or boolean algebra in test and software design.

Although *programmed tools* are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right *conceptual tool:* the **Karnaugh-Veitch** diagram is that conceptual tool. Few programmers, unless they're retreaded logic designers like me, or unless they had a fling with hardware design, learn about this method of doing boolean algebra. Without it, boolean algebra is tedious and error-prone and I don't wonder that people won't use it. With it, with practice, boolean algebra is no worse than arithmetic.

DECISION TABLES

Definitions and Notation

The below figure is a **limited-entry decision table**. It consists of four areas called the **condition stub**, the **condition entry**, the **action stub**, and the **action entry**. Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place. **The condition stub** is a list of names of conditions. A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is **immaterial** to that rule. The **action stub** names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES," the action will take place; if "NO," the action will not take place. Table 10.1 can be translated as follows:



Decision-Table Processors

Decision tables can be automatically translated into code and, as such, are a higher-order language. The decision table's translator checks the source decision table for consistency and completeness and fills in any required default rules. The usual processing order in the resulting object code is, first, to examine rule 1. If the rule is satisfied, the corresponding action takes place. Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken. Decision tables as a source language have the virtue of clarity, direct

correspondence to specifications, and maintainability. The principal deficiency is possible object-code inefficiency. There was a time when it was thought by some that decision tables would herald a new era in programming. Their use, it was claimed, would eliminate most bugs and poor programming practices and would reduce testing to trivia. Such claims are rarely made now, but despite such unrealistically high hopes, decision tables have become entrenched as a useful tool in the programmer's kit, especially in business data processing.

Decision Tables as a Basis for Test Case Design

If a specification is given as a decision table, it follows that decision tables should be used for test case design. Similarly, if a program's logic is to be implemented as decision tables, decision tables should also be used as a basis for test design. But if that's so, the consistency and completeness of the decision table is checked by the decision-table processor; therefore, it would seem that there would be no need to design those test cases.

Expansion of Immaterial Cases

Improperly specified **immaterial entries** (**I**) cause most decision-table contradictions. If a condition's truth value is immaterial in a rule, satisfying the rule does not depend on the condition. It doesn't mean that the case is impossible. For example,

Rule 1: "If the persons are male and over 30, then they shall receive a 15% raise."

Rule 2: "But if the persons are female, then they shall receive a 10% raise."

The above rules state that age is material for a male's raise, but immaterial for determining a female's raise. No one would suggest that females either under or over 30 are impossible. If there are *n* predicates there are 2^n cases to consider. You find the cases by expanding the immaterial cases. This is done by converting each I entry into a pair of entries, one with a YES and the other with a NO. Each I entry in a rule doubles the number of cases in the expansion of that rule.

PATH EXPRESSIONS

Logic-based testing is structural testing when it's applied to structure (e.g., control flowgraph of an implementation); it's functional testing when it's applied to a specification. As with all test techniques, we start with a model: we focus on one program characteristic and ignore the others. In logic-based testing we focus on the truth values of control flow predicates.

Predicates and Relational Operators

A **predicate** is implemented as a process whose outcome is a truth-functional value. Don't restrict your notion of predicate to arithmetic relations such as >, >=, =, <, <, ≠. Predicates are based on **relational operators**, of which the arithmetic relational operators are merely the most common. Here's a sample of some other relational operators: ... is a member of ..., ... is a subset of ..., ... is a substring of ..., ... is a subgraph of ..., ... dominates ..., ... is dominated by ..., ... is the greatest lower bound of ..., ... hides ..., ... is in the shadow of ..., ... is above ..., ... is below .... The point about thinking of predicates as processes that yield truth values is that it usually pays to look at predicates top-down—typically from the point of view of predicates as specified in requirements rather than from the point of view of predicates as implemented. Almost all programming languages have arithmetic relational operators and few others. Therefore, in most languages you construct the predicates you need by using the more primitive arithmetic relations. For example, you need a set membership

predicate for numbers (e.g., . . . is a member of . . .) but it's implemented as an equality predicate in a loop that scans the whole set.

What Goes Wrong with Predicates

Several things can go wrong with predicates, especially if the predicate has to be interpreted in order to express it as a predicate over input values.

1. The wrong relational operator is used: e.g., > instead of <=.
2. The predicate expression of a compound predicate is incorrect: e.g., A + B instead of AB.
3. The wrong operands are used: e.g., A > X instead of A > Z.
4. The processing leading to the predicate (along the predicate's interpretation path) is faulty.

Logic-based testing is useful against the first two bug types, whereas data flow testing is more useful against the latter two.

Boolean Algebra

The Rules of Boolean Algebra

Boolean algebra has three operators:

RULES

	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
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

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

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

several literals is called a **product term** (e.g., ABC, DE). Usually, product terms are simplified by removing duplicate appearances of the same literal barred or unbarred. For example, AAB and are replaced by AB and respectively. Also, any product term that has both a barred and unbarred appearance of the same literal is removed because it equals zero by rule 10.

