UNIT II
TRANSACTION FLOW TESTING AND DATA FLOW TESTING

INTRODUCTION:

- 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 begins 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:

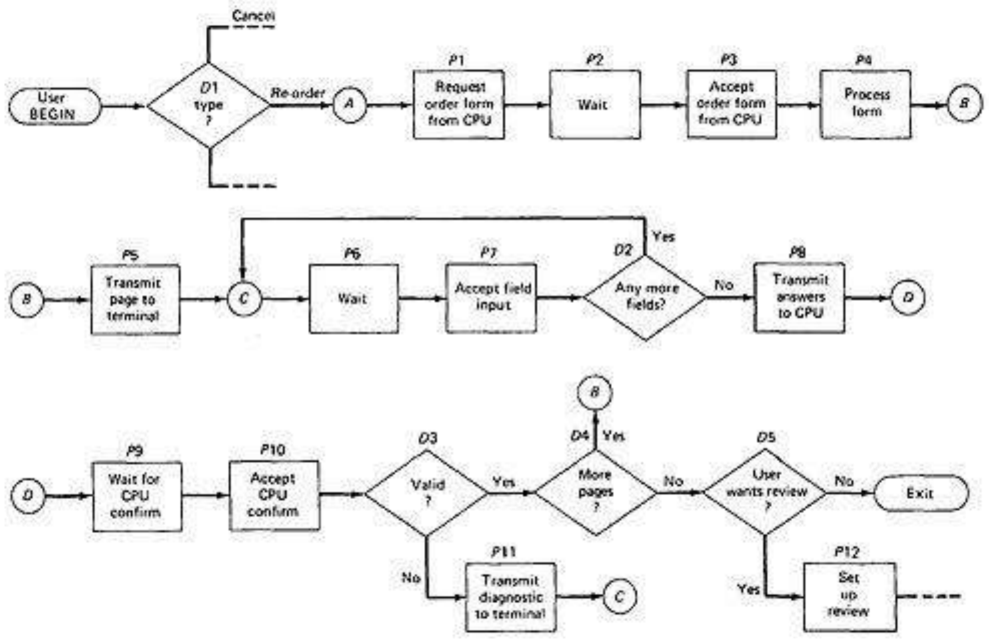


Figure 3.1: An Example of a Transaction Flow

- **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.
    1. **Decision:** Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2 (a))
    2. **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))
    3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created.(See Figure 3.2 (c))

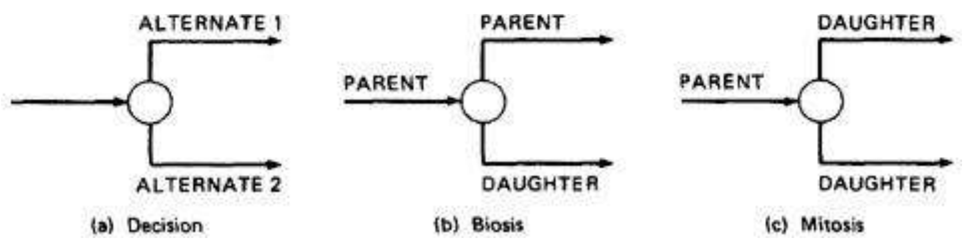


Figure 3.2: Nodes with multiple outlinks

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

1 **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))
2 **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))
3 **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))

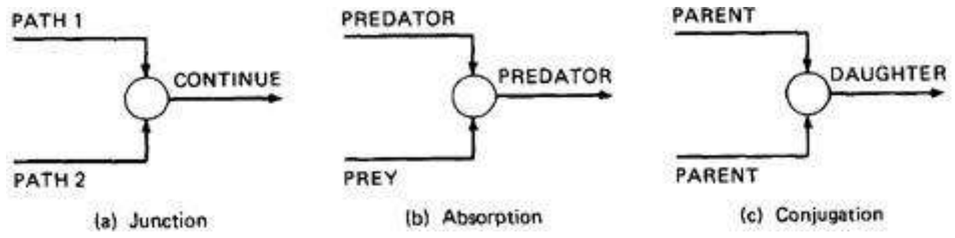


Figure 3.3: Transaction Flow Junctions and Mergers

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.

TRANSACTION FLOW TESTING TECHNIQUES:

- **GET THE TRANSACTIONS FLOWS:**
  - Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
  - Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
  - The system's design documentation should contain an overview section that details the main transaction flows.
  - Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

- **INSPECTIONS, REVIEWS AND WALKTHROUGHS:**
  - Transaction flows are natural agenda for system reviews or inspections.
  - 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.
    - Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
    - Select additional flow paths for loops, extreme values, and domain boundaries.
    - Design more test cases to validate all births and deaths.
    - 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.

- **PATH SELECTION:**
    - Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
    - Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
    - Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

- **PATH SENSITIZATION:**
    - Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
    - The remaining small percentage is often very difficult.
    - 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.

- **PATH INSTRUMENTATION:**
    - Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
    - The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
    - In some systems, such traces are provided by the operating systems or a running log.
- **DESIGN AND MAINTAIN TEST DATABASE**
    - Design and maintenance of the test databases constitute about 30% to 40% of the effort in transaction flow test design.
    - People are often unaware that a test database needs to designed.
    - Test databases must be centrally administrated and configuration controlled with a comprehensive design plan.
    - Creating a comprehensive test databases is itself a big project on its own.
    - It requires talented, matured, and diplomatic designers who are experienced in both system design and test design.
- **TEST EXECUTION**
    - Commit to automation of test execution if you want to do transaction flow testing for a system of any size.
    - If the numbers of test cases are limited, you need not worry about test execution automation.
    - If the number of test cases run into several hundred, performing transaction flow testing to achieve ($C_1 + C_2$) needs execution automation without which

you cannot get it right.

BASICS OF DATA FLOW TESTING:

- **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.
    - **Motivation:** It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should

not feel confident about a program without having seen the effect of using the value produced by each and every computation.

- **DATA FLOW MACHINES:**
    - There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines (MIMD).
    - **Von Neumann Machine Architecture:**
        - Most computers today are von-neumann machines.
        - This architecture features interchangeable storage of instructions and data in the same memory units.
        - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
            - Fetch instruction from memory
            - Interpret instruction
            - Fetch operands
            - Process or Execute
            - Store result
            - Increment program counter
            - 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 flow graphs as such. Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at the moment.

