STM STEP

1) State and explain various dichotomies in S/W testing.(CO1)

Testing Versus Debugging: Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

Debugging usually follows testing, but they differ as to goals, methods and most important psychology. The below tab le shows few important differences between testing and debugging.

Testing	Debugging
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown intial conditions and the end can not be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intutive leaps, experimentation and freedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

Function Versus Structure: Tests can be designed from a functional or a structural point of view. In **functional testing**, the program or system is treated as a blackbox. It is subjected to inputs, and its outputs are verified for conformance to specified behaviour. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation.**Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.

Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tets can detect all bugs but would take infinite time

to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.

Designer Versus Tester: Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person.

Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.

Modularity Versus Efficiency: A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organised into modular components. Small, independent test cases can be designed to test independent modules.

Small Versus Large: Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.

Builder Versus Buyer: Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.

The different roles / users in a system include:

1. **Builder:** Who designs the system and is accountable to the buyer.
2. **Buyer:** Who pays for the system in the hope of profits from providing services.
3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
4. **Tester:** Who is dedicated to the builder's destruction.
5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints.

2) Discuss path testing criteria?(CO5)

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

1 **Path:**a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

2 A path may go through several junctions, processes, or decisions, one or more times.

3 Paths consists of segments.

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

5 A path segment is succession of consecutive links that belongs to some path.

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

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

2 **FUNDAMENTAL PATH SELECTION CRITERIA:**

1 There are many paths between the entry and exit of a typical routine.

2 Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

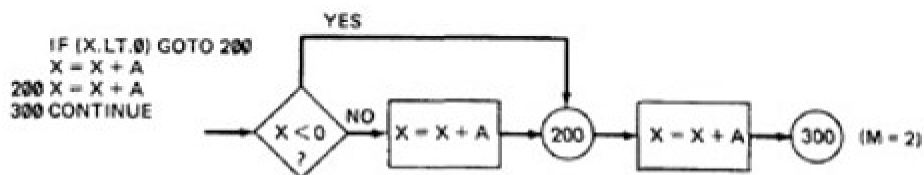
3 Defining complete testing:

1. Exercise every path from entry to exit
2. Exercise every statement or instruction at least once
3. Exercise every branch and case statement, in each direction at least once

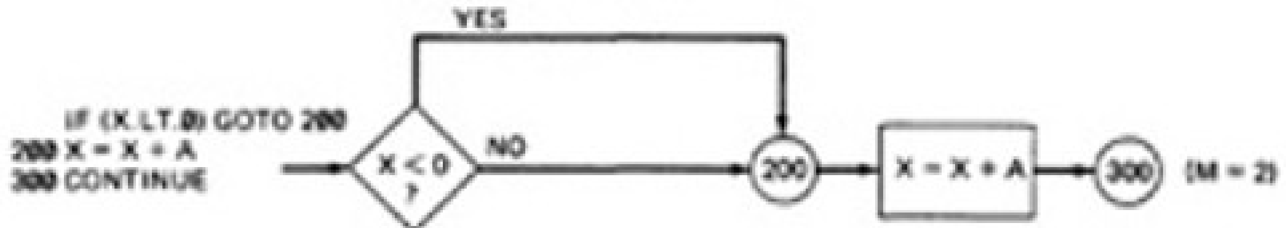
4 If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

5

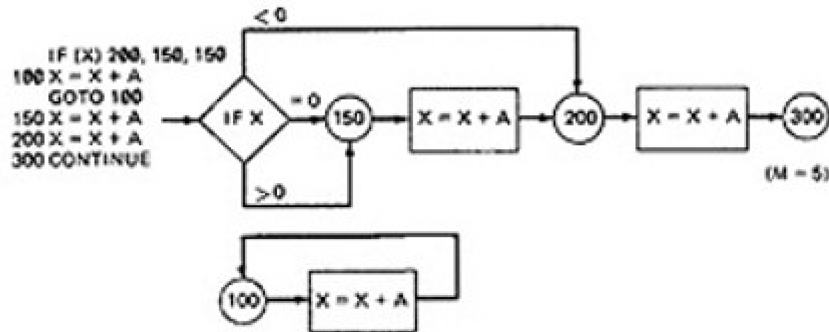
6 ***EXAMPLE:*****Here is the correct version.**



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



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



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

7 A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

8 Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

3 **PATH TESTING CRITERIA:**

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

2 A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

3 So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

1. **Path Testing (P_{inf})**:
   - Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
   - If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.
2. **Statement Testing (P_1)**:
   - Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
   - An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
   - This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or can not be accepted) and should be criminalized.
3. **Branch Testing (P_2)**:
   - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   - If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
   - An alternative characterization is to say that we have achieved 100% link coverage.
   - For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
   - We denote branch coverage by C2.

2 **Commonsense and Strategies:**

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.
- **Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help

automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

- **Path Selection Example:**

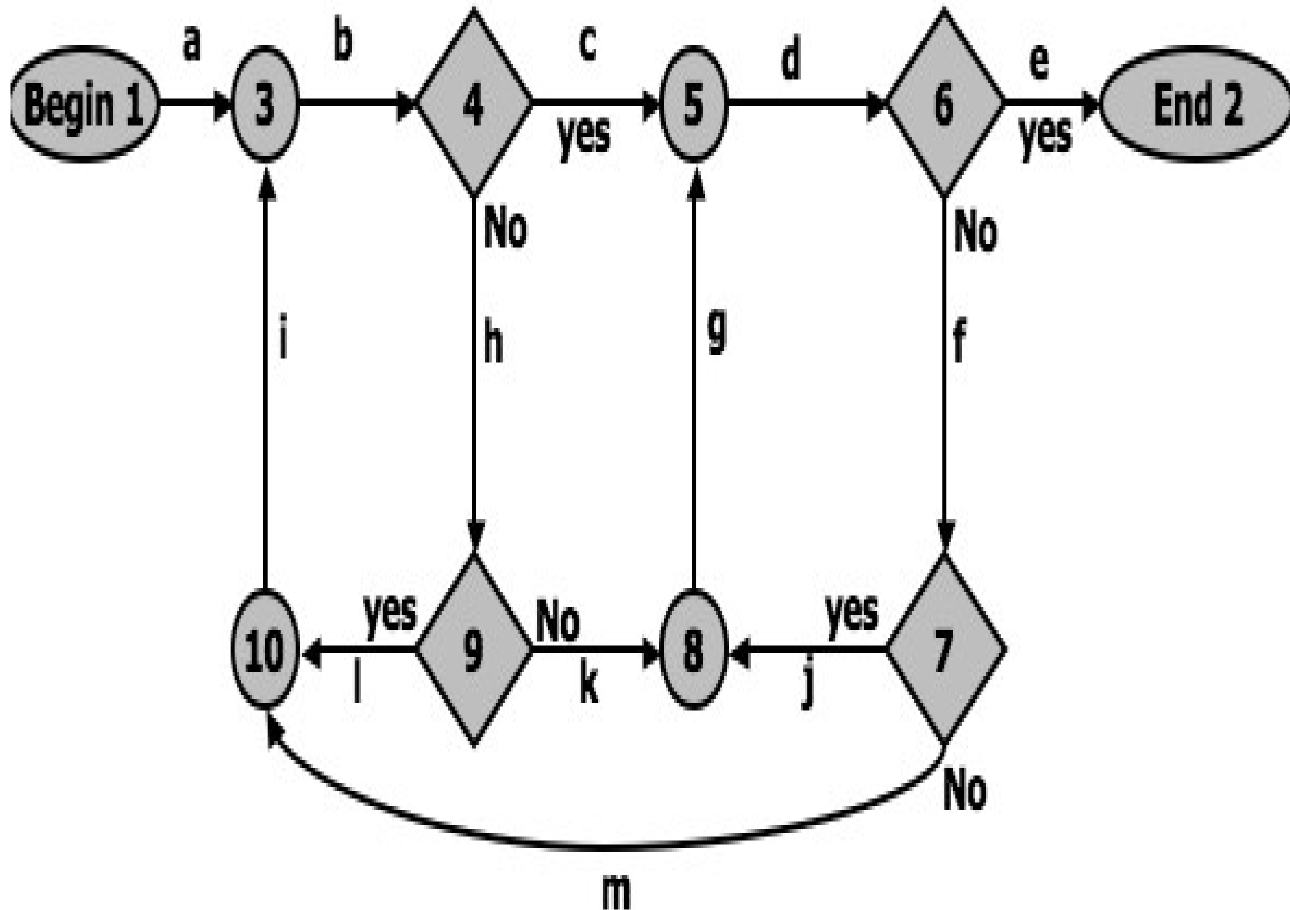


Figure 2.9: An example flowgraph to explain path selection

- **Practical Suggestions in Path Testing:**

  - Draw the control flow graph on a single sheet of paper.
  - Make several copies - as many as you will need for coverage (C1+C2) and several more.
  - Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.

- Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
- As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
- The above paths lead to the following table considering Figure 2.9:

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

- After you have traced a a covering path set on the master sheet and filled in the table for every path, check the following:
    1. Does every decision have a YES and a NO in its column? (C2)
    2. Has every case of all case statements been marked? (C2)
    3. Is every three - way branch (less, equal, greater) covered? (C2)
    4. Is every link (process) covered at least once? (C1)
- **Revised Path Selection Rules:**

    - Pick the simplest, functionally sensible entry/exit path.
    - Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
    - Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.

- Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly (blindly) - except for coverage.

3) Discuss in detail data flow testing strategies?(CO4)

- Data Flow Testing Strategies are structural strategies.
- In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
- In other words, data flow strategies require data-flow link weights (d,k,u,c,p).
- Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.
- For example, all subpaths that contain a d (or u, k, du, dk).
- A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for**weaker**.

- **TERMINOLOGY:**

1. **Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. ll paths in Figure 3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).
2. **Loop-Free Path Segment** is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.
3. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.
4. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition-clear.

2 **STRATEGIES:** The structural test strategies discussed below are based on the program's control flowgraph. They differ in the extent to which predicate uses and/or

computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

1. **All - du Paths (ADUP):** The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

   For variable X and Y: In Figure 3.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

   For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

   For variable V: Variable V (Figure 3.11) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4). Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

   The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.
2. **All Uses Startegy (AU):** The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test. Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

   For variable V: In Figure 3.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath. Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

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

   For variable Z: In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered. Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables in this example - it only takes two tests.

   For variable V: In Figure 3.11, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the c-use at (9,10) need not be included under the APU+C criterion.
4. **All c-uses/some p-uses strategy (ACU+P) :** The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

   For variable Z: In Figure 3.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

   > *The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.*

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

For variable Z: Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.

> *From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.*

6. **All Predicate Uses (APU), All Computational Uses (ACU) Strategies :** The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the

requirement that we include a p-use for the variable if there are no c-uses for the variable.

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

3 **ORDERING THE STRATEGIES:**

- Figure 3.12 compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head.

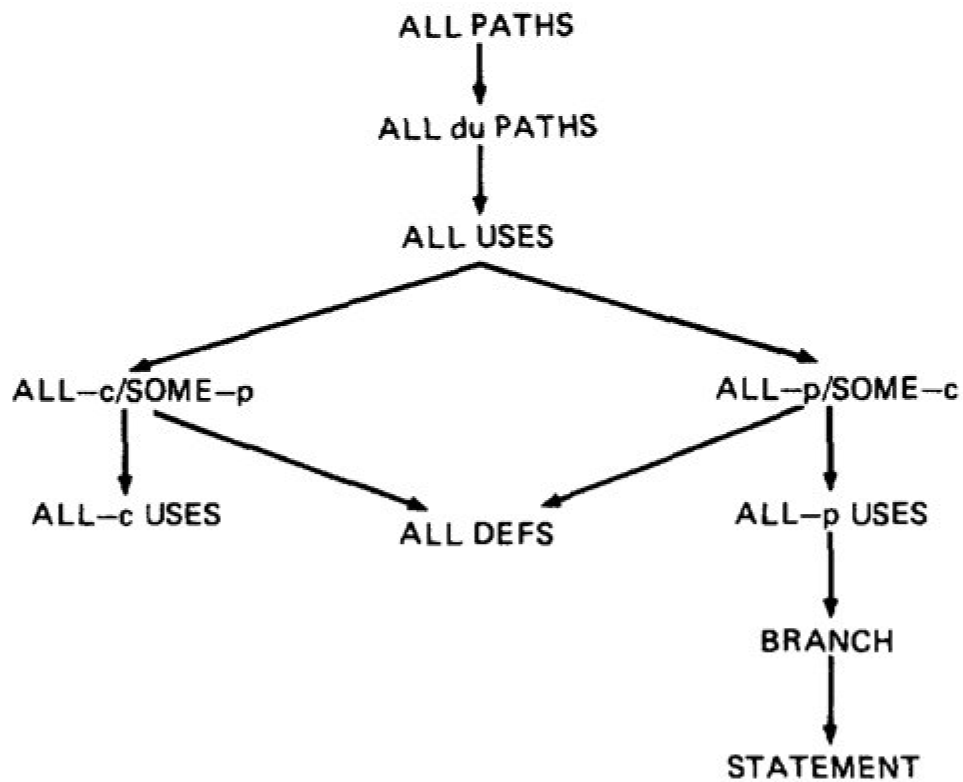


Figure 3.12: Relative Strength of Structural Test Strategies.