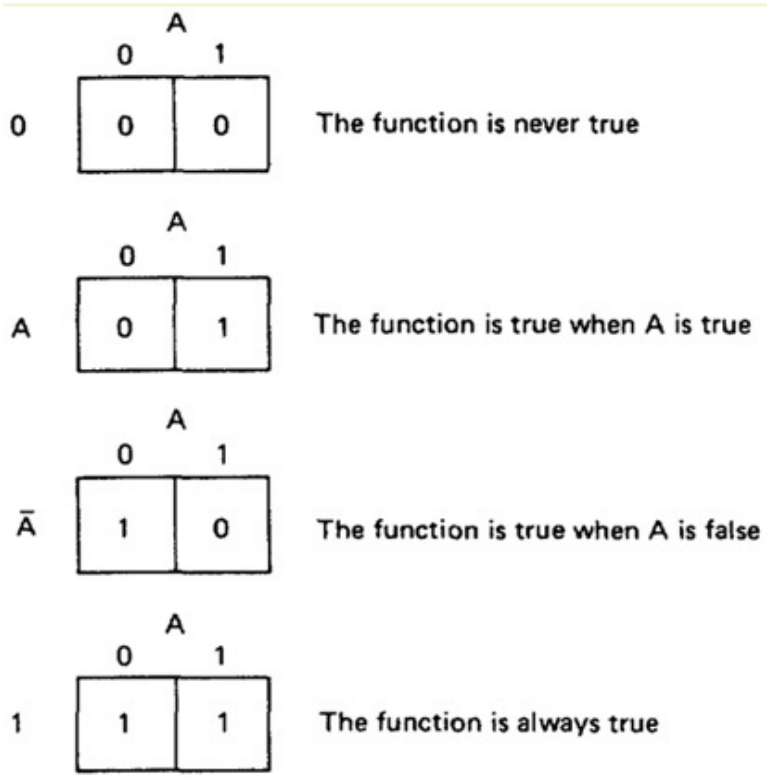
KV CHARTS

The Problem

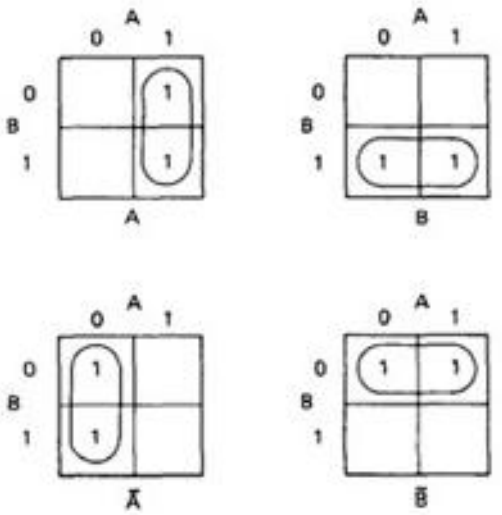
It's okay to slug through boolean algebra expressions to determine which cases are interesting and which combination of predicate values should be used to reach which node; it's okay, but not necessary. If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing. The **Karnaugh-Veitch chart** (this is known by every combination of "Karnaugh" and/or "Veitch" with any one of "map," "chart," or "diagram") reduces boolean algebraic manipulations to graphical trivia Beyond six variables these diagrams get cumbersome, and other techniques such as the Quine-McCluskey method (which are beyond the scope of this book) should be used.

Simple Forms

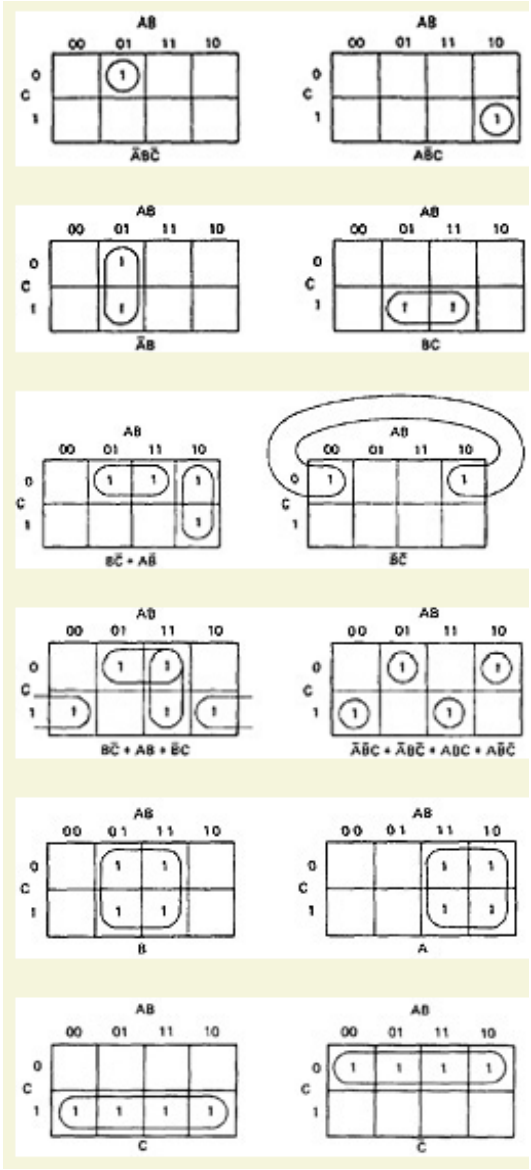
The below figure shows all the boolean functions of a single variable and their equivalent representation as a KV chart. The charts show all possible truth values that the variable A can have. The heading above each box in the chart denotes this fact. A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE. The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable. We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.