- **DATA FLOW GRAPHS:**
    - The data flow graph is a graph consisting of nodes and directed links.
    - We will use a control graph to show what happens to 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.

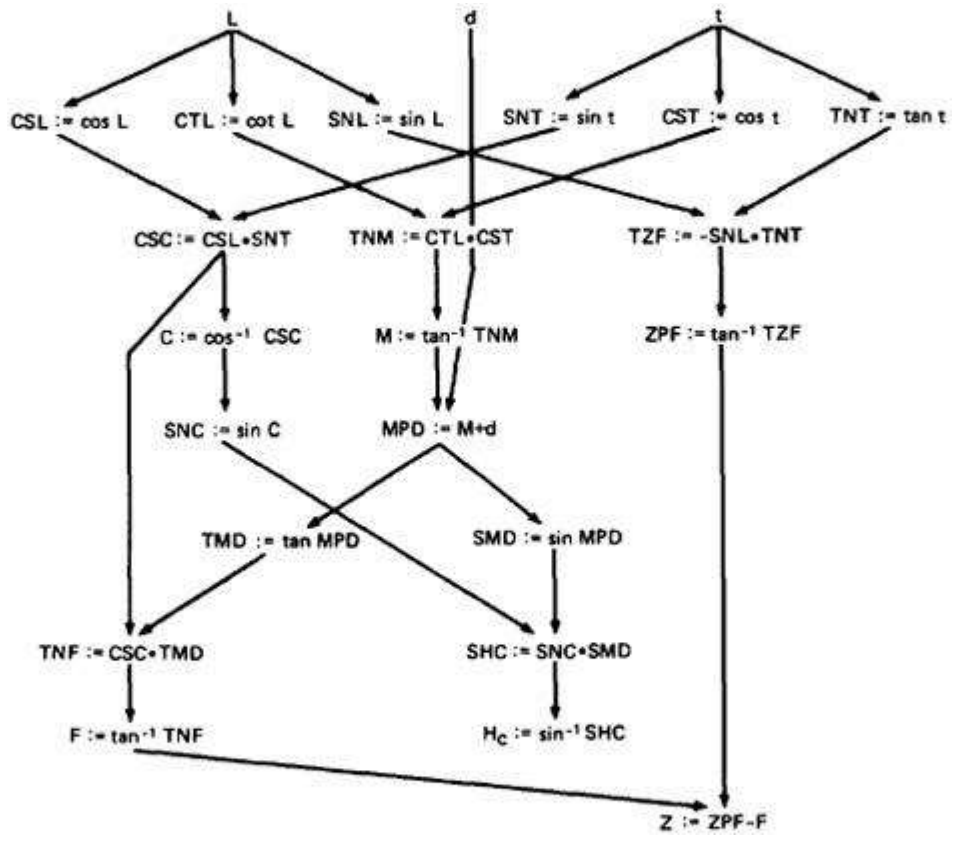


Figure 3.4: Example of a data flow graph

- **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 absolute certainty / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A

3. Usage (u):

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

DATA FLOW ANOMALIES:

An anomaly is denoted by a two-character sequence of actions. For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.

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

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

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

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

They possible anomalies are:

1. **-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.
2. **-d** :- okay. This is just the first definition along this path.
3. **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.

4 **k-** :- not anomalous. The last thing done on this path was to kill the variable.
5 **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
6 **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.

DATA FLOW ANOMALY STATE GRAPH:

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

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

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.

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

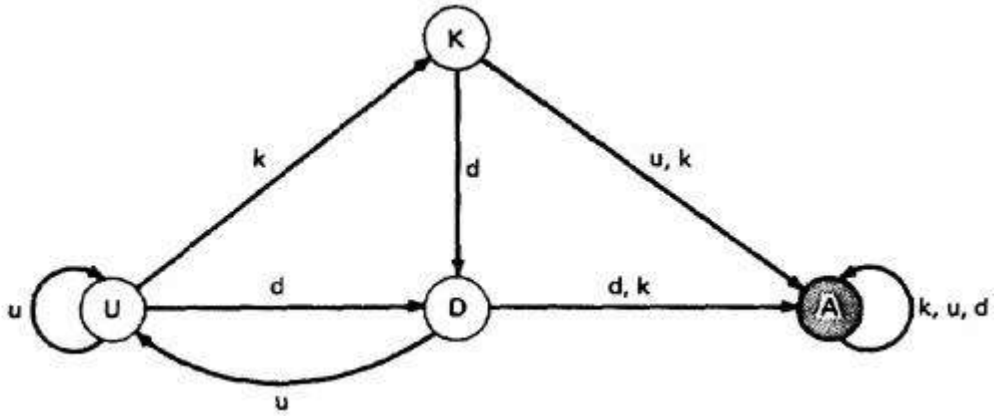

$$\begin{tikzcd} & & K \arrow[dl, "d"] \arrow[drr, "{u, k}"] & & \\ U \arrow[loop left, "u"] \arrow[r, "d"] \arrow[urr, "k"] & D \arrow[l, bend left, "u"] \arrow[rrr, "{d, k}"] & & & A \arrow[loop right, "{k, u, d}"] \end{tikzcd}$$

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.