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

2 **SLICING AND DICING:**

- A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.
- If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i
- A program **dice** is a part of a slice in which all statements which are known to be correct have been removed.
- In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).
- The debugger first limits her scope to those prior statements that could have caused the faulty value at statement i (the slice) and then eliminates from further consideration those statements that testing has shown to be correct.
- Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.
- **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

4) Explain path sensitizing? Define path instrumentation?(CO5)

PATH SENSITIZING:

2 REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.
- Extract the programs control flowgraph and select a set of tentative covering paths.
- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as

(A+BC) (D+E) (FGH) (IJ) (K) (l) (L).

- Multiply out the expression to achieve a sum of products form:

ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL

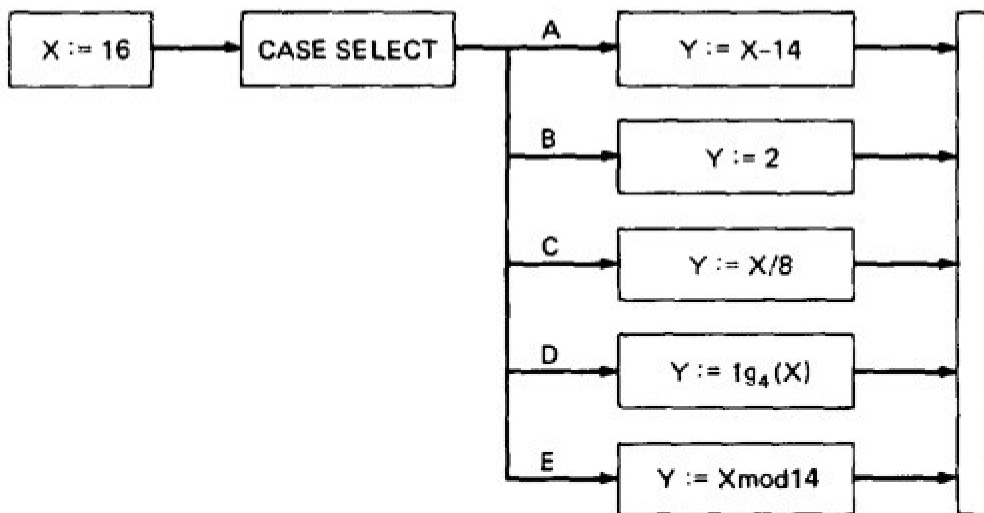
- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you cant find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

3 HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

4 PATH INSTRUMENTATION:

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



5 ***Figure 2.11: Coincidental Correctness***

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

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

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

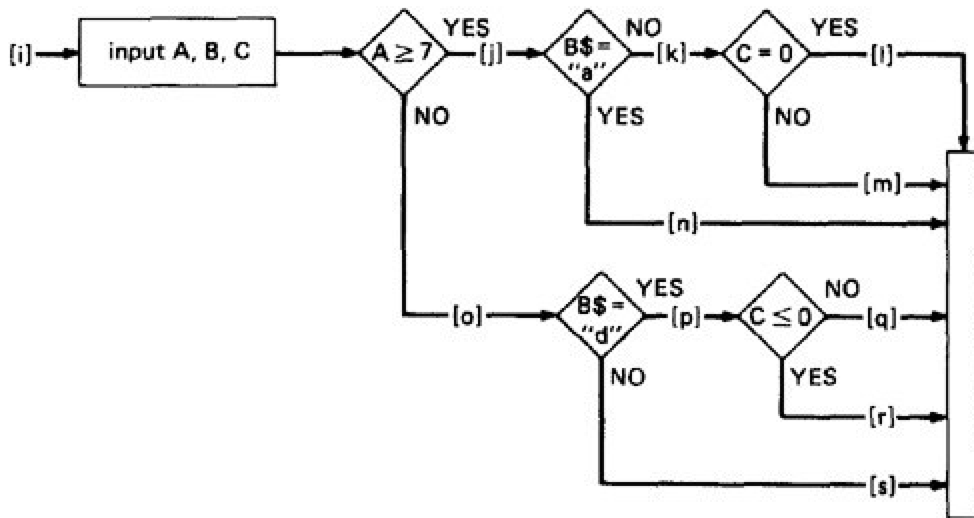
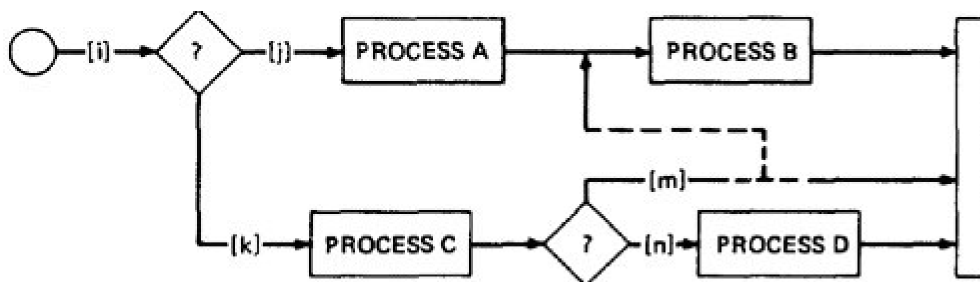


Figure 2.12: Single Link Marker Instrumentation

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



- ***Figure 2.13: Why Single Link Markers aren't enough.***
- We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

- **Two Link Marker Method:**
    - The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and on at the end.
    - The two link markers now specify the path name and confirm both the beginning and end of the link.

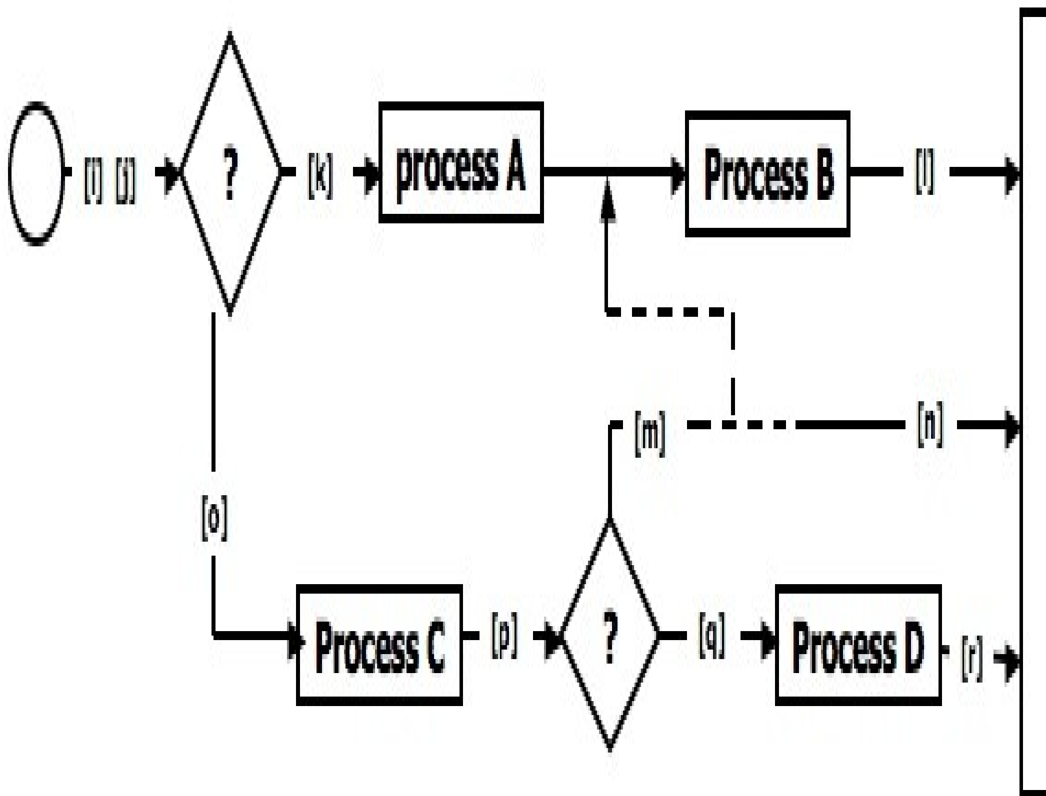


Figure 2.14: Double Link Marker Instrumentation.

- **Link Counter:** A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

5) List and explain the data flow anomalies?(CO2)

DATA FLOW TESTING:

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

.

- **DATA FLOW MACHINES:**
  - There are two types of data flow machines with different architectures. (1) Von Neumann machnes (2) Multi-instruction, multi-data machines (MIMD).
  - **Von Neumann Machine Architecture:**
    - Most computers today are von-neumann machines.
    - This architecture features interchangeable storage of instructions and data in the same memory units.
    - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
      1. Fetch instruction from memory
      2. Interpret instruction
      3. Fetch operands
      4. Process or Execute
      5. Store result
      6. Increment program counter
      7. GOTO 1
  - **Multi-instruction, Multi-data machines (MIMD) Architecture:**
    - These machines can fetch several instructions and objects in parallel.
    - They can also do arithmetic and logical operations simultaneously on different data objects.

    - The decision of how to sequence them depends on the compiler.

- **BUG ASSUMPTION:**
    - The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.
    - Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
    - Although we'll be doing data-flow testing, we won't be using data flowgraphs as such. Rather, we'll use an ordinary control flowgraph annotated to show what happens to the data objects of interest at the moment.
- **DATA FLOW GRAPHS:**
    - The data flow graph is a graph consisting of nodes and directed links.

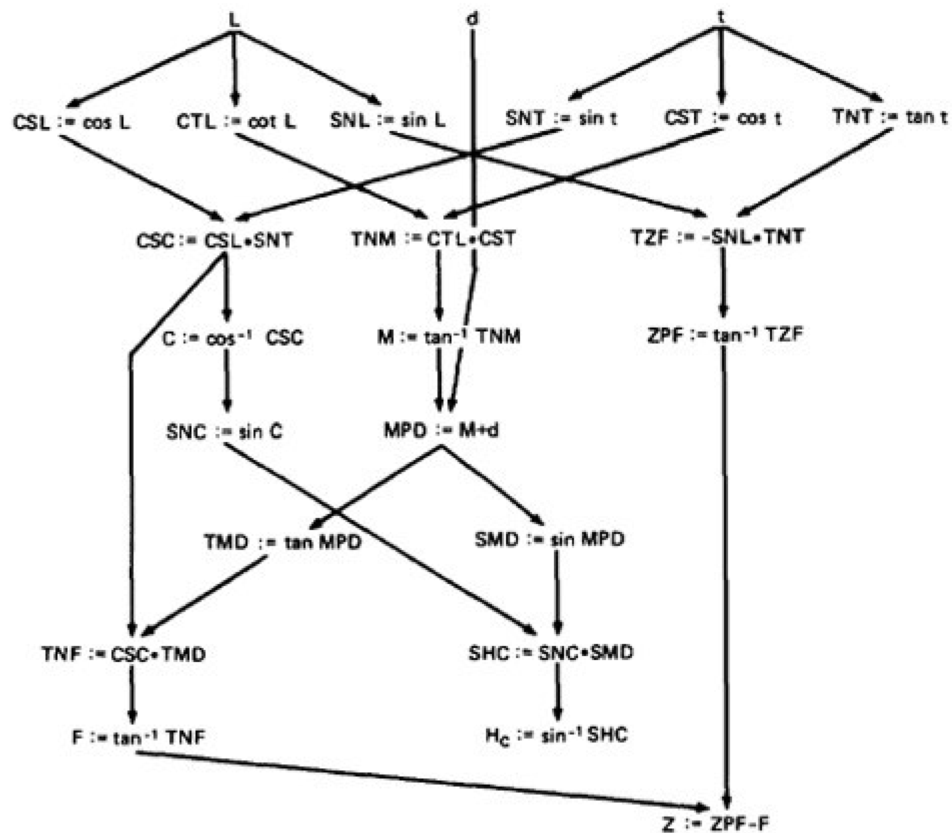


Figure 3.4: Example of a data flow graph

    - We will use an control graph to show what happens to data objects of interest at that moment.

- Our objective is to expose deviations between the data flows we have and the data flows we want.
- **Data Object State and Usage:**
  - Data Objects can be created, killed and used.
  - They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
  - The following symbols denote these possibilities:
    1. **Defined:** d - defined, created, initialized etc
    2. **Killed or undefined:** k - killed, undefined, released etc
    3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)
  - **1. Defined (d):**

    - An object is defined explicitly when it appears in a data declaration.
    - Or implicitly when it appears on the left hand side of the assignment.
    - It is also to be used to mean that a file has been opened.
    - A dynamically allocated object has been allocated.
    - Something is pushed on to the stack.
    - A record written.