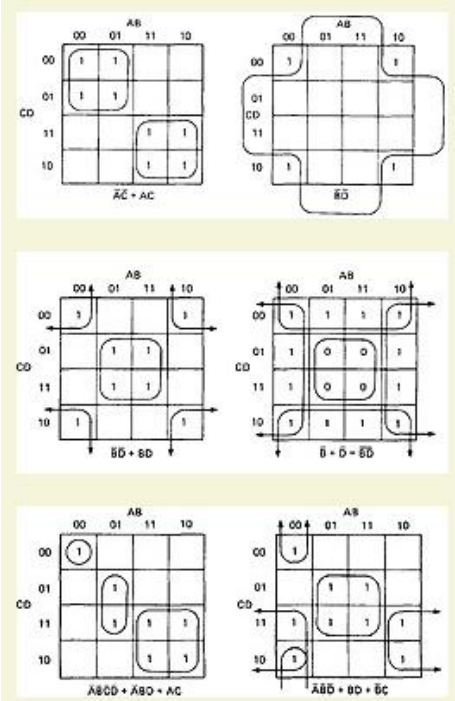
Two Variables



Three Variables



Four Variables and More



TESTABILITY TIPS

Logic-intensive software designed by the seat of the pants is almost never right. We learned this lesson decades ago in the simpler hardware logic design arena. It is in our interest as software engineers to use the simplest possible predicate expressions in our design. The objective is not to simplify the code in order to save a few bytes of memory but to reduce the opportunities for bugs. Hardware logic designers learned that there were many advantages to designing their logic in a **canonical form**—that is, a form that followed certain rules.

1. Identify your predicates (simple or compound).
2. If starting from code, get a branch covering set of path predicates.
3. Interpret the predicates so that they are expressed in terms of the input vector for the chosen path.
4. Simplify the path predicate expression for each selected path. If any expression is logically zero, the path is unachievable. Pick another path or paths to achieve branch coverage.
5. If any path predicate expression equals logical I then all other paths must be unachievable—find and fix the design bug.
6. The logical sum of the path predicate expressions must equal 1 or else there is an unsuspected loop, dangling code, or branch coverage is an inadequate test criterion.

The canonical processor has three successive stages:

1. Predicate calculator.
2. Logic analyzer.
3. Domain processor.

★★★★★★★

UNIT- V

STATES, STATE GRAPHS, AND TRANSITION TESTING

The **state graph** and its associated **state table** are useful models for describing software behavior. The **finite-state machine** is a functional testing tool and testable design programming tool. Methods analogous to path testing are described and discussed.

OVERVIEW

The **finite-state machine** is as fundamental to software engineering as boolean algebra. State testing strategies are based on the use of finite-state machine models for software structure, software behavior, or specifications of software behavior. Finite-state machines can also be implemented as table-driven software, in which case they are a powerful design option. Independent testers are likeliest to use a finite-state machine model as a guide to the design of functional tests—especially system tests. Software designers are likelier to want to exploit and test finite-state machine software implementations.

STATE GRAPHS

States

The word **"state"** is used in much the same way it's used in ordinary English, as in "state of the union," or "state of health." The Oxford English Dictionary defines "state" as: "A combination of circumstances or attributes belonging for the time being to a person or thing."

A moving automobile whose engine is running can have the following states with respect to its transmission:

1. Reverse gear
2. Neutral gear
3. First gear
4. Second gear
5. Third gear
6. Fourth gear

A person's checkbook can have the following states with respect to the bank balance:

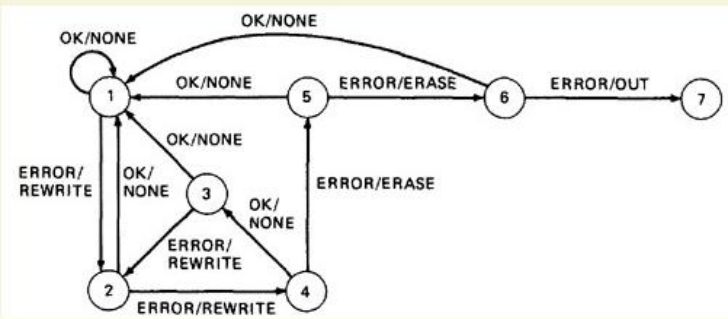
1. Equal
2. Less than
3. Greater than

Inputs and Transitions

Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a **transition.** Transitions are denoted by links that join the states. The input that causes the transition are marked on the link; that is, the inputs are link weights. There is one outlink from every state for every input. If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: "input1, input2, input3. . .". A **finite-state machine** is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

Outputs

An output* can be associated with any link. Outputs are denoted by letters or words and are separated from inputs by a slash as follows: "input/output." As always, "output" denotes anything of interest that's observable and is not restricted to explicit outputs by devices. Outputs are also link weights. If every input associated with a transition causes the same output, then denote it as: "input 1, input 2, . . . input 3/output." If there are many different combinations of inputs and outputs, it's best to draw a separate parallel link for each output.



Tape Control Recovery Routine State Graph.

State Tables

Big state graphs are cluttered and hard to follow. It's more convenient to represent the state graph as a table (the **state table** or **state-transition table**) that specifies the states, the inputs, the transitions, and the outputs. The following conventions are used:

	INPUT	
STATE	**OKAY**	**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	. . .	. . .

State Table

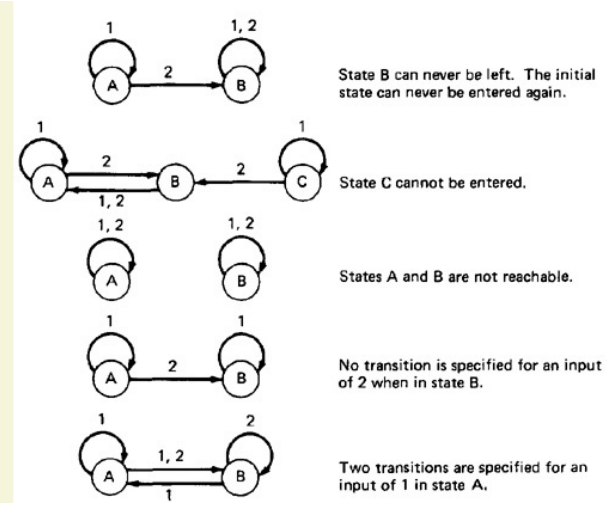
1. Each row of the table corresponds to a state.
2. Each column corresponds to an input condition.
3. The box at the intersection of a row and column specifies the next state (the transition) and the output, if any.