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

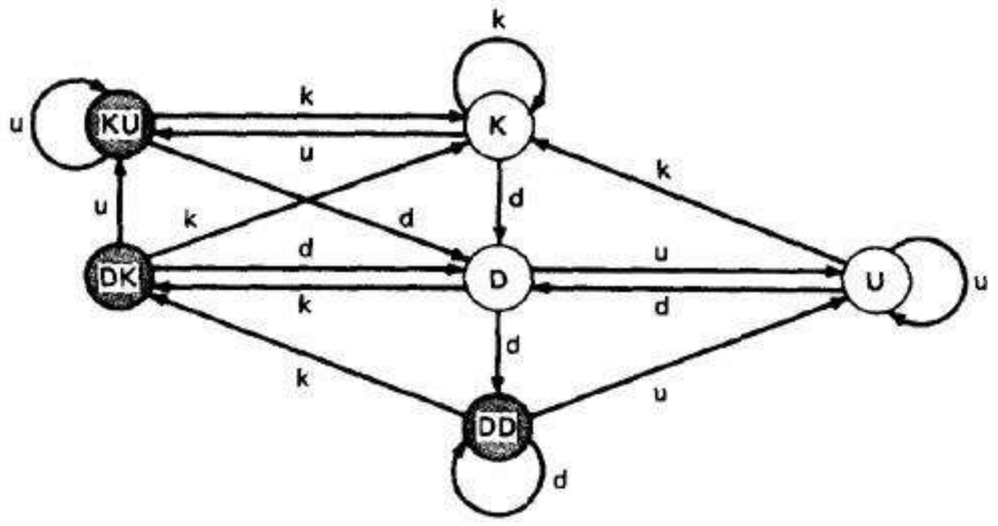


Figure 3.6: Forgiving Data Flow Anomaly State Graph

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.

STATIC Vs DYNAMIC ANOMALY DETECTION:

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesn’t belongs in testing - it belongs in the language processor.
There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.But still there are many things for which current notions of static analysis are INADEQUATE.

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 pseudo concurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

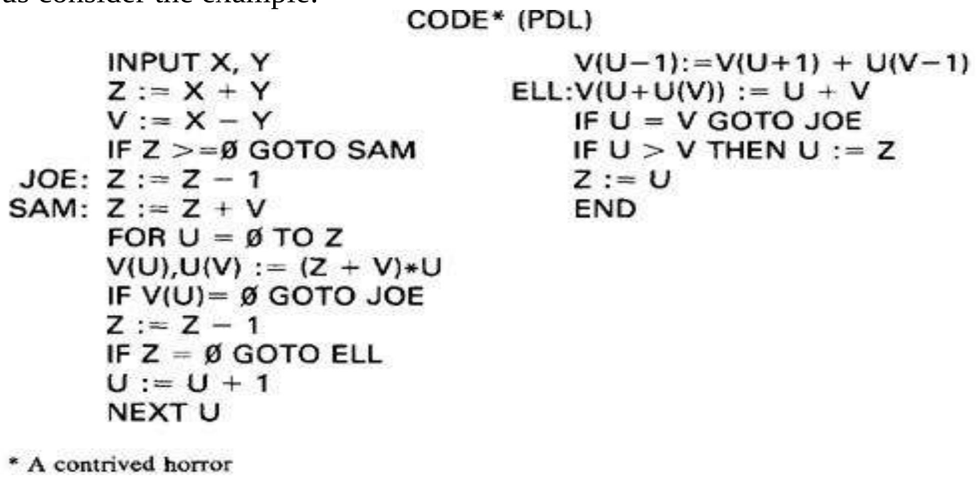
  Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do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

DATA FLOW MODEL:

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.
Here we annotate each link with symbols (for example, d, k, u, c, and p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.
The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

1. To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.
3. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.
4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
5. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.
   Let us consider the example:



CODE* (PDL)

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

* A contrived horror

Figure 3.7: Program Example (PDL)

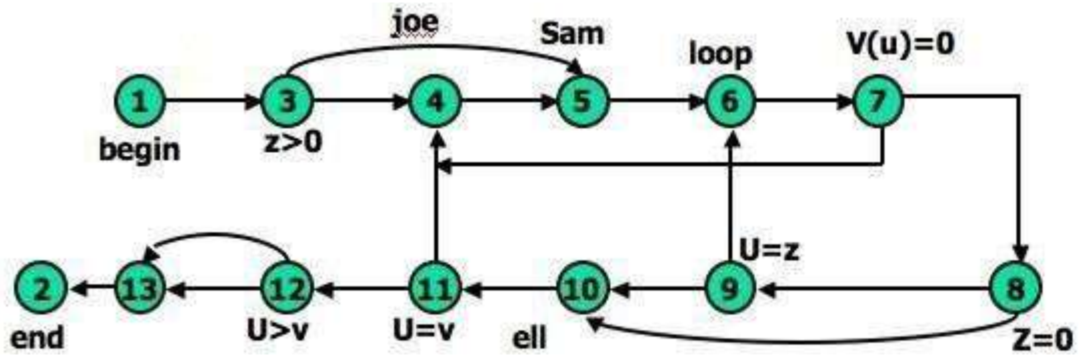


Figure 3.8: Unannotated flow graph for example program in Figure 3.7

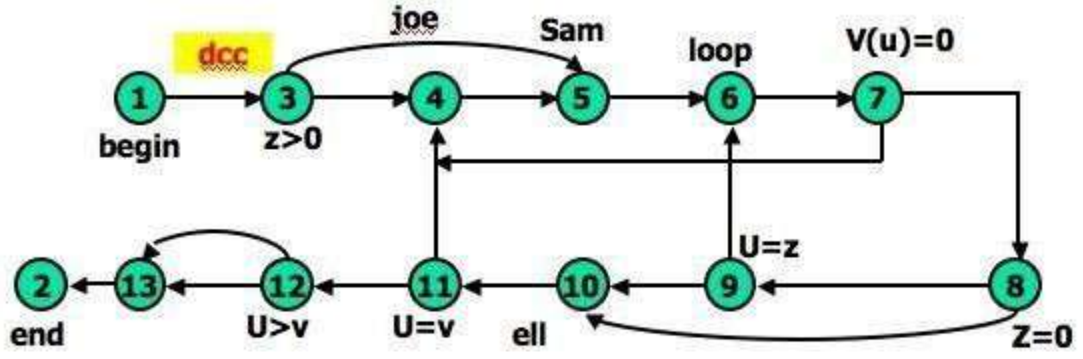


Figure 3.9: Control flow graph annotated for X and Y data flows.