- **2. Killed or Undefined (k):**
    - An object is killed on undefined when it is released or otherwise made unavailable.
    - When its contents are no longer known with certitude (with aboslute certainity / perfectness).
    - Release of dynamically allocated objects back to the availability pool.
    - Return of records.
    - The old top of the stack after it is popped.
    - An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A
- **3. Usage (u):**
    - A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
    - A file record is read or written.
    - It is used in a Predicate (p) when it appears directly in a predicate.

2 DATA FLOW ANOMALIES:

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

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

3 What is an anomaly is depend on the application.

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

- **dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
- **dk** :- probably a bug. Why define the object without using it?
- **du** :- the normal case. The object is defined and then used.
- **kd** :- normal situation. An object is killed and then redefined.
- **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
- **ku** :- a bug. the object doesnot exist.
- **ud** :- usually not a bug because the language permits reassignment at almost any time.
- **uk** :- normal situation.
- **uu** :- normal situation.

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

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

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

5 They possible anomalies are:

- **-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
- **-d** :- okay. This is just the first definition along this path.
- **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
- **k-** :- not anomalous. The last thing done on this path was to kill the variable.
- **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
- **u-** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

2 DATA FLOW ANOMALY STATE GRAPH:

1 Data flow anomaly model prescribes that an object can be in one of four distinct states:

- **K** :- undefined, previously killed, doesnot exist
- **D** :- defined but not yet used for anything
- **U** :- has been used for computation or in predicate
- **A** :- anomalous

2 These capital letters (K,D,U,A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

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

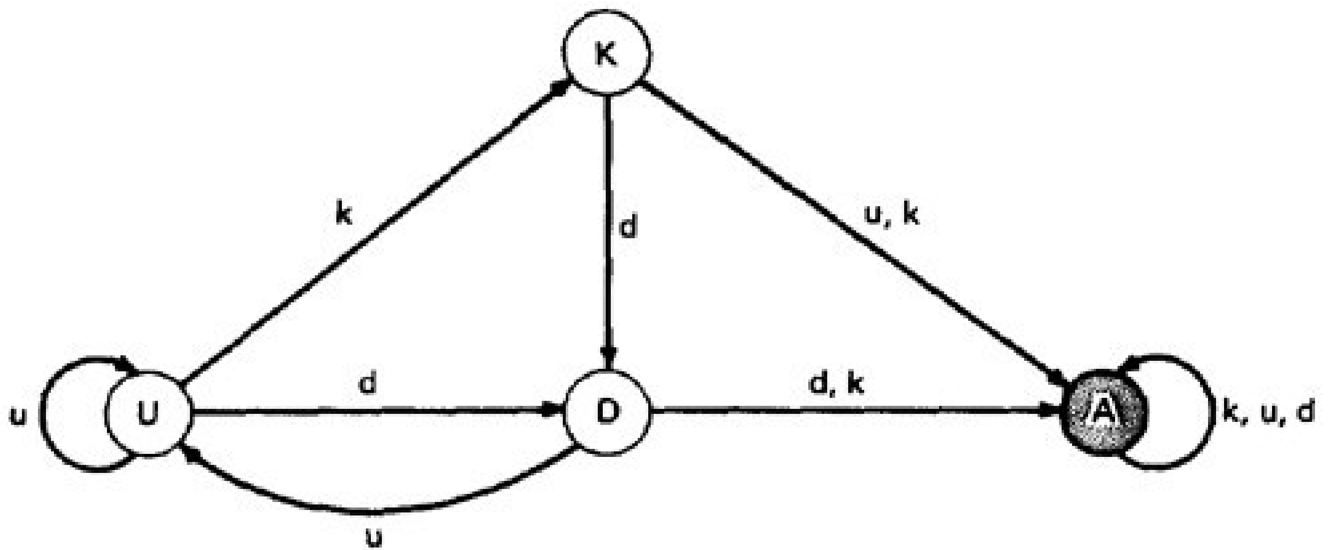


Figure 3.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state. If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

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

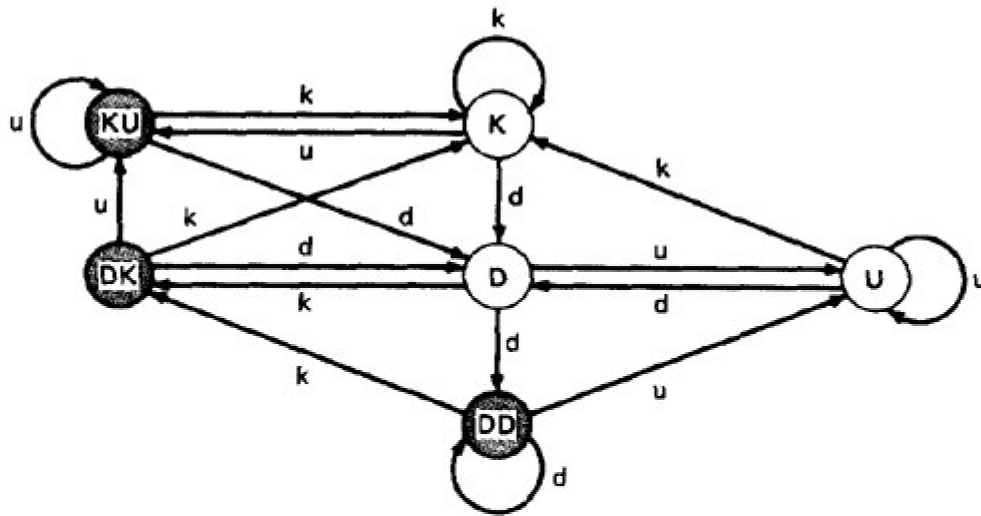


Figure 3.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 3.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

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

3 STATIC Vs DYNAMIC ANOMALY DETECTION:

1. Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.
2. Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.
3. If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesnot belongs in testing - it belongs in the language processor.
4. There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.
5. For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.

6 But still there are many things for which current notions of static analysis are INADEQUATE.

7 **Why Static Analysis isn't enough?** There are many things for which current notions of static analysis are inadequate. They are:

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

flow anomalies that cannot be detected in the context of a single routine.

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

2

3

4

6) How do convert flow graph into flowchart?(CO4)

1 A program's flow chart resembles a control flow graph.

2 In flow graphs, we don't show the details of what is in a process block.

3 In flow charts every part of the process block is drawn.

4 The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.

5 The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

2 **NOTATIONAL EVOULTION:**

1 The control flow graph is simplified representation of the program's structure.

2 The notation changes made in creation of control flow graphs:

- The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- We don't need to know the specifics of the decisions, just the fact that there is a branch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called**Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.
- The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like.

CODE* (PDL)

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

* A contrived horror

Figure 2.2: Program Example (PDL)

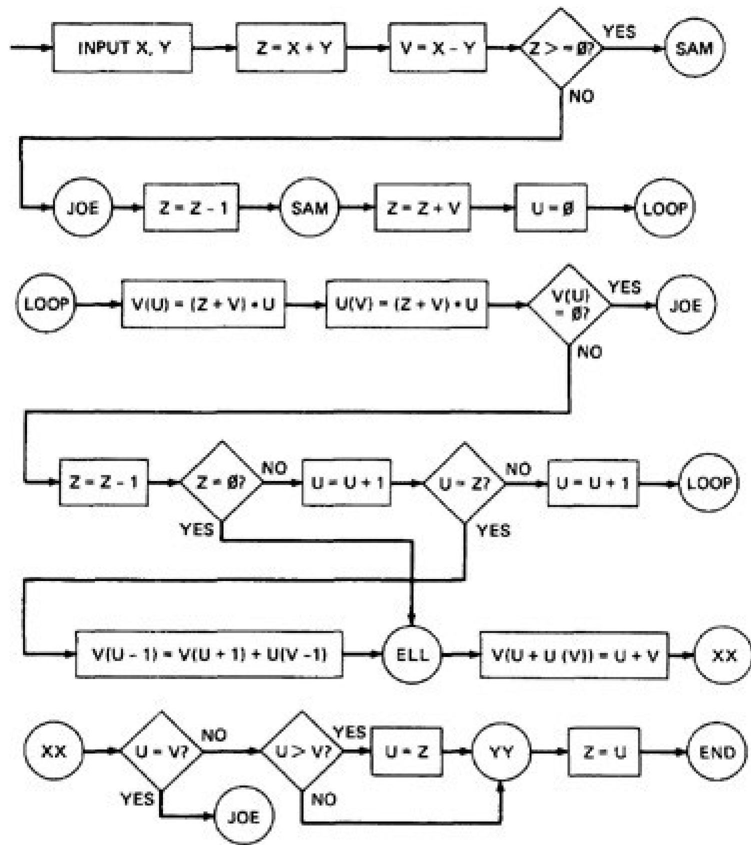


Figure 2.3: One-to-one flowchart for example program in Figure 2.2

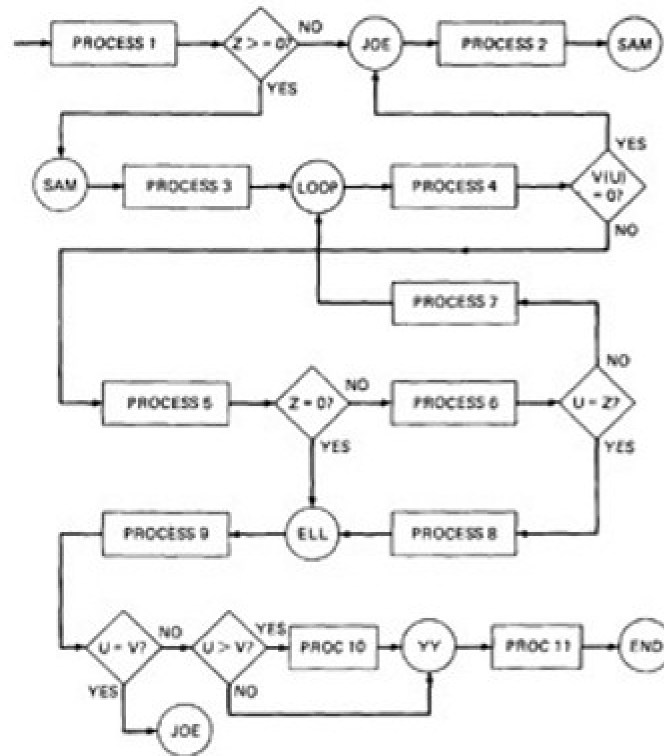


Figure 2.4: Control Flowgraph for example in Figure 2.2

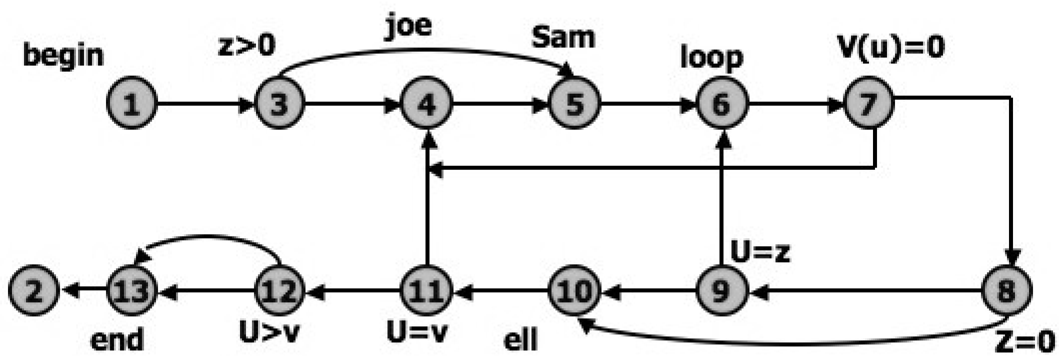


Figure 2.5: Simplified Flowgraph Notation

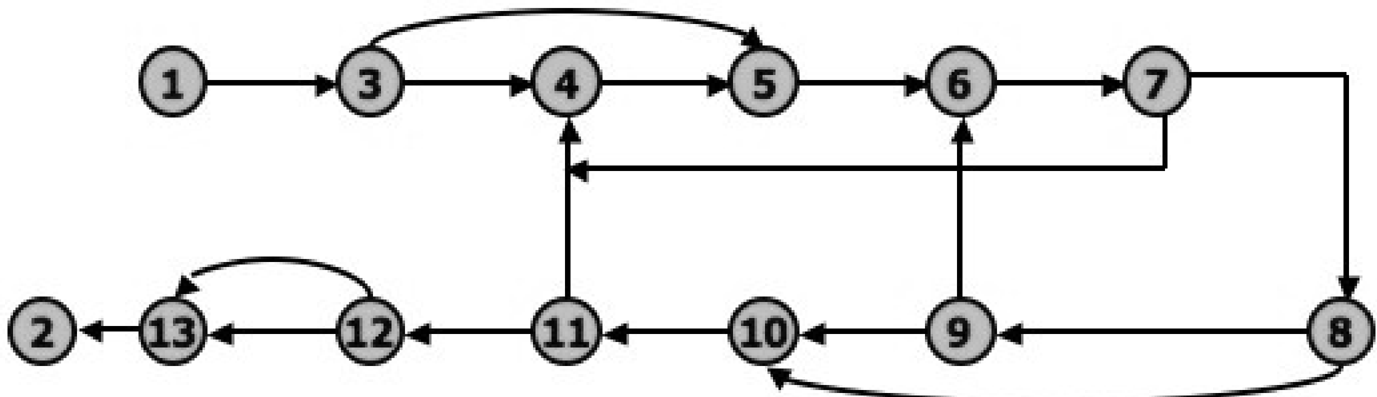


Figure 2.6: Even Simplified Flowgraph Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation.