GOOD STATE GRAPHS AND BAD

This is a book on testing so we deal not just with good state graphs, but also with bad ones. What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context. Here are some principles for judging:

1. The total number of states is equal to the product of the possibilities of factors that make up the state.
2. For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
3. For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.*

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



Improper State Graphs

State Bugs

Number of States

The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base. As an example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9,

Impossible States

Some combinations of factors may appear to be impossible. Say that the factors are:

GEAR	R, N, 1, 2, 3, 4	= 6 factors
DIRECTION	Forward, reverse, stopped	= 3 factors
ENGINE	Running, stopped	= 2 factors
TRANSMISSION	Okay, broken	= 2 factors
ENGINE	Okay, broken	= 2 factors
TOTAL		= 144 states

But broken engines can't run, so the combination of factors for engine condition and engine operation yields only 3 rather than 4 states. Therefore, the total number of states is at most 108. A car with a broken transmission won't move for long, thereby further decreasing the number of feasible states. The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states."

Equivalent States

Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to sets of states. Figure 11.4 shows the situation.

Say that the system is in state S and that an input of *a* causes a transition to state A while an input of *b* causes a transition to state B. The blobs indicate portions of the state graph whose details are unimportant. If, starting from state A, *every* possible sequence of inputs produces *exactly* the same sequence of outputs that would occur when starting from state B, then there is no way that an outside observer can determine which of the two sets of states the system is in without looking at the record of the state. The state graph can be reduced to that of Figure 11.5 without harm.

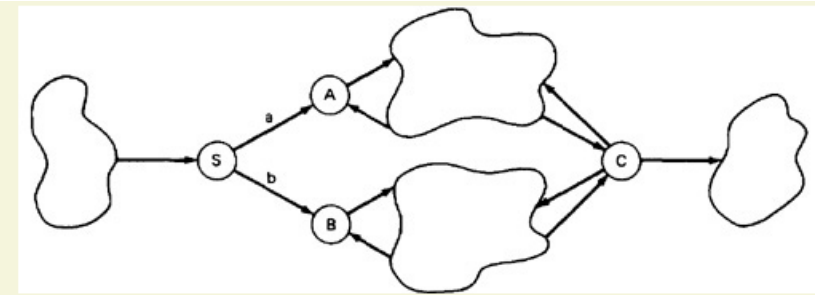


Figure 11.4. Equivalent States.

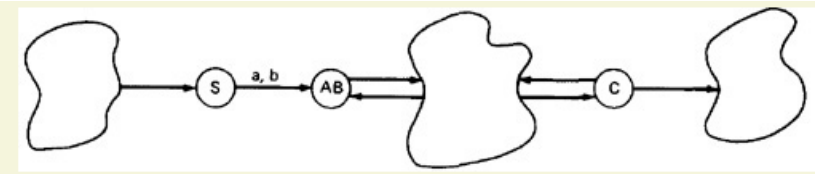


Figure 11.5. Equivalent States of Figure 11.4 Merged.

Transition Bugs

Unspecified and Contradictory Transitions

Every input-state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state—look for it. If there is no such mechanism, what will the program do if, through a malfunction or an alpha particle, the impossible input occurs in that state? The transition for a given state-input combination may not be specified because of an oversight. *Exactly one transition must be specified for every combination of input and state.* However you model it or test it, the system will do *something* for every combination of input and state. It's better that it does what you want it to do, which you assure by specifying a transition rather than what some bugs want it to do.

An Example

Specifications are one of the most common source of ambiguities and contradictions. Specifications, unlike programs, can be full of ambiguities and contradictions. The following example illustrates how to convert a specification into a state graph and how contradictions can come about. The tape control routine will be used. Start with the first statement in the specification and add to the state graph one statement at a time. Here is the first statement of the specification:

Unreachable States

An **unreachable state** is like unreachable code—a state that no input sequence can reach. An unreachable state is not impossible, just as unreachable code is not impossible. Furthermore, there may be transitions from the unreachable state to other states; there usually are because the state became unreachable as a result of incorrect transitions.

Dead States

A **dead state**, (or set of dead states) is a state that once entered cannot be left. This is not necessarily a bug, but it is suspicious. If the software was designed to be the fuse for a bomb, we would expect at least one such state. A set of states may appear to be dead because the program has two modes of operation. In the first mode it goes through an initialization process that consists of several states. Once initialized, it goes to a strongly connected set of working states, which, within the context of the routine, cannot be exited. The initialization states are unreachable to the working states, and the working states are dead to the initialization states. The only way to get back might be after a system crash and restart. Legitimate dead states are rare. They occur mainly with system-level issues and device handlers. In normal software, if it's not possible to get from any state to any other, there's reason for concern.

Output Errors

The states, the transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect. Output actions must be verified independently of states and transitions. That is, you should distinguish between a program whose state graph is correct but has the wrong output for a transition and one whose state graph is incorrect. The likeliest reason for an incorrect output is an incorrect call to the routine that executes the output. This is usually a localized and minor bug. Bugs in the state graph are more serious because they tend to be related to fundamental control-structure problems. If the routine is implemented as a state table, both types of bugs are comparably severe.

Encoding Bugs

It would seem that encoding bugs for input coding, output coding, state codes, and state-symbol product formation could exist as such only in an explicit finite-state machine implementation. The possibility of such bugs is obvious for a finite-state machine implementation, but the bugs can also occur when the finite-state machine is implicit. If the programmer has a notion of state and has built an implicit finite-state machine, say by using a bunch of program flags, switches, and "condition" or "status" words, there may be an encoding process in place.