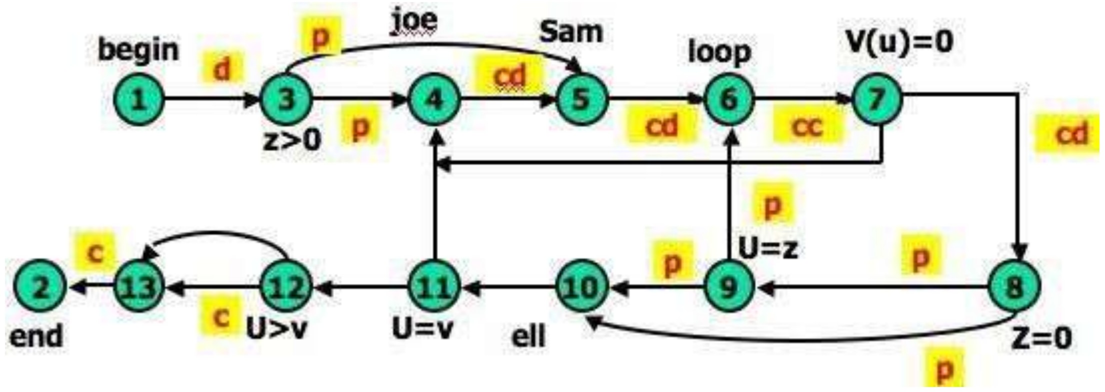


Figure 3.10: Control flow graph annotated for Z data flow.

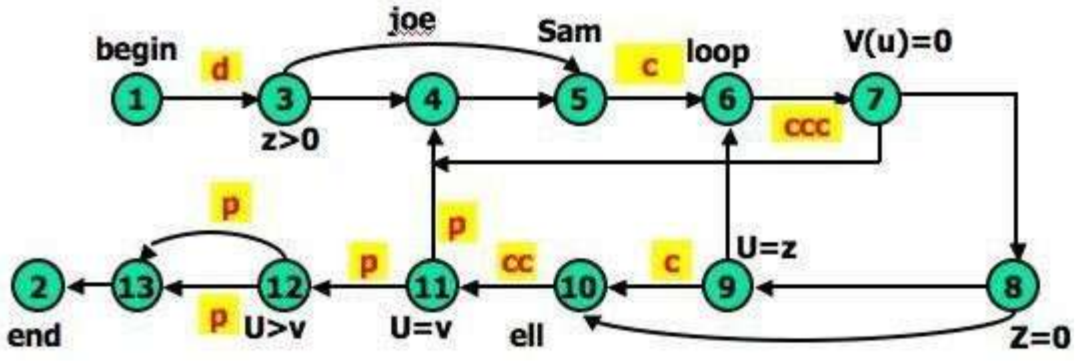


Figure 3.11: Control flow graph annotated for V data flow.

STRATEGIES IN DATA FLOW TESTING:

- **INTRODUCTION:**
    - 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 sub paths 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.

STRATEGIES: The structural test strategies discussed below are based on the program's control flow graph. 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:

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 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.

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.

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.

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.

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.

1. **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.

ORDERING THE STRATEGIES:

Figure 3.12compares 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

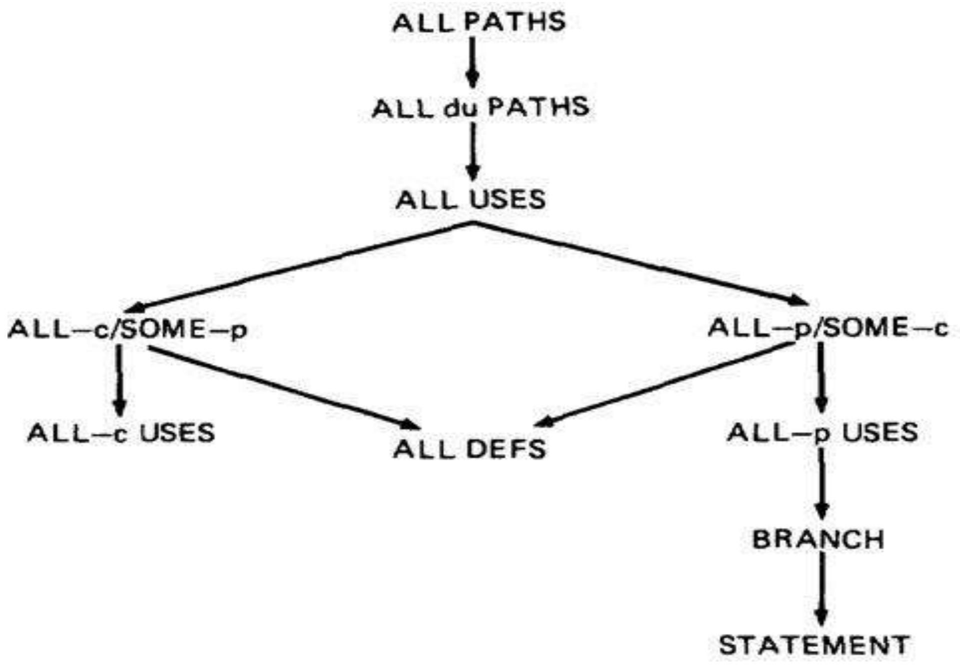


Figure 3.12: Relative Strength of Structural Test Strategies.

- 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.