7) Explain the process of achieving c1+c2 coverage(CO4)

4 **PATH TESTING CRITERIA:**

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

2 A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

3 So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

1. **Path Testing (P_{inf})**:

   - Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
   - If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

2. **Statement Testing (P_1)**:

- Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
- An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
- This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or can not be accepted) and should be criminalized.

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

3 Commonsense and Strategies:

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.
- **Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

- **Path Selection Example:**

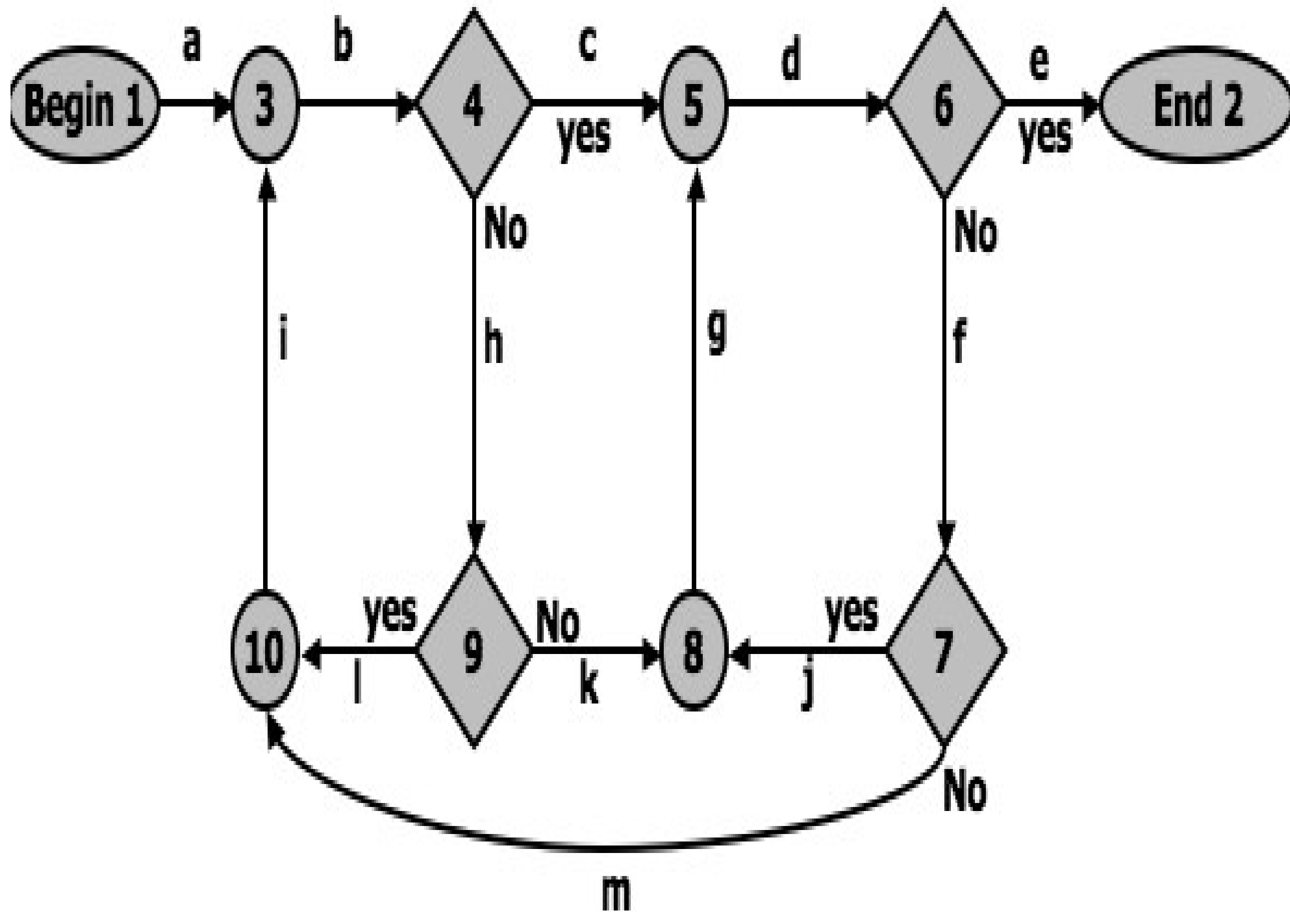


Figure 2.9: An example flowgraph to explain path selection

- **Practical Suggestions in Path Testing:**

  - Draw the control flow graph on a single sheet of paper.
  - Make several copies - as many as you will need for coverage (C1+C2) and several more.
  - Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.

- Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
- As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
- The above paths lead to the following table considering Figure 2.9:

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

- After you have traced a a covering path set on the master sheet and filled in the table for every path, check the following:
  1. Does every decision have a YES and a NO in its column? (C2)
  2. Has every case of all case statements been marked? (C2)
  3. Is every three - way branch (less, equal, greater) covered? (C2)
  4. Is every link (process) covered at least once? (C1)
- **Revised Path Selection Rules:**

  - Pick the simplest, functionally sensible entry/exit path.
  - Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
  - Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.

- Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly (blindly) - except for coverage.

8) Explain kinds of loops with respect to path testing?(CO6)

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

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

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

- Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
- The minimum number of iterations.
- One more than the minimum number of iterations.

- Once, unless covered by a previous test.
- Twice, unless covered by a previous test.
- A typical value.
- One less than the maximum value.
- The maximum number of iterations.
- Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

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

- **Kinds of Loops:**There are only three kinds of loops with respect to path testing:
    - **Nested Loops:**
        - The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.
        - As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:
            1. Start at the inner most loop. Set all the outer loops to their minimum values.
            2. Test the minimum, minimum+1, typical, maximum-1 , and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values.
            3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.
            4. Continue outward in this manner until all loops have been covered.

5. Do all the cases for all loops in the nest simultaneously.

- **Concatenated Loops:**
  - Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.
  - If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- **Horrible Loops:**
  - A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.
  - Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

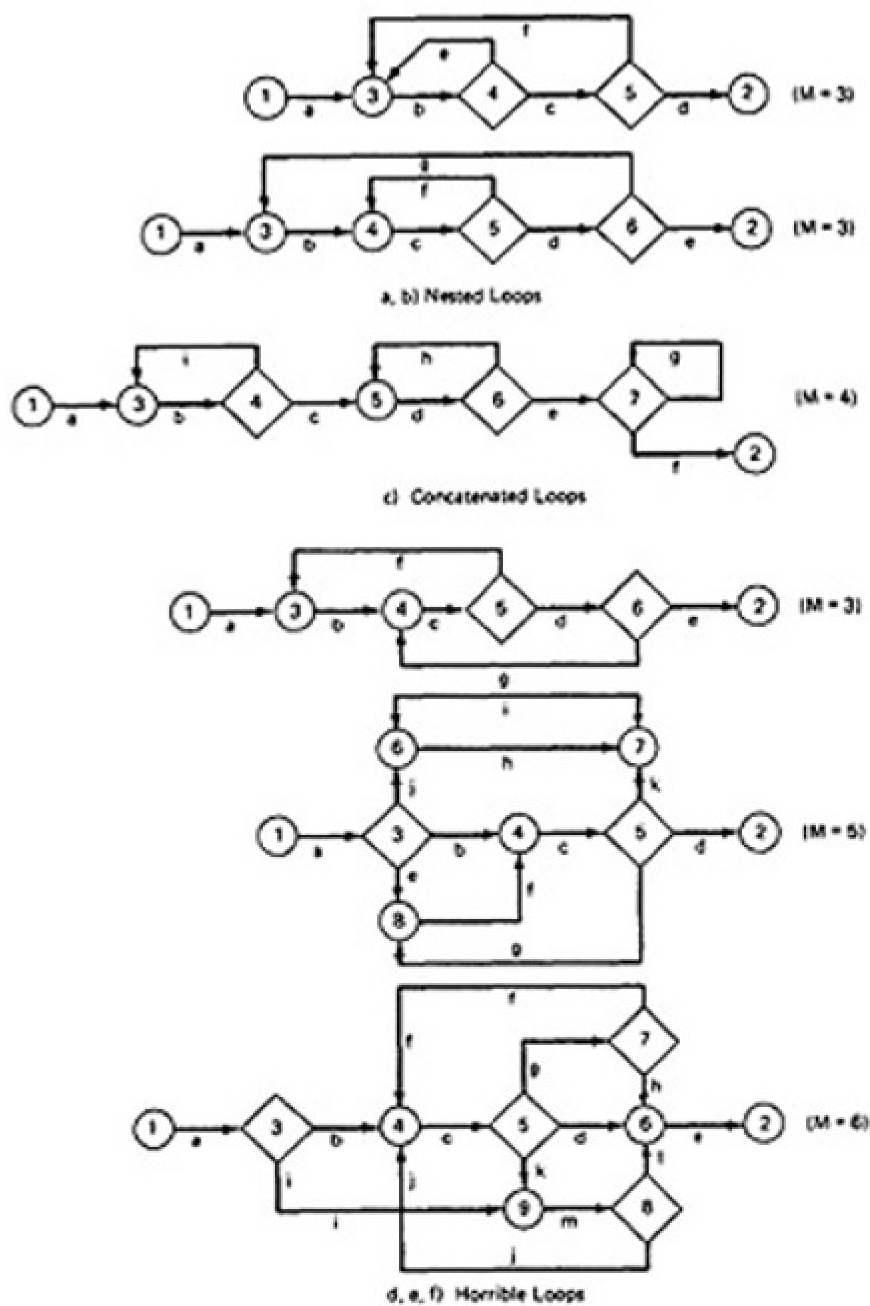


Figure : Example of Loop types

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

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

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

9)Discuss about testing Blindness?(CO1)

2 **TESTING BLINDNESS:**

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

 Assignment Blindness:

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

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

 Equality Blindness:

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

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

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

Self Blindness:

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

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

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

10) Explain the transaction flow techniques?(CO2)

2 **GET THE TRANSACTIONS FLOWS:**

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

2 Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.

3 The system's design documentation should contain an overview section that details the main transaction flows.

4 Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

3 **INSPECTIONS, REVIEWS AND WALKTHROUGHS:**

1 Transaction flows are natural agenda for system reviews or inspections.

2 In conducting the walkthroughs, you should:

- Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
- Discuss paths through flows in functional rather than technical terms.
- Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.

3 Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.

4 Select additional flow paths for loops, extreme values, and domain boundaries.

5 Design more test cases to validate all births and deaths.

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

7
4 **PATH SELECTION:**
1 Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
2 Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
3 Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.
5 **PATH SENSITIZATION:**
1 Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
2 The remaining small percentage is often very difficult.
3 Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.
6 **PATH INSTRUMENTATION:**
1 Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
2 The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
3 In some systems, such traces are provided by the operating systems or a running log.
7

11) Explain about nice domain and ugly domain?CO4)

8
9 **NICE DOMAINS:**
1
Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction.
2 Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent.
3 Conversely, specified domains can be incomplete and/or inconsistent. Incomplete in this context means that there are input vectors for which no path is specified, and inconsistent means that there are at least two contradictory specifications over the same segment of the input space.
4 Some important properties of nice domains are: **Linear, Complete, Systematic, Orthogonal, Consistently closed, Convex and Simply connected.**
5 To the extent that domains have these properties domain testing is easy as testing gets.
6 The bug frequency is lesser for nice domain than for ugly domains.

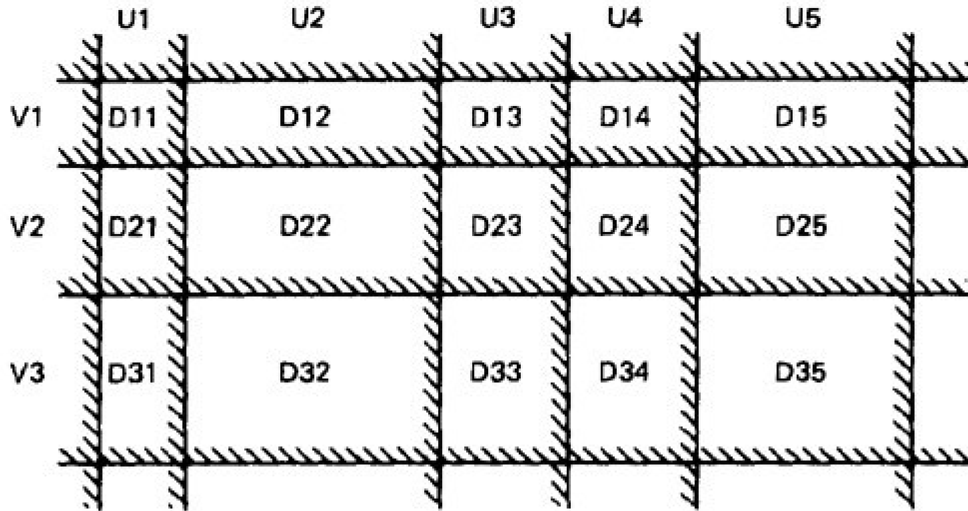


Figure 4.3: Nice Two-Dimensional Domains.