STATE TESTING

Impact of Bugs

Let's say that a routine is specified as a state graph that has been verified as correct in all details. Program code or tables or a combination of both must still be implemented. A bug can manifest itself as one or more of the following symptoms:

1. Wrong number of states.
2. Wrong transition for a given state-input combination.
3. Wrong output for a given transition.
4. Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
5. States or sets of states that are split to create inequivalent duplicates.
6. States or sets of states that have become dead.
7. States or sets of states that have become unreachable.

Principles

The strategy for state testing is analogous to that used for path-testing flowgraphs. Just as it's impractical to go through every possible path in a flowgraph, it's impractical to go through every path in a state graph. A path in a state graph, of course, is a succession of transitions caused by a sequence of inputs. The notion of coverage is identical to that used for flowgraphs—pass through each link (i.e., each transition must be exercised). Assume that some state is especially interesting—call it the initial state. Because most realistic state graphs are strongly connected, it should be possible to go through all states and back to the initial state, when starting from there.

The starting point of state testing is:

1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.

A set of tests, then, consists of three sets of sequences:

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

TESTABILITY TIPS

A Balm for Programmers

Most of this chapter has taken the independent tester's viewpoint and has been a prescription for making programmers squirm. What is testability but means by which programmers can protect themselves from the ravages of sinister independent testers? What is testability but a guide to cheating—how to design software so that the pesticide paradox works and the tester's strongest technique is made ineffectual? The key to testability design is easy: build explicit finite-state machines.

How Big, How Small?

I understand every two-state finite-state machine because, including the good and bad ones, there are only eight of them. There are about eighty possible good and bad three-state machines, 2700 four-state machines, 275,000 five-state machines, and close to 100 million six-state machines, most of which are bad. We learned long ago, as hardware logic designers, that it paid to build explicit finite-state machines for even very small machines. I think you can safely get away with two states, it's getting difficult for three states, a heroic act for four, and beyond human comprehension for five states. That doesn't mean that you have to build your finite-state machine as in the explicit PDL example given above, but that you must do a finite-state machine model and identify how you're implementing every part of that model for anything with four or more states.

GRAPH MATRICES AND APPLICATIONS

Graph matrices are introduced as another representation for graphs; some useful tools resulting there from are examined. Matrix operations, relations, node-reduction algorithm revisited, equivalence class partitions.

MOTIVATIONAL OVERVIEW

The Problem with Pictorial Graphs

Graphs were introduced as an abstraction of software structure early in this book and used throughout. Yet another graph that modeled software behavior was introduced in earlier. There are many other kinds of graphs, not discussed in this book, that are useful in software testing. Whenever a graph is used as a model, sooner or later we trace paths through it—to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether a state is reachable or not. Even algebraic representations such as BNF and regular expressions can be converted to equivalent graphs. Much of test design consists of tracing paths through a graph and most testing strategies define some kind of cover over some kind of graph.

Path tracing is not easy, and it's subject to error. You can miss a link here and there or cover some links twice—even if you do use a marking pen to note which paths have been taken. You're tracing a long complicated path through a routine when the telephone rings—you've lost your place before you've had a chance to mark it. I get confused tracing paths, so naturally I assume that other people also get confused.

One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods aren't necessarily easier than path tracing, but because they're more methodical and mechanical and don't depend on your ability to "see" a path, they're more reliable.

Even if you use powerful tools that do everything that can be done with graphs, and furthermore, enable you to do it graphically, it's still a good idea to know how to do it by hand; just as having a calculator should not mean that you don't need to know how to do arithmetic. Besides, with a little practice, you might find these methods easier and faster than doing it on the screen; moreover, you can use them on the plane or anywhere.

Tool Building

If you build test tools or want to know how they work, sooner or later you'll be implementing or investigating analysis routines based on these methods—or you should be. Think about how a naive tool builder would go about finding a property of all paths (a possibly infinite number) versus how one might do it based on the methods which was graphical and it's hard to build algorithms over visual graphs. The properties of graph matrices are fundamental to test tool building.

THE MATRIX OF A GRAPH

Basic Principles

A **graph matrix** is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column. The relation, for example, could be as simple as the link name, if there is a link between the nodes. Some examples of graphs and their associated matrices are shown in Figure 12. 1 a through g. Observe the following:

1. The size of the matrix (i.e., the number of rows and columns) equals the number of nodes.

2. There is a place to put every possible direct connection or link between any node and any other node.

3. The entry at a row and column intersection is the link weight of the link (if any) that connects the two nodes in that direction.

4. A connection from node *i* to node *j* does not imply a connection from node *j* to node *i*. Note that in Figure 12.1h the (5,6) entry is *m*, but the (6,5) entry is *c*.