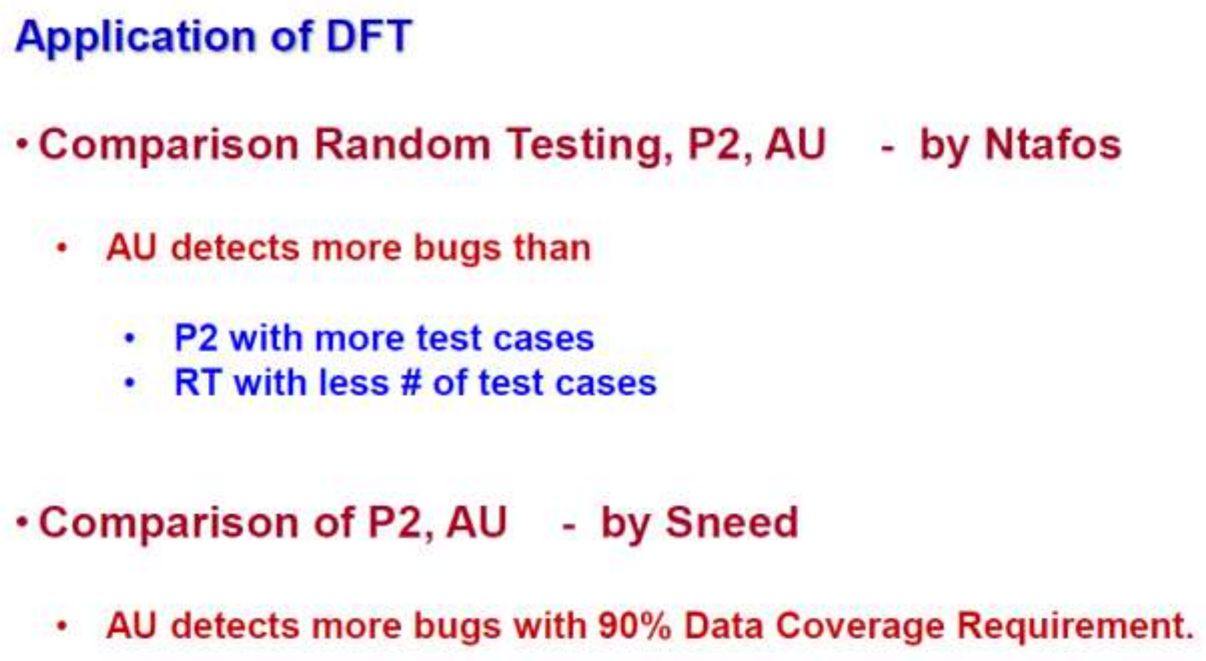
SLICING AND DICING:

- 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.

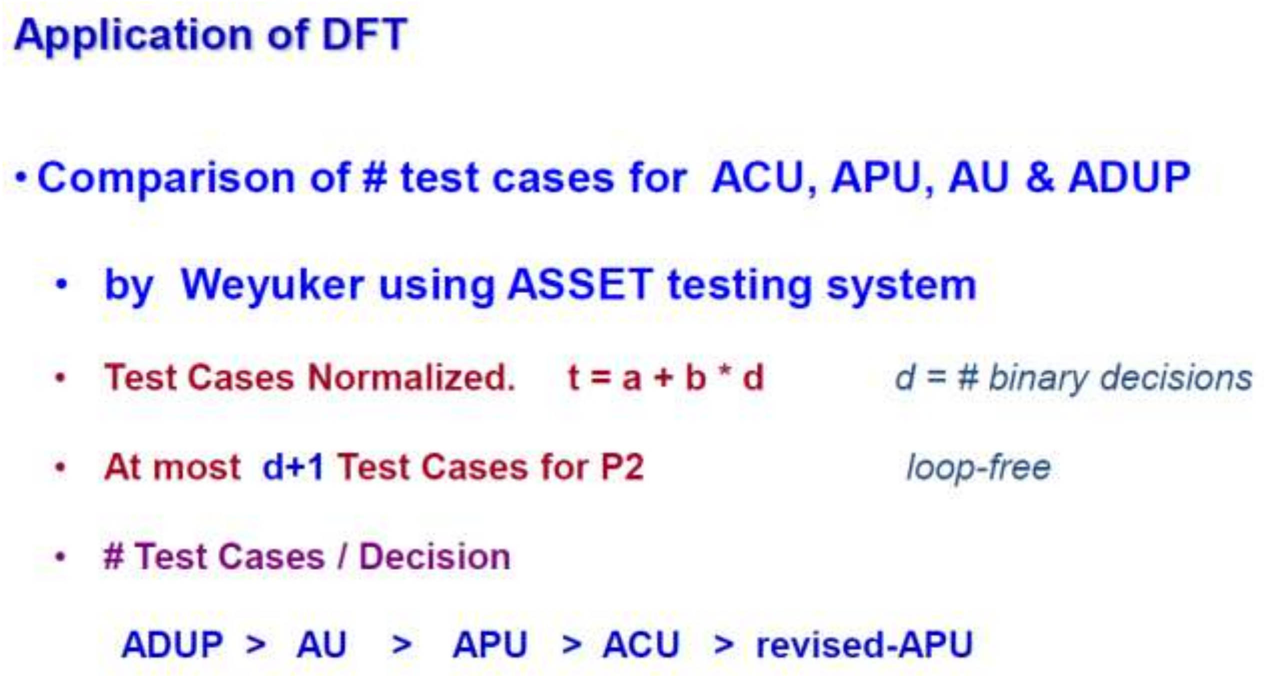
- **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.
- Slicing methods bring together testing, maintenance & debugging.

application of dataflow testing



Application of DFT

- Comparison Random Testing, P2, AU - by Ntafos
  - AU detects more bugs than
    - P2 with more test cases
    - RT with less # of test cases
- Comparison of P2, AU - by Sneed
  - AU detects more bugs with 90% Data Coverage Requirement.



Application of DFT

- Comparison of # test cases for ACU, APU, AU & ADUP
  - by Weyuker using ASSET testing system
  - Test Cases Normalized. t = a + b * d *d = # binary decisions*
  - At most d+1 Test Cases for P2 *loop-free*
  - # Test Cases / Decision

ADUP > AU > APU > ACU > revised-APU

Application of DFT

Comparison of # test cases for ACU, APU, AU & ADUP by Shimeall & Levenson

Test Cases Normalized. $t = a + b * d$ *(d = # binary decisions)*

At most d+1 Test Cases for P2 *loop-free*

Test Cases / Decision

ADUP ~ ½ APU*

AP ~ AC

Application of DFT

DFT vs P1, P2

- DFT is Effective
- Effort for Covering Path Set ~ Same
- DFT Tracks the Coverage of Variables
- Test Design is similar