10 **LINEAR AND NON LINEAR BOUNDARIES:**

1 Nice domain boundaries are defined by linear inequalities or equations.

2 The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general n+1 points to determine a n-dimensional hyper plane.

3 In practice more than 99.99% of all boundary predicates are either linear or can be linearized by simple variable transformations.

11 **COMPLETE BOUNDARIES:**

1 Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.

2 Figure 4.4 shows some incomplete boundaries. Boundaries A and E have gaps.

3 Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a null set.

4 The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds.

5 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.

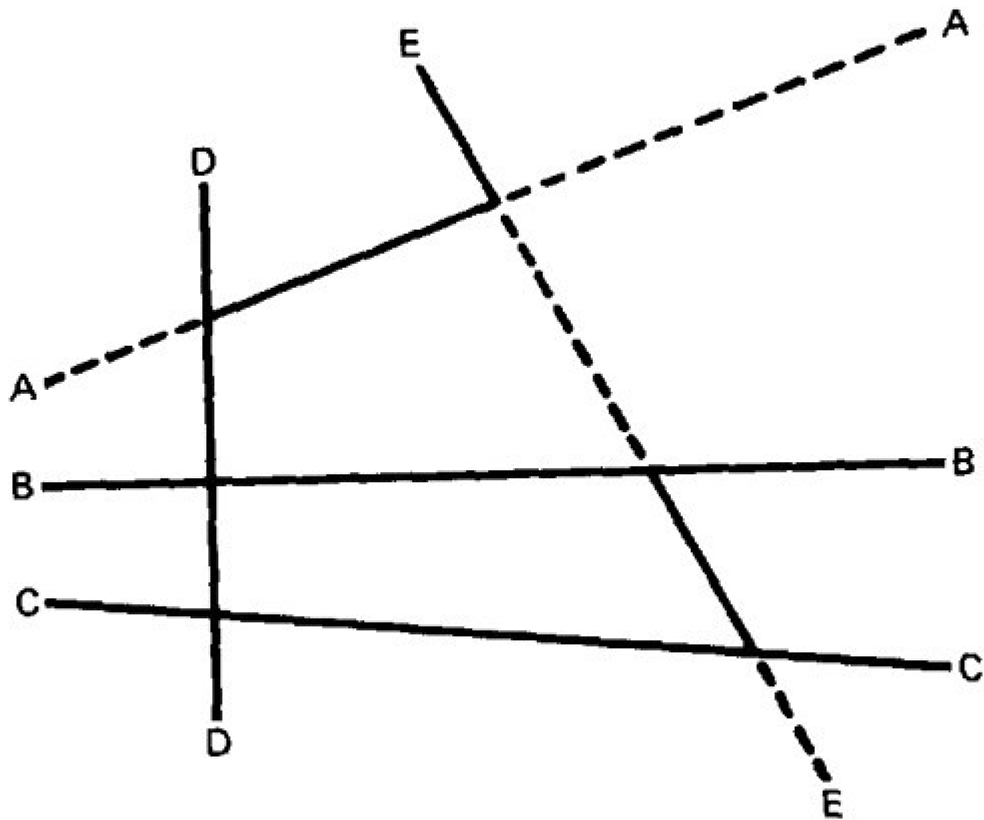


Figure 4.4: Incomplete Domain Boundaries.

12 **SYSTEMATIC BOUNDARIES:**

1 Systematic boundary means that boundary inequalities related by a simple function such as a constant.

2 In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant. We want relations such as

$$
\begin{array}{ll}
f_1(X) >= k_1 \text{ or } f_1(X) >= g(1,c) \\
f_1(X) >= k_2 & f_2(X) >= g(2,c) \\
\ldots\ldots\ldots\ldots & \ldots\ldots\ldots\ldots \\
f_i(X) >= k_i & f_i(X) >= g(i,c)
\end{array}
$$

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, such as $k + ic$.

3 The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines (such as the spokes of a wheel, if equally spaced in angles, systematic).

4 If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.

13 **ORTHOGONAL BOUNDARIES:**

1 Two boundary sets U and V (See Figure 4.3) are said to be orthogonal if every inequality in V is perpendicular to every inequality in U.

2 If two boundary sets are orthogonal, then they can be tested independently

3 In Figure 4.3 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)). If we tilt the boundaries to get Figure 4.5, we must now test the intersections. We've gone from a linear number of cases to a quadratic: from O(n) to $O(n^2)$.

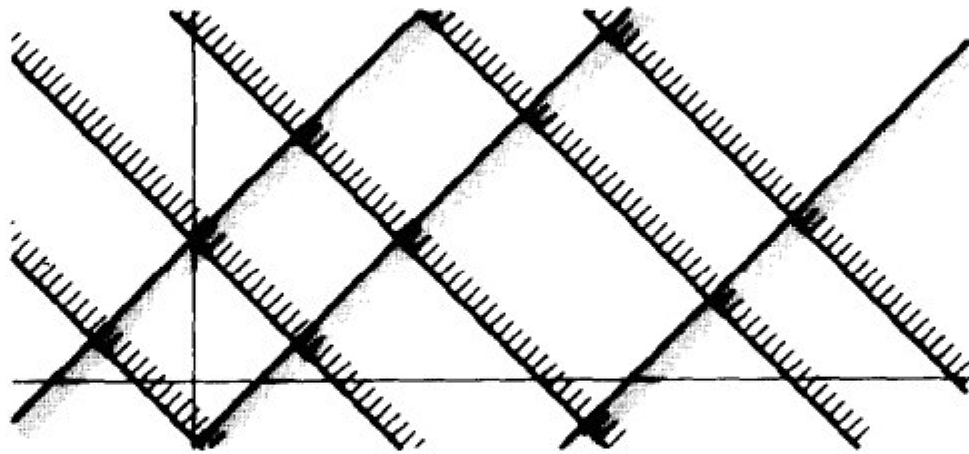


Figure 4.5: Tilted Boundaries.

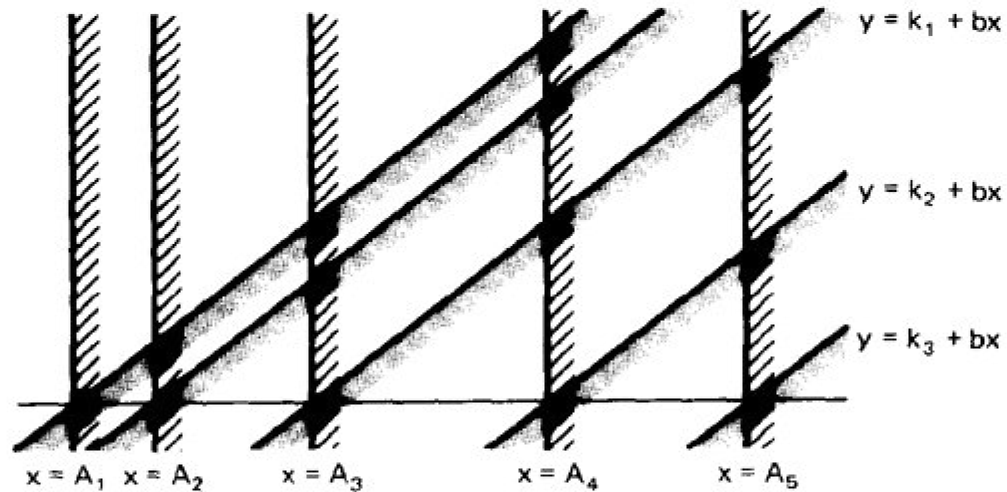


Figure 4.6: Linear, Non-orthogonal Domain Boundaries.

4 Actually, there are two different but related orthogonality conditions. Sets of boundaries can be orthogonal to one another but not orthogonal to the coordinate axes (condition 1), or boundaries can be orthogonal to the coordinate axes (condition 2).

14 **CLOSURE CONSISTENCY:**

1 Figure 4.6 shows another desirable domain property: boundary closures are consistent and systematic.

2 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.

3 Consistent closure means that there is a simple pattern to the closures - for example, using the same relational operator for all boundaries of a set of parallel boundaries.

15 **CONVEX:**

1 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.

2 Nice domains are convex; dirty domains aren't.

3 You can smell a suspected concavity when you see phrases such as: ". . . except if . . .," "However . . .," ". . . but not. . . ." In programming, it's often the buts in the specification that kill you.

16 **SIMPLY CONNECTED:**

1 Nice domains are simply connected; that is, they are in one piece rather than pieces all over the place interspersed with other domains.

2 Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

3 Consider domain boundaries defined by a compound predicate of the (boolean) form ABC. Say that the input space is divided into two domains, one defined by ABC and, therefore, the other defined by its negation .

4 For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than 17.

5 Simple connectivity, especially for default cases, may be impossible.

---

17 UGLY DOMAINS:

1 Some domains are born ugly and some are uglified by bad specifications.

2 Every simplification of ugly domains by programmers can be either good or bad.

3 Programmers in search of nice solutions will "simplify" essential complexity out of existence. Testers in search of brilliant insights will be blind to essential complexity and therefore miss important cases.

4 If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.

5 But if the domain's complexity is essential (e.g., the income tax code), such "simplifications" constitute bugs.

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

18 AMBIGUITIES AND CONTRADICTIONS:

1 Domain ambiguities are holes in the input space.

2 The holes may lie with in the domains or in cracks between domains.

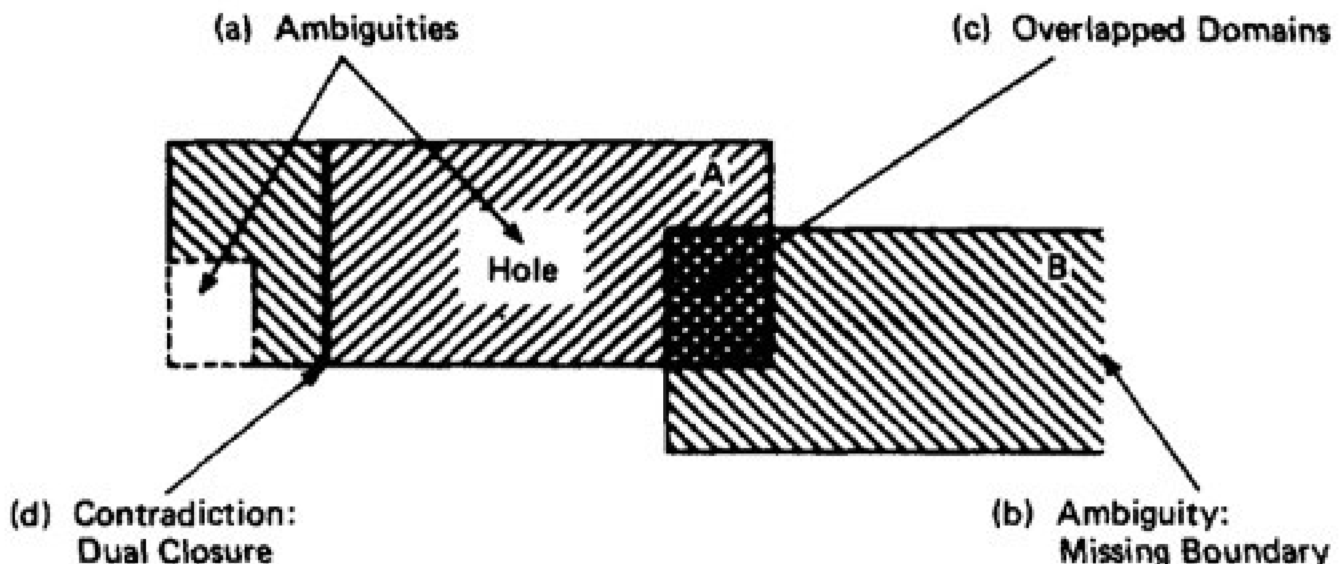


Figure 4.7: Domain Ambiguities and Contradictions.

3 Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specifications

4 Figure 4.7c shows overlapped domains and Figure 4.7d shows dual closure assignment.

19 **SIMPLIFYING THE TOPOLOGY:**

1 The programmer's and tester's reaction to complex domains is the same - simplify

2 There are three generic cases: **concavities**, **holes** and **disconnected pieces**.

3 Programmers introduce bugs and testers misdesign test cases by: smoothing out concavities (Figure 4.8a), filling in holes (Figure 4.8b), and joining disconnected pieces (Figure 4.8c).