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

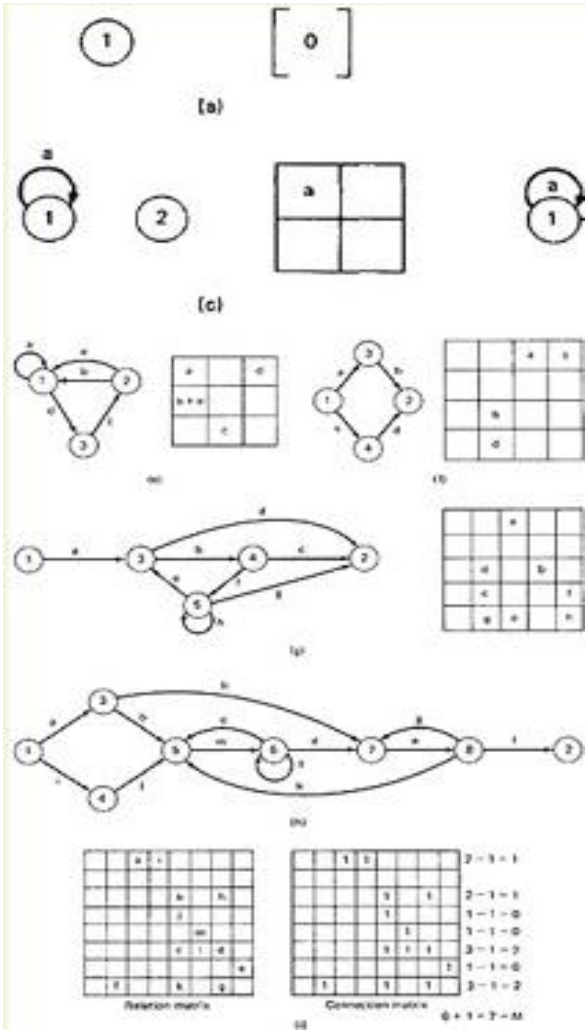


Figure 12.1. Some Graphs and Their Matrices.

A Simple Weight

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

$$1 + 1 = 1, \qquad 1 + 0 = 1, \qquad 0 + 0 = 0,$$
$$1 \times 1 = 1, \qquad 1 \times 0 = 0, \qquad 0 \times 0 = 0.$$

A matrix with weights defined like this is called a **connection matrix**.

RELATIONS

General

This isn't a section on aunts and uncles but on abstract relations that can exist between abstract objects, although family and personal relations can also be modeled by abstract relations, if you want to. A **relation** is a property that exists between two (usually) objects of interest. We've had many examples of relations in this book. Here's a sample, where *a* and *b* denote objects and R is used to denote that a has the relation R to *b*:

1. "Node *a is connected* to node *b*" or *a*R*b* where "R" means "is connected to."
2. "*a* >= *b*" or *a*R*b* where "R" means "greater than or equal."
3. "*a is a subset* of *b*" where the relation is "is a subset of."
4. "It takes 20 microseconds of processing time to get from node *a* to node *b*." The relation is expressed by the number 20.
5. "Data object X is defined at program node *a* and used at program node *b*." The relation between nodes *a* and *b* is that there is a *du* chain between them.

Let's now redefine what we mean by a graph.

A **graph** consists of a set of abstract objects called **nodes** and a relation R between the nodes. If *a*R*b*, which is to say that a has the relation R to *b*, it is denoted by a **link** from *a* to *b*. In addition to the fact that the relation exists, for some relations we can associate one or more properties. These are called **link weights**. A link weight can be numerical, logical, illogical, objective, subjective, or whatever. Furthermore, there is no limit to the number and type of link weights that one may associate with a relation.

"Is connected to" is just about the simplest relation there is: it is denoted by an unweighted link. Graphs defined over "is connected to" are called, as we said before, **connection matrices**.* For more general relations, the matrix is called a **relation matrix**.

Properties of Relations

General

The least that we can ask of relations is that there be an algorithm by which we can determine whether or not the relation exists between two nodes. If that's all we ask, then our relation arithmetic is too weak to be useful. The following sections concern some properties of relations that have been found to be useful. Any given relation may or may not have these properties, in almost any combination.

Transitive Relations

A relation R is **transitive** if *a*R*b* and *b*R*c* implies *a*R*c*. Most relations used in testing are transitive. Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of. Examples of **intransitive** relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

Reflexive Relations

A relation R is **reflexive** if, for every *a*, *a*R*a*. A reflexive relation is equivalent to a self-loop at every node. Examples of reflexive relations include: equals, is acquainted with (except, perhaps, for amnesiacs), is a relative of. Examples of **irreflexive relations** include: not equals, is a friend of (unfortunately), is on top of, is under.

Symmetric Relations

A relation R is **symmetric** if for every *a* and *b*, *a*R*b* implies *b*R*a*. A symmetric relation means that if there is a link from *a* to *b* then there is also a link from *b* to *a*; which furthermore means that we can do away with arrows and replace the pair of links with a single **undirected** link. A graph whose relations are not symmetric is called a **directed graph** because we must use arrows to denote the relation's direction. A graph over a symmetric relation is called an **undirected graph**.* The matrix of an undirected graph is symmetric ($a_{ij} = a_{ji}$ for all i, j).

Examples of symmetric relations: is a relative of, equals, is alongside of, shares a room with, is married (usually), is brother of, is similar (in most uses of the word), OR, AND, EXOR. Examples of **asymmetric** relations: is the boss of, is the husband of, is greater than, controls, dominates, can be reached from.

Antisymmetric Relations

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

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time. Examples of **nonantisymmetric** relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of.

Equivalence Relations

An **equivalence relation** is a relation that satisfies the reflexive, transitive, and symmetric properties. Numerical equality is the most familiar example of an equivalence relation. If a set of objects satisfy an equivalence relation, we say that they form an **equivalence class** over that relation. The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class. The idea behind **partition-testing strategies** such as domain testing and path testing, is that we can partition the input space into equivalence classes.

Partial Ordering Relations

A **partial ordering relation** satisfies the reflexive, transitive, and antisymmetric properties. Partial ordered graphs have several important properties: they are loop-free, there is at least one maximum element, there is at least one minimum element, and if you reverse all the arrows, the resulting graph is

also partly ordered. A **maximum element** *a* is one for which the relation *x*R*a* does not hold for any other element *x*. Similarly, a **minimum element** *a*, is one for which the relation *a*R*x* does not hold for any other element *x*. Trees are good examples of partial ordering. The importance of partial ordering is that while strict ordering (as for numbers) is rare with graphs, partial ordering is common. Loop-free graphs are partly ordered.