DFT - TOOLS

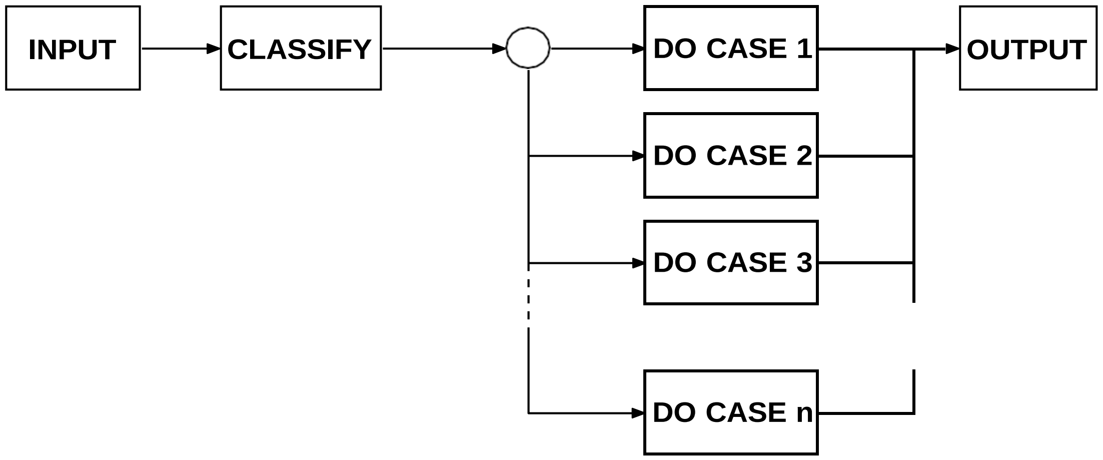
- Cost-effective development
- Commercial tools :
- Can possibly do Better than Commercial Tools
  - Easier Integration into a Compiler
  - Efficient Testing

DOMAIN TESTING

(1) Domains and paths:

(i) The Model:

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



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

(ii) A domain is a set:

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

(iii) Domains, paths and predicates:

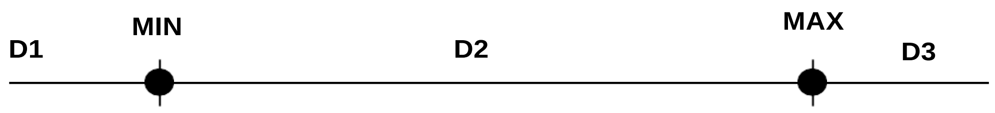
- In domain testing, predicates are assumed to be interpreted in terms of input vector variables.
- If domain testing is applied to structure (implementation), then predicate interpretation must be based on control flowgraph.
- If domain testing is applied to specifications, then predicate interpretation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

Review:

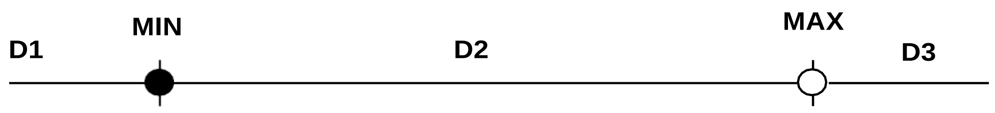
1. A domain is a loop free program.
2. For every domain there is at least one path through the routine.
3. The set of interpreted predicates defines the domain boundaries.

(iv) Domain Closure:

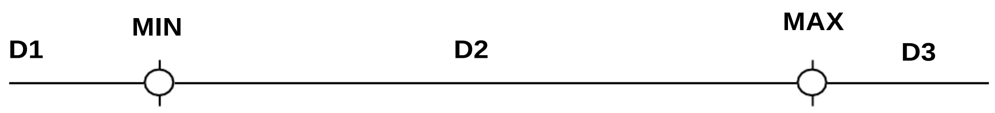
➢ To understand the domain closure, consider the following figure.



(a) Both side closed



(b) One side open



(c) Both side open

➢ If the domain boundary point belongs to the same domain then the boundary is said to close. If the domain boundary point belongs to some other domain then the boundary is said to open.

➢ In the above figure there are three domains D1, D2, D3.

➢ In **figure a** D2's boundaries are closed both at the minimum and maximum values. If D2 is closed, then the adjacent domains D1 and D3 must be open.

➢ In **figure b** D2 is closed on the minimum side and open on the maximum side, meaning that D1 is open and D3 is closed. In **figure c** D2 is open on both sides, which mean that the adjacent domains D1 and D3 must be closed.

(v) Domain Dimensionality:

➢ Depending on the input variables, the domains can be classified as number line domains, planer domains or solid domains.

➢ That is for one input variable the value of the domain is on the number line, for two variables the resultant is planer and for three variables the domain is solid.

➢ One important thing here is to note that we need not worry about the domains dimensionality with the number of predicates. Because there might be one or more boundary predicates.

(vi) The Bug Assumptions:

➢ The bug assumption for domain testing is that processing is okay but the domain definition is wrong.

➢ An incorrectly implemented domain means that boundaries are wrong, which mean that control-flow predicates are wrong.

➢ The following are some of the bugs that give to domain errors.

(a) Double-Zero Representation:

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

(b) Floating-Point Zero Check:

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

(c) Contradictory Domains:

- ❖ Here at least two assumed distinct domains overlap.

(d) Ambiguous Domains:

- ❖ These are missing domain, incomplete domain.

(e) Over specified Domains:

- ❖ The domain can be overloaded with so many conditions.

(f) Boundary Errors:

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

(g) Closure Reversal

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

(h) Faulty Logic:

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

(vii) Restrictions:

(a) General

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

(b) Coincidental Correctness

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

(c) Representative Outcome

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

(d) Simple Domain Boundaries and Compound Predicates

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

(e) Functional Homogeneity of Bugs

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

(f) Linear Vector Space

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

(g) Loop-free Software

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

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

(2) Nice Domains:

(i) Where Do Domains Come From?