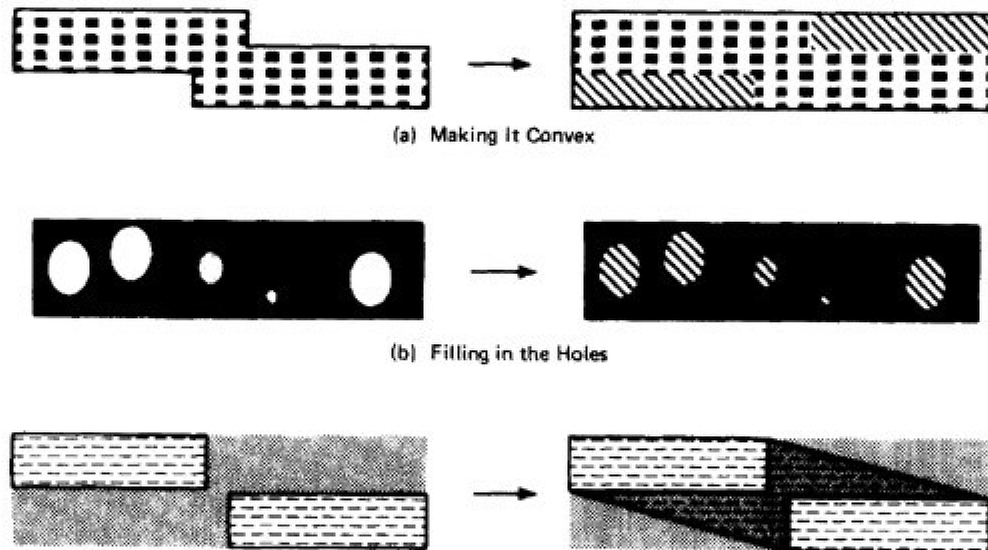


Figure : Simplifying the topology.

20 **RECTIFYING BOUNDARY CLOSURES:**

1 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 4.9).

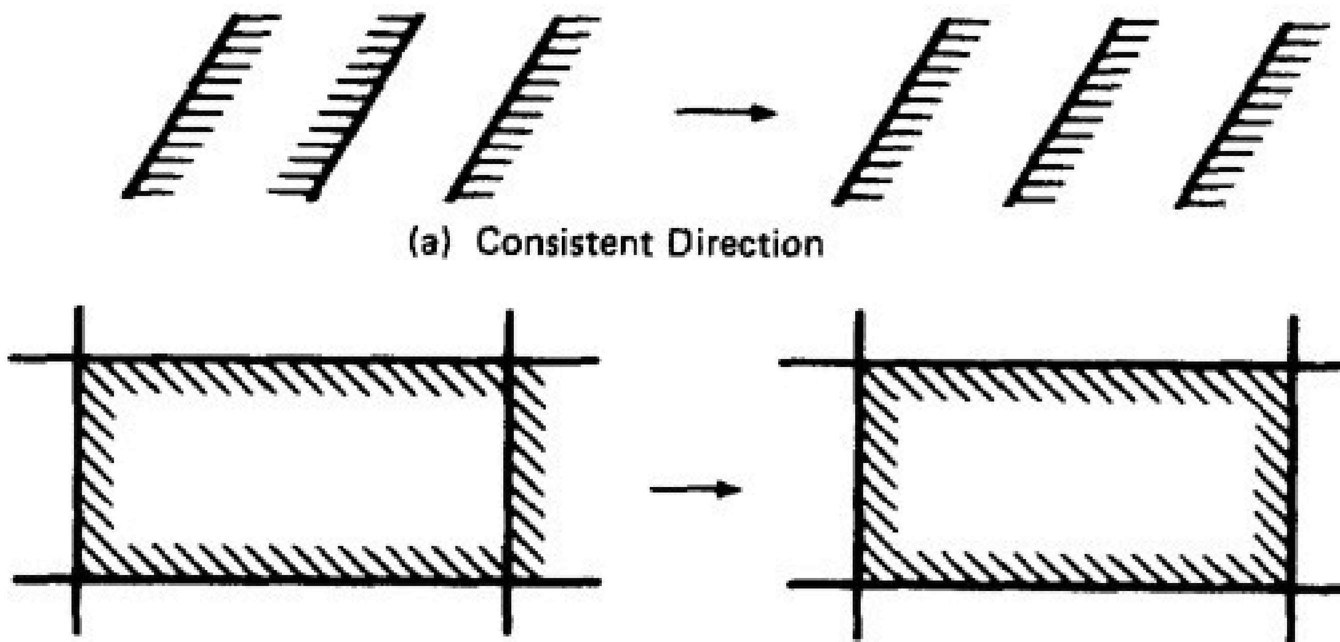


Figure 4.9: Forcing Closure Consistency.

12)Explain briefly about Domain Testing?

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

    - An **interior point** (Figure 4.10) is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.

- A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in the domain.
- An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain.

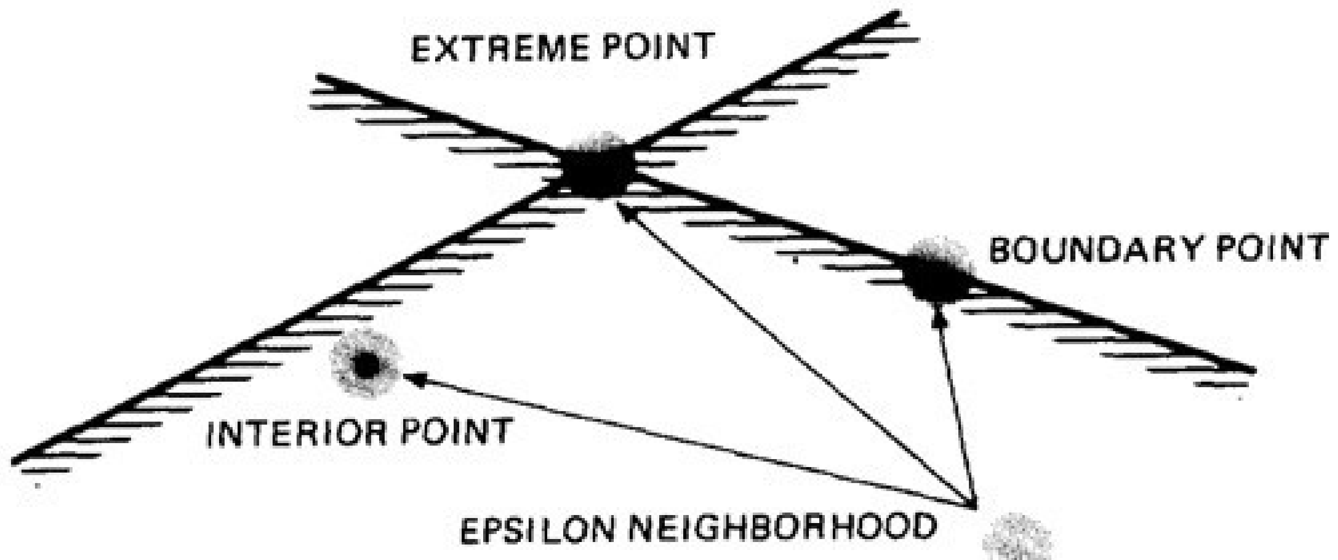


Figure 4.10: Interior, Boundary and Extreme points.

- An **on point** is a point on the boundary.
- If the domain boundary is closed, an **off point** is a point near the boundary but in the adjacent domain.
- If the boundary is open, an off point is a point near the boundary but in the domain being tested; see Figure 4.11. You can remember this by the acronym COOOOI: Closed Off Outside, Open Off Inside.

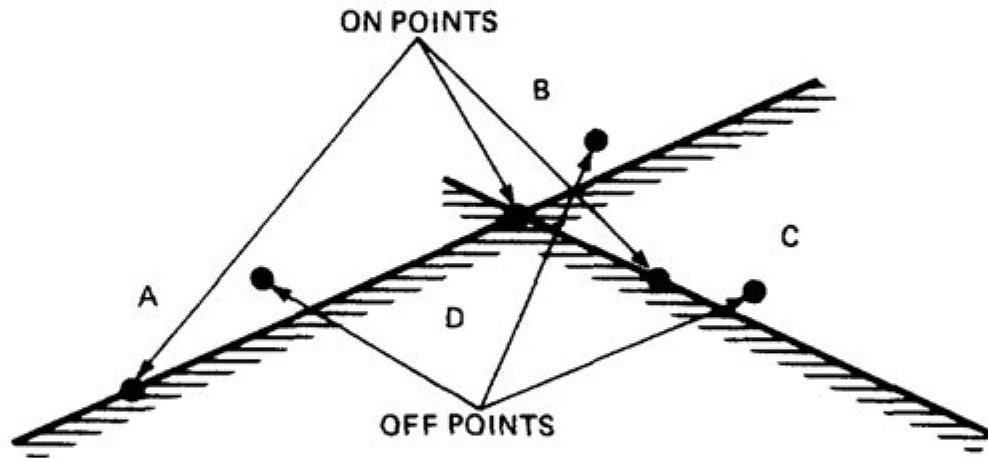


Figure 4.11: On points and Off points.

- Figure 4.12 shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.

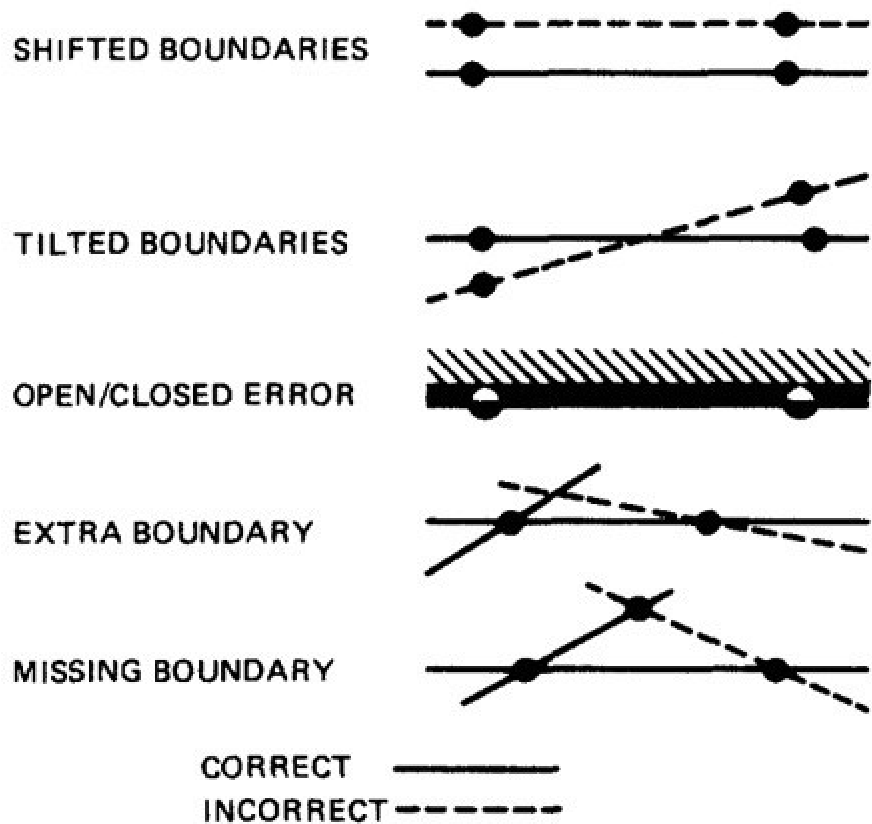


Figure 4.12: Generic Domain Bugs.

- **TESTING ONE DIMENSIONAL DOMAINS:**
  - Figure 4.13 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.

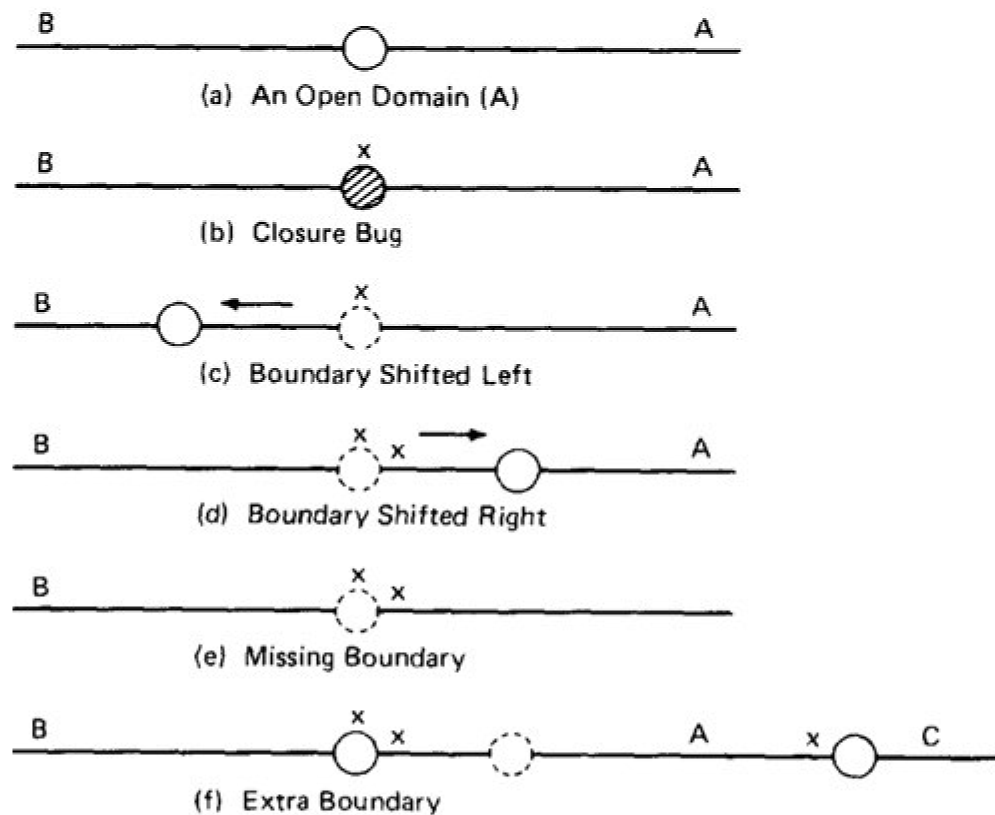


Figure 4.13: One Dimensional Domain Bugs, Open Boundaries.