THE POWERS OF A MATRIX

Principles

Each entry in the graph's matrix (that is, each link) expresses a relation between the pair of nodes that corresponds to that entry. It is a direct relation, but we are usually interested in indirect relations that exist by virtue of intervening nodes between the two nodes of interest. Squaring the matrix (using suitable arithmetic for the weights) yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive. The square of the matrix represents all path segments two links long. Similarly, the third power represents all path segments three links long. And the *k*th power of the matrix represents all path segments *k* links long. Because a matrix has at most *n* nodes, and no path can be more than $n - 1$ links long without incorporating some path segment already accounted for, it is generally not necessary to go beyond the $n - 1$ power of the matrix. As usual, concatenation of links or the weights of links is represented by multiplication, and parallel links or path expressions by addition.

Matrix Powers and Products

Given a matrix whose entries are a_{ij}, the square of that matrix is obtained by replacing every entry with

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

More generally, given two matrices A and B, with entries a_{ik} and b_{kj}, respectively, their product is a new matrix C, whose entries are c_{ij}, where:

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

$$\begin{bmatrix} a_{11}a_{12}a_{13}a_{14} \\ a_{21}a_{22}a_{23}a_{24} \\ a_{31}a_{32}a_{33}a_{34} \\ a_{41}a_{42}a_{43}a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11}b_{12}b_{13}b_{14} \\ b_{21}b_{22}b_{23}b_{24} \\ b_{31}b_{32}b_{33}b_{34} \\ b_{41}b_{42}b_{43}b_{44} \end{bmatrix} = \begin{bmatrix} c_{11}c_{12}c_{13}c_{14} \\ c_{21}c_{22}c_{23}c_{24} \\ c_{31}c_{32}c_{33}c_{34} \\ c_{41}c_{42}c_{43}c_{44} \end{bmatrix}$$

$$\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \\
c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} \\
c_{13} &= a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43} \\
&\ldots \\
&\ldots \\
&\ldots \\
c_{32} &= a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42} \\
&\ldots \\
&\ldots \\
&\ldots \\
c_{44} &= a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}
\end{aligned}$$

Partitioning Algorithm

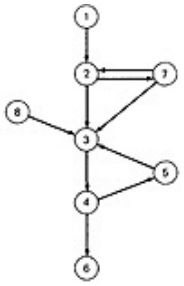
Consider any graph over a transitive relation. The graph may have loops. We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another. Such a graph is partly ordered. There are many used for an algorithm that does that:

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

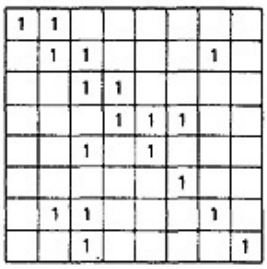
2. Many graphs with loops are easy to analyze if you know where to break the loops.

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

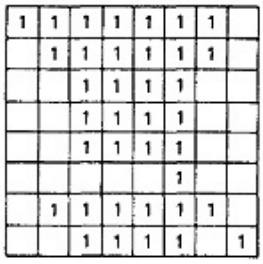
The way to do this is straightforward. Calculate the following matrix: $(A + I)^n \# (A + I)^{nT}$. This groups the nodes into strongly connected sets of nodes such that the sets are partly ordered. Furthermore, every such set is an equivalence class so that any one node in it represents the set. Now consider all the places in this book where we said "except for graphs with loops" or "assume a loop-free graph" or words to that effect. If you can bury the loop in a real subroutine, you can as easily bury it in a conceptual subroutine. Do the analysis over the partly ordered graph obtained by the partitioning algorithm and treat each loop-connected node set as if it is a subroutine to be examined in detail later. For each such component, break the loop and repeat the process. You now have a divide-and-conquer approach for handling loops. Here's an example, worked with an arbitrary graph:



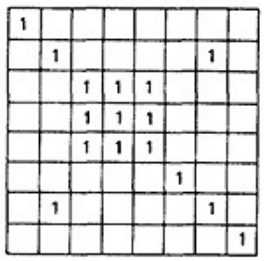
The relation matrix is



The transitive closure matrix is



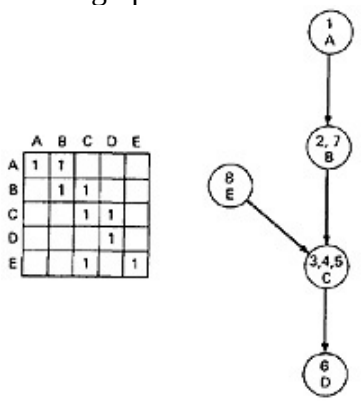
Intersection with its transpose yields



You can recognize equivalent nodes by simply picking a row (or column) and searching the matrix for identical rows. Mark the nodes that match the pattern as you go and eliminate that row. Then start again from the top with another row and another pattern. Eventually, all rows have been grouped. The algorithm leads to the following equivalent node sets:

A = [1]
B = [2,7]
C = [3,4,5]
D = [6]
E = [8]

whose graph is



NODE-REDUCTION ALGORITHM

General

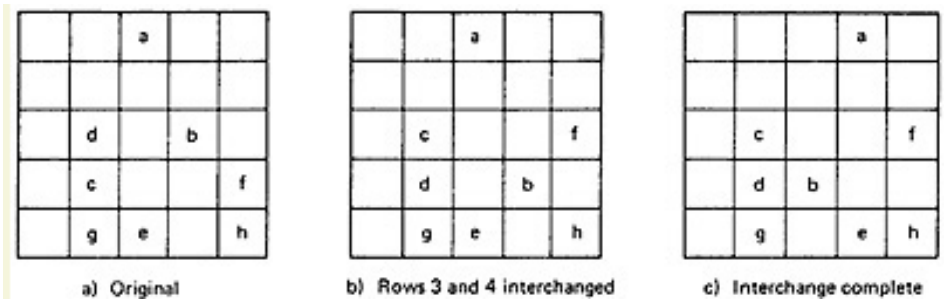
The matrix powers usually tell us more than we want to know about most graphs. In the context of testing, we're usually interested in establishing a relation between two nodes — typically the entry and exit nodes—rather than between every node and every other node. In a debugging context it is unlikely that we would want to know the path expression between every node and every other node; there also, it is the path expression or some other related expression between a specific pair of nodes that is sought: for example, "How did I get *here* from *there*?" The method of this section is a matrix equivalence to the node-by-node reduction procedure of Unit-IV. The advantage of the matrix-reduction method is that it is more methodical than the graphical method of Unit-IV and does not entail continually redrawing the graph. It's done as follows:

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

Some Matrix Properties

If you numbered the nodes of a graph from 1 to *n*, you would not expect that the behavior of the graph or the program that it represents would change if you happened to number the nodes differently. Node numbering is arbitrary and cannot affect anything. The equivalent to renumbering the nodes of a graph is to interchange the rows and columns of the corresponding matrix. Say that you

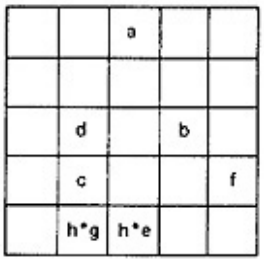
wanted to change the names of nodes *i* and *j* to *j* and *i*, respectively. You would do this on the graph by erasing the names and rewriting them. To interchange node names in the matrix, you must interchange both the corresponding rows and the corresponding columns. Interchanging the names of nodes 3 and 4 in the graph of Figure 12.1g results in the following:



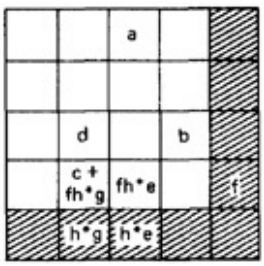
If you redraw the graph based on c, you will see that it is identical to the original except that node 3's name has been changed to 4, and node 4's name to 3.

The Algorithm

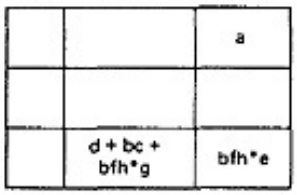
The first step is the most complicated one: eliminating a node and replacing it with a set of equivalent links. Using the example of Figure 12.1g, we must first remove the self-loop at node 5. This produces the following matrix:



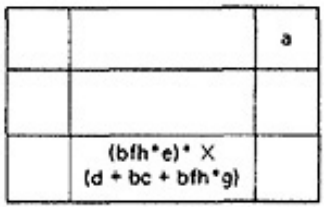
The reduction is done one node at a time by combining the elements in the last column with the elements in the last row and putting the result into the entry at the corresponding intersection. In the above case, the *f* in column 5 is first combined with *h*g* in column 2, and the result (*fh*g*) is added to the *c* term just above it. Similarly, the *f* is combined with *h*e* in column 3 and put into the 4,3 entry just above it. The justification for this operation is that the column entry specifies the links entering the node, whereas the row specifies the links leaving the node.



If any loop terms had occurred at this point, they would have been taken care of by eliminating the loop term and premultiplying every term in that row by the loop term starred. There are no loop terms at this point. The next node to be removed is node 4.



Removing the loop term yields



There is only one node to remove now, node 3. This will result in a term in the (1,2) entry whose value is

$$a(bfh^*e)^*(d + bc + bfh^*g)$$

BUILDING TOOLS

Matrix Representation Software

Overview

We draw graphs or display them on screens as visual objects; we prove theorems and develop graph algorithms by using matrices; and when we want to process graphs in a computer, because we're building tools, we represent them as linked lists. We use linked lists because graph matrices are usually very sparse; that is, the rows and columns are mostly empty.

Node Degree and Graph Density

The **out-degree** of a node is the number of outlinks it has. The **in-degree** of a node is the number of inlinks it has. The **degree** of a node is the sum of the out-degree and in-degree. The average degree of a node (the mean over all nodes) for a typical graph defined over software is between 3 and 4. The degree of a simple branch is 3, as is the degree of a simple junction. The degree of a loop, if wholly contained in one statement, is only 4. A mean node degree of 5 or 6 say, would be a very busy flowgraph indeed.

What's 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:

1. *Space*—Space grows as n^2 for the matrix representation, but for a linked list only as kn, where k is a small number such as 3 or 4.

2. *Weights*—Most weights are complicated and can have several components. That would require an additional weight matrix for each such weight.

3. *Variable-Length Weights*—If the weights are regular expressions, say, or algebraic expressions (which is what we need for a timing analyzer), then we need a two-dimensional string array, most of whose entries would be null.

4. *Processing Time*—Even though operations over null entries are fast, it still takes time to access such entries and discard them. The matrix representation forces us to spend a lot of time processing combinations of entries that we know will yield null results.

Linked-List Representation

Give every node a unique name or number. A link is a pair of node names. The linked list for Figure in UNIT-IV is shown below:

1,3;*a*
2,
3,2;*d*
3,4;*b*
4,2;*c*
4,5;*f*
5,2;*g*
5,3;*e*
5,5;*h*

★★★★★★★