- In Figure 4.13a 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 4.13b). 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 4.13c 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 4.13d 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. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition, the off point is in A (Open Off Inside).
- The same open off point also suffices to detect a missing boundary because what should have been processed in A is now processed in B.
- To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The

two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would do it.

- For closed domains look at Figure 4.14. 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.

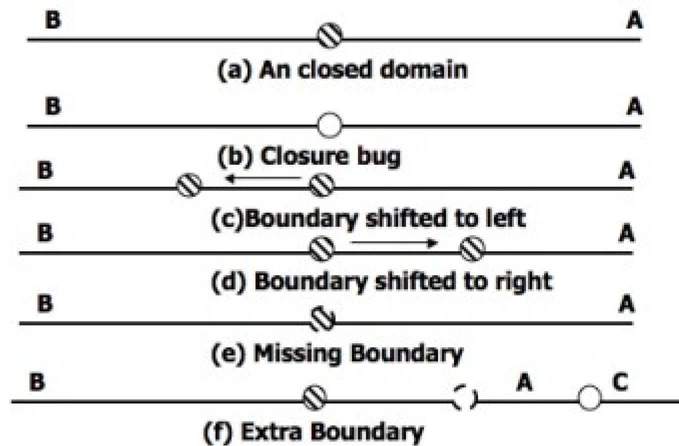


Figure 4.14: One Dimensional Domain Bugs, Closed Boundaries.

- **TESTING TWO DIMENSIONAL DOMAINS:**
  - Figure 4.15 shows possible domain boundary bugs for a two-dimensional domain.
  - 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.

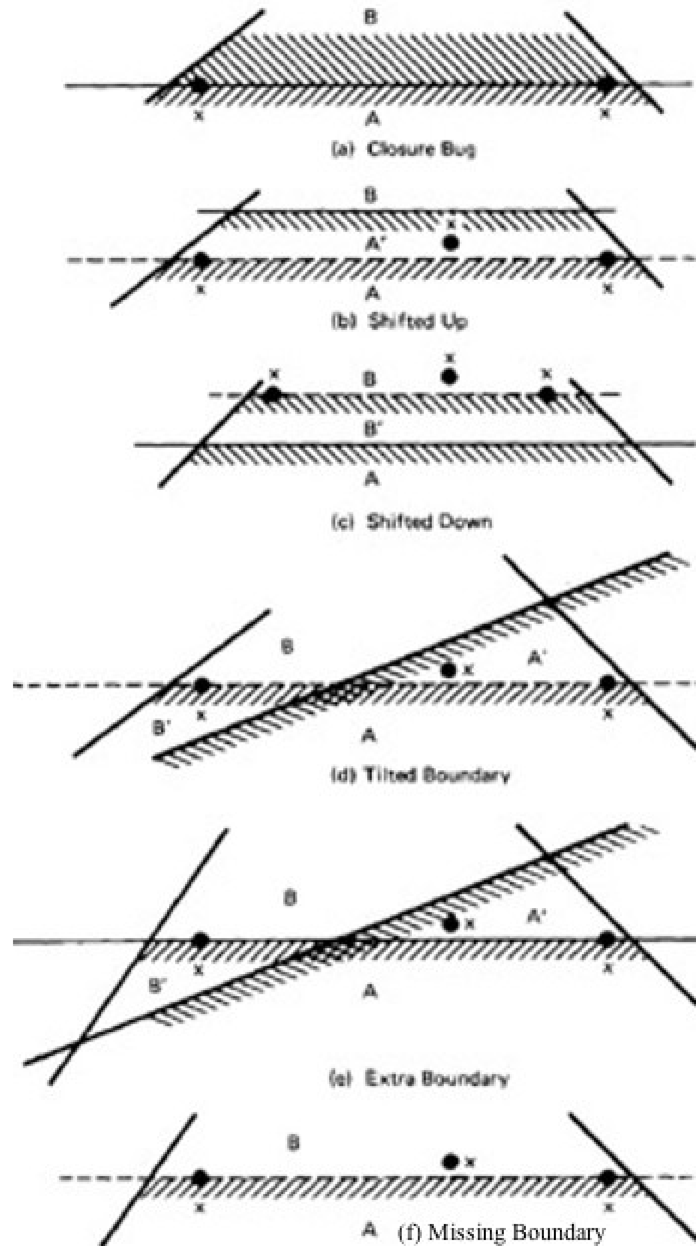


Figure 4.15: Two Dimensional Domain Bugs.

- **For Closed Boundaries:**
  1. **Closure Bug:** Figure 4.15a 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 4.15b 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 4.15c 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 4.15d 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 4.15e is caught by two on points, and depending on which way the extra boundary goes, possibly by the off point also.
5. **Missing Boundary:** A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and will cause many test failures although there is only one bug. A missing boundary, shown in Figure 4.15f, is caught by the two on points because the processing for A and B is the same - either A or B processing.

- **PROCEDURE FOR TESTING:** The procedure is conceptually is straight forward. It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.

1. Identify input variables.
2. Identify variable which appear in domain defining predicates, such as control flow predicates.
3. Interpret all domain predicates in terms of input variables.
4. For p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. The result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. For example ABC+DEF+GHI ...... Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connected domains.

Solve these inequalities to find all the extreme points of each domain using any of the linear programming methods

13)Explain Domain and Interface Testing in detail?

- **DOMAINS AND RANGE:**
  - The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function is defined.
  - For most testing, our aim has been to specify input values and to predict and/or confirm output values that result from those inputs.
  - Interface testing requires that we select the output values of the calling routine *i.e.* caller's range must be compatible with the called routine's domain.
  - An interface test consists of exploring the correctness of the following mappings:
  - `caller domain --> caller range (caller unit test)`
  - `caller range --> called domain (integration test)`
  - `called domain --> called range (called unit test)`

- **CLOSURE COMPATIBILITY:**
  - Assume that the caller's range and the called domain spans the same numbers - for example, 0 to 17.
  - Figure 4.16 shows the four ways in which the caller's range closure and the called's domain closure can agree.
  - The thick line means closed and the thin line means open. Figure 4.16 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.

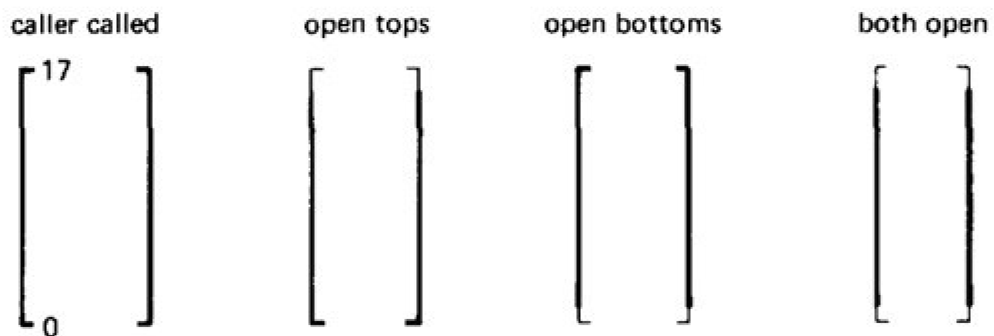


Figure 4.16: Range / Domain Closure Compatibility.

- Figure 4.17 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.

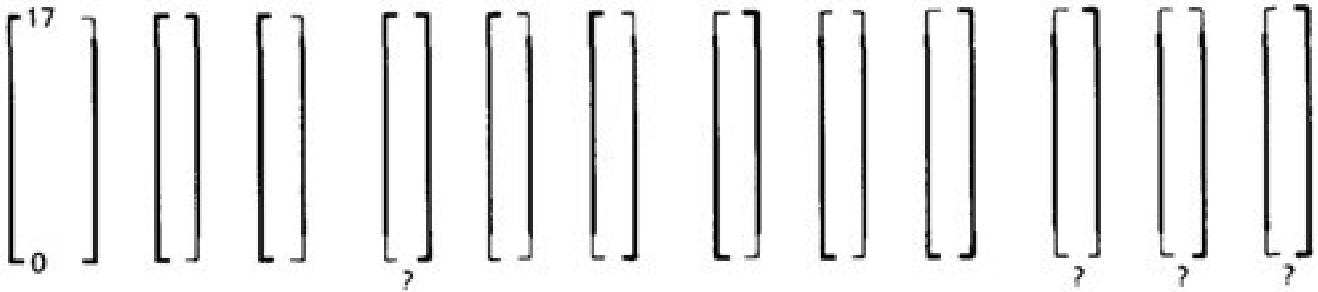


Figure 4.17: Equal-Span Range / Domain Compatibility Bugs.

- **SPAN COMPATIBILITY:**
  - Figure 4.18 shows three possibly harmless span incompatibilities.

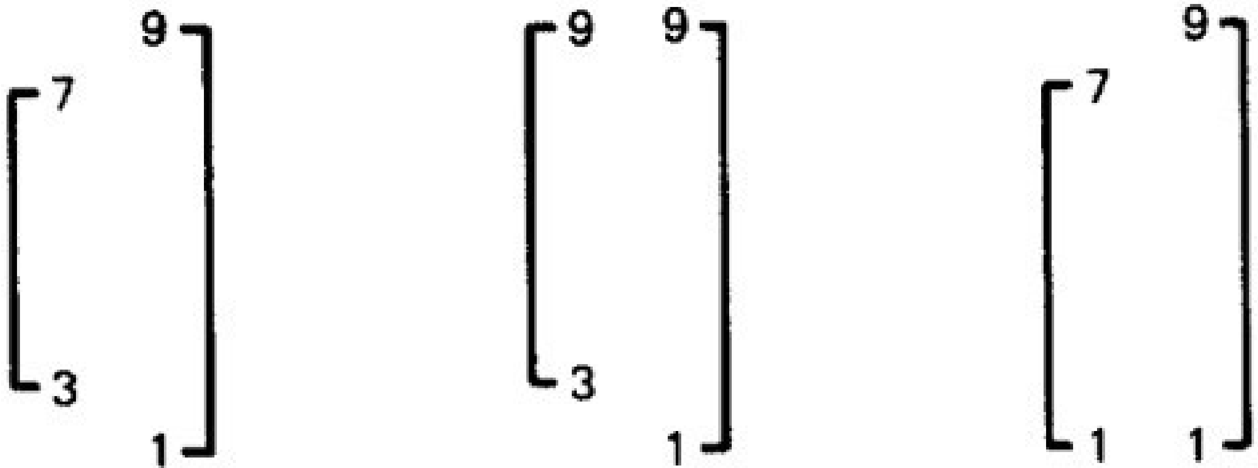


Figure 4.18: Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

- In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug.
- The routine is used by many callers; some require values inside a range and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for the caller.
- Figure 4.19a 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.

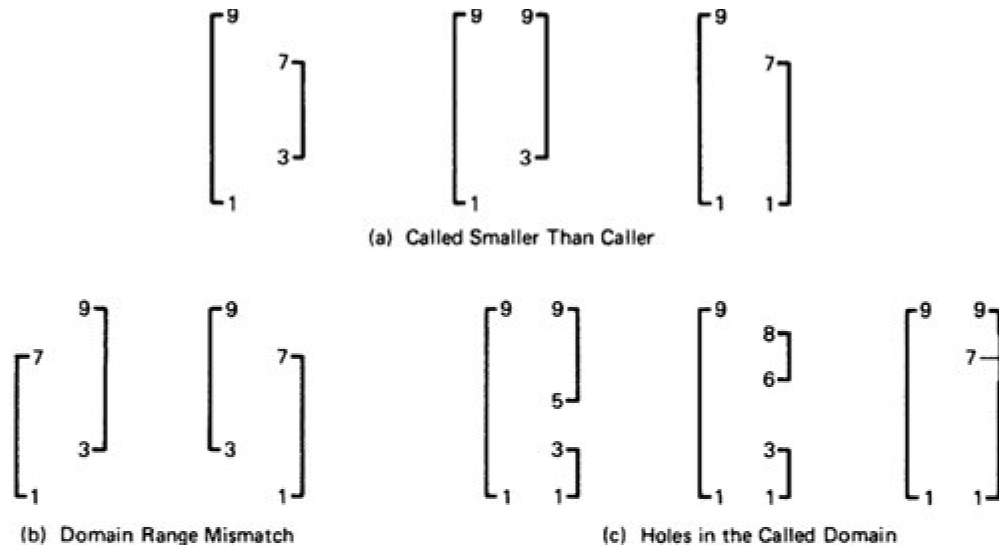


Figure 4.19: Buggy Range / Domain Mismatches

- In Figure 4.19b 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 4.19c combines these notions to show various ways we can have holes in the domain: these are all probably buggy.

- **INTERFACE RANGE / DOMAIN COMPATIBILITY TESTING:**
  - For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or more variables.
  - Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable.
  - There are two boundaries to test and it's a one-dimensional domain; therefore, it requires one on and one off point per boundary or a total of two on points and two off points for the domain - pick the off points appropriate to the closure (COOOOI).

- Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in component testing.
- Unless you're a mathematical whiz you won't be able to do this without tools for more than one variable at a time.

14)Explain Transaction flows in detail?

TRANSACTION FLOWS:

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begin with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.
- **Example of a transaction:** A transaction for an online information retrieval system might consist of the following steps or tasks:
  - Accept input (tentative birth)
  - Validate input (birth)
  - Transmit acknowledgement to requester
  - Do input processing
  - Search file
  - Request directions from user
  - Accept input
  - Validate input
  - Process request
  - Update file
  - Transmit output
  - Record transaction in log and clean up (death)

- **TRANSACTION FLOW GRAPHS:**
  - Transaction flows are introduced as a representation of a system's processing.
  - The methods that were applied to control flow graphs are then used for functional testing.
  - Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing are to the programmer.
  - The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
  - The transaction flowgraph is a model of the structure of the system's behavior (functionality).
  - An example of a Transaction Flow is as follows:

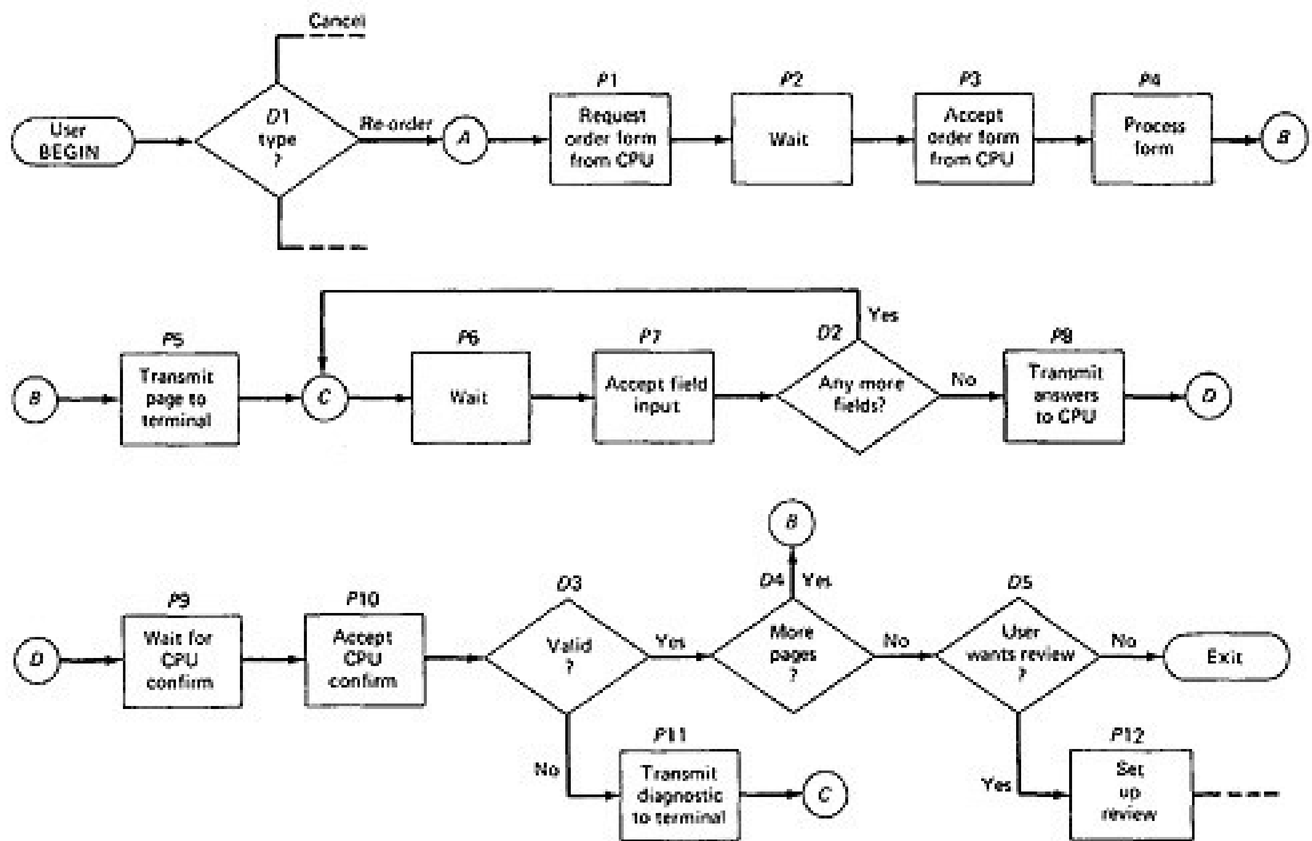


Figure 3.1: An Example of a Transaction Flow

- **USAGE:**
  - Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
  - A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows.
  - The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.
  - Loops are infrequent compared to control flowgraphs.
  - The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.
- **COMPLICATIONS:**
  - In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
  - In many systems the transactions can give birth to others, and transactions can also merge.

- **Births:** There are three different possible interpretations of the decision symbol, or nodes with two or more out links. It can be a Decision, Biosis or a Mitosis.

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

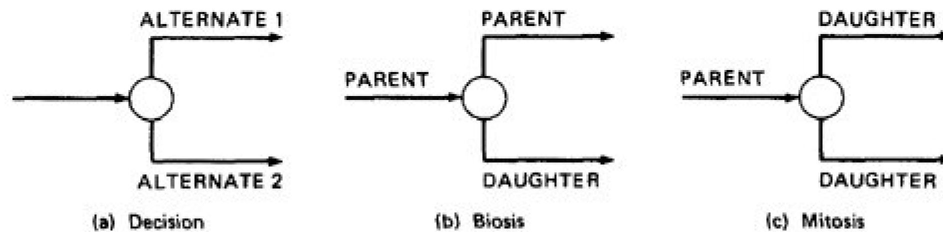


Figure 3.2: Nodes with multiple outlinks

- **Mergers:** Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation
    - **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other. (See Figure 3.3 (a))
    - **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))
    - **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.(See Figure 3.3 (c))

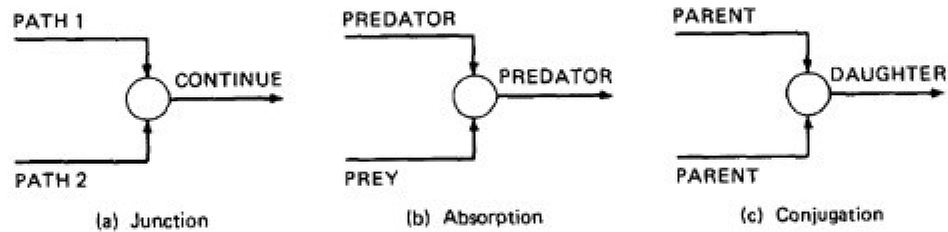


Figure 3.3: Transaction Flow Junctions and Mergers

- We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

15) what are the different kinds of bugs that arise in Software testing?

- The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and Test Design Bugs.
    - **REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS:** Various categories in Requirements, Features and Functionlity bugs include:
        1. **Requirements and Specifications Bugs:**
            - Requirements and specifications developed from them can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.
            - The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted.
            - Requirements, especially, as expressed in specifications are a major source of expensive bugs.
            - The range is from a few percentage to more than 50%, depending on the application and environment.
            - What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.
        2. **Feature Bugs:**

- Specification problems usually create corresponding feature problems.
- A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect and correct. A wrong feature could have deep design implications.
- Removing the features might complicate the software, consume more resources, and foster more bugs.

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

Specification and Feature Bug Remedies:

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

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

- **STRUCTURAL BUGS:** Various categories in Structural bugs include:

1. **Control and Sequence Bugs:**
   - Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, sphagetti code, and worst of all, pachinko code.
   - One reason for control flow bugs is that this area is amenable (supportive) to theoritical treatment.
   - Most of the control flow bugs are easily tested and caught in unit testing.
   - Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.
   - Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.
2. **Logic Bugs:**
   - Bugs in logic, especially those related to misundertanding how case statements and logic operators behave singly and combinations
   - Also includes evaluation of boolean expressions in deeply nested IF-THEN-ELSE constructs.
   - If the bugs are parts of logical (i.e. boolean) processing not related to control flow, they are characterized as processing bugs.
   - If the bugs are parts of a logical expression (i.e control-flow statement) which is used to direct the control flow, then they are categorized as control-flow bugs.
3. **Processing Bugs:**
   - Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
   - Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of grater-than-or-eual etc
   - Although these bugs are frequent (12%), they tend to be caught in good unit testing.
4. **Initialization Bugs:**
   - Initialization bugs are common. Initialization bugs can be improper and superfluous.
   - Superfluous bugs are generally less harmful but can affect performance.
   - Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc

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

5. **Data-Flow Bugs and Anomalies:**
   - Most initialization bugs are special case of data flow anamolies.
   - A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

- **DATA BUGS:**

  - 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 atleast as common as bugs in code, but they are foten treated as if they didnot exist at all.
  - *Code migrates data:* Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.
  - Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equal attention.
  - **Dynamic Data Vs Static data:**
    - Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes and residues.
    - Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Clean up
    - Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bit pattern.
    - Compile time processing will solve the bugs caused by static data.
  - **Information, parameter, and control:** Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.
  - **Content, Structure and Attributes: Content** can be an actual bit pattern, character string, or number put into a data

structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content. **Structure** relates to the size, shape and numbers that describe the data object, that is memory location used to store the content. (e.g A two dimensional array). **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (e.g. an integer, an alphanumeric string, a subroutine). *The severity and subtlelty of bugs increases as we go from content to attributes because the things get less formal in that direction.*

- **CODING BUGS:**
  - Coding errors of all kinds can create any of the other kind of bugs.
  - Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
  - If a program has many syntax errors, then we should expect many logic and coding bugs.
  - The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.

- **INTERFACE, INTEGRATION, AND SYSTEM BUGS:**
  - Various categories of bugs in Interface, Integration, and System Bugs are:
    - **External Interfaces:**
      - The external interfaces are the means used to communicate with the world.
      - These include devices, actuators, sensors, input terminals, printers, and communication lines.
      - The primary design criterion for an interface with outside world should be robustness.
      - All external interfaces, human or machine should employ a protocol. The protocol may be wrong or incorrectly implemented.
      - Other external interface bugs are: invalid timing or sequence assumptions related to external signals
      - Misunderstanding external input or output formats.
      - Insufficient tolerance to bad input data.

- **Internal Interfaces:**
    - Internal interfaces are in principle not different from external interfaces but they are more controlled.
    - A best example for internal interfaces are communicating routines.
    - The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.
    - Internal interfaces have the same problem as external interfaces.
- **Hardware Architecture:**
    - Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
    - Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
    - The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.
- **Operating System Bugs:**
    - Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does.
    - Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls.
    - This approach may not eliminate the bugs but at least will localize them and make testing easier.
- **Software Architecture:**
    - Software architecture bugs are the kind that called - interactive.
    - Routines can pass unit and integration testing without revealing such bugs.
    - Many of them depend on load, and their symptoms emerge only when the system is stressed.

- Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
- Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

- **Control and Sequence Bugs (Systems Level):**
  - These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.
  - The remedy for these bugs is highly structured sequence control.
  - Specialize, internal, sequence control mechanisms are helpful.
- **Resource Management Problems:**
  - Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.
  - External mass storage units such as discs, are subdivided into memory resource pools.
  - Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc
  - **Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discovers them.
  - The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.
- **Integration Bugs:**
  - Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
  - These bugs results from inconsistencies or incompatibilities between components.

- The communication methods include data structures, call sequences, registers, semaphores, communication links and protocols results in integration bugs.
- The integration bugs do not constitute a big bug category(9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.

- **System Bugs:**
  - System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.
  - There can be no meaningful system testing until there has been thorough component and integration testing.
  - System bugs are infrequent(1.7%) but very important because they are often found only after the system has been fielded.

- **TEST AND TEST DESIGN BUGS:**

  - Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.
  - They require code or the equivalent to execute and consequently they can have bugs.
  - Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavior is judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.
  - **Remedies:** The remedies of test bugs are:

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

simpler than programs and donot have to make concessions to efficiency.

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