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

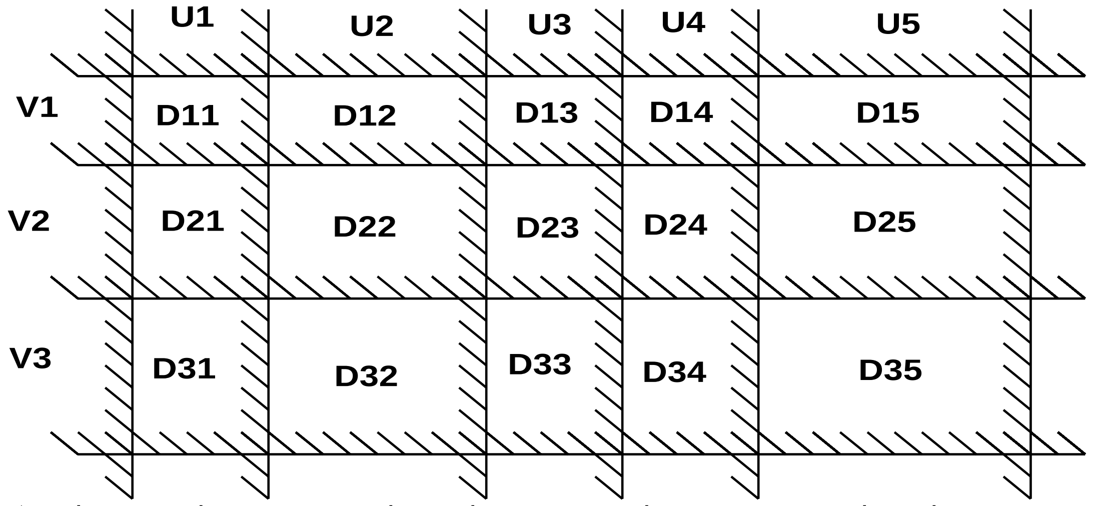
(ii) Specified versus Implemented Domains:

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

(iii) Nice Domains:

(1) General

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



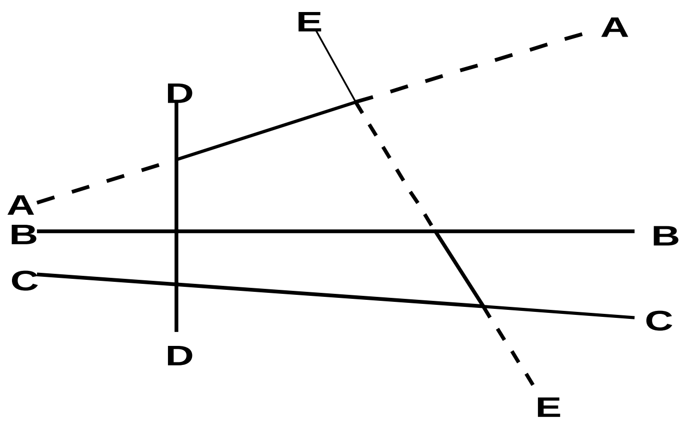
- ❖ The U and V represent boundary sets and D represents domains.
- ❖ The boundaries have several important properties. They are linear, complete, systematic, orthogonal, consistently closed, simply connected and convex.
- ❖ If domains have these properties, domain testing is very easy otherwise domain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

(2) Linear and Nonlinear Boundaries

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

(3) Complete Boundaries

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



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

(4) Systematic Boundaries

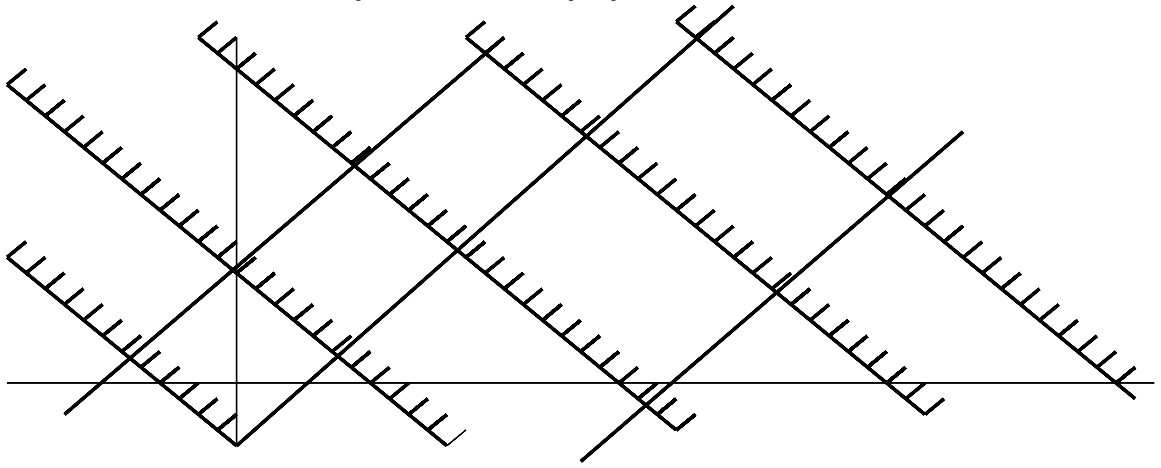
- Systematic boundaries refer to boundary inequalities with simple mathematical functions such as a constant.
- Consider the following relations,

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

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

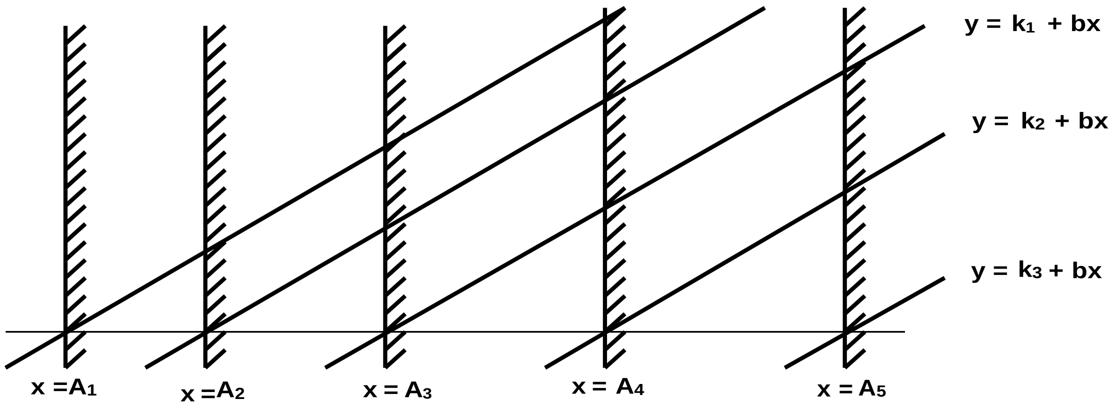
(5) Orthogonal Boundaries

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



(6) Closure Consistency

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



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

(7) Convex

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

(8) Simply Connected

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

(iv) Ugly Domains:

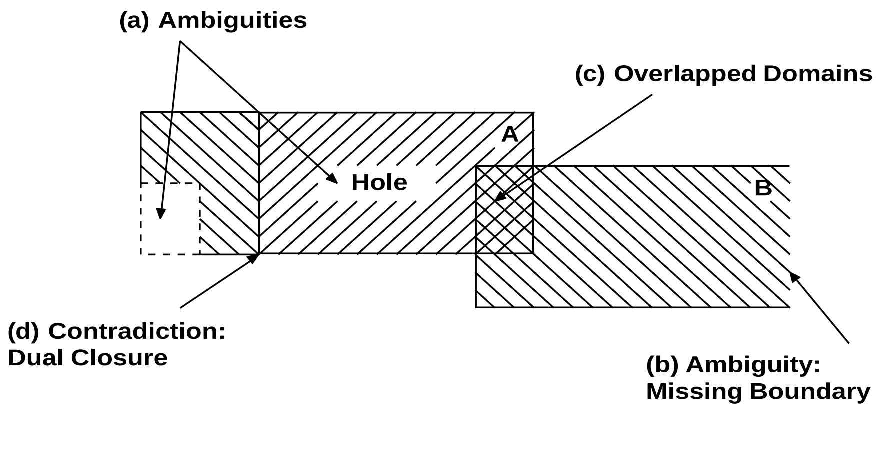
(a) General

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

(b) Nonlinear Boundaries

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

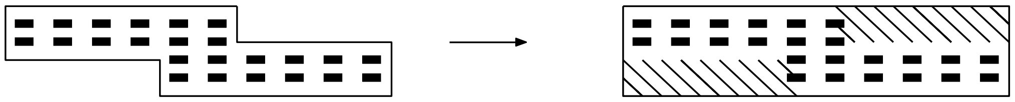
(c) Ambiguities and Contradictions:.



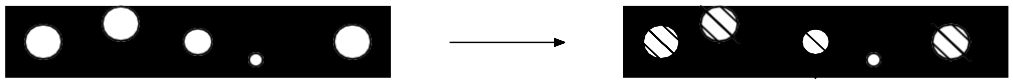
- ❖ Domain ambiguity is missing or incomplete domain boundary.
- ❖ In the above figure Domain ambiguities are holes in the A domain and missing boundary in the B domain.
- ❖ An ambiguity for one variable can be see easy.
- ❖ An ambiguity for two variables can be difficult to spot.
- ❖ An ambiguity for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

(d) Simplifying the Topology

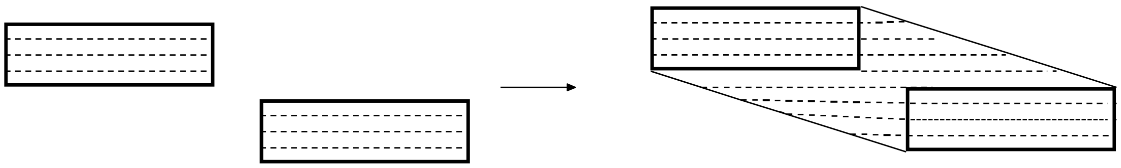
- ❖ Connecting disconnected boundary segments and extending boundaries is called simplifying the topology
- ❖ There are three generic cases of simplifying the topology.



(a) Making it convex



(b) Filling in the Holes

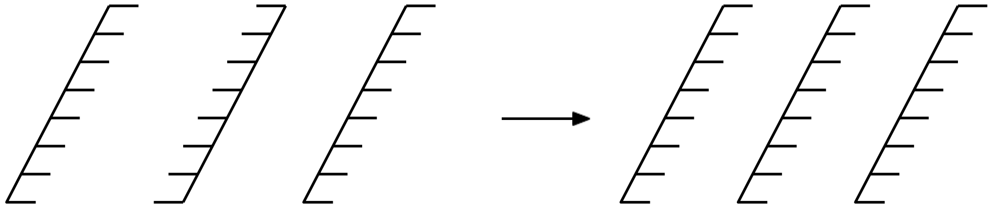


(c) Joining the Pieces

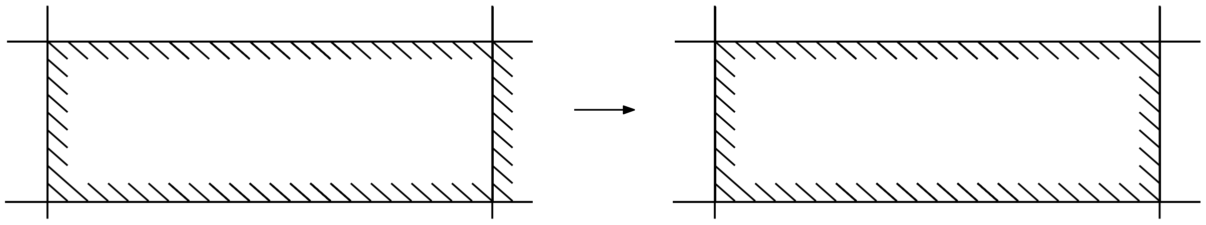
- ❖ Programmers introduce bugs and testers misdesign test cases by, smoothing out concavities, filling in holes, joining disconnected pieces.

(e) Rectifying Boundary Closures

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



(a) Consistent Direction



(b) Inclusion/Exclusion Consistency

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

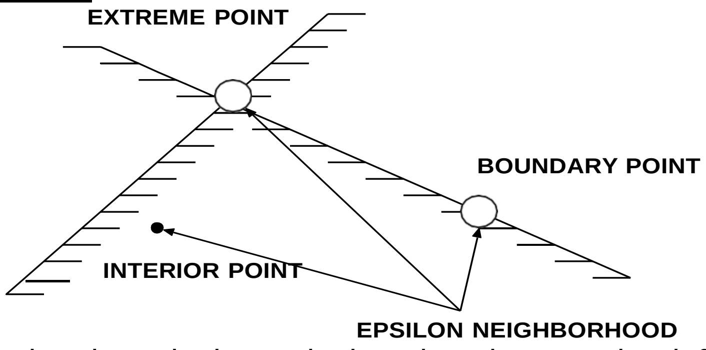
(3) Domain Testing:

(i) Overview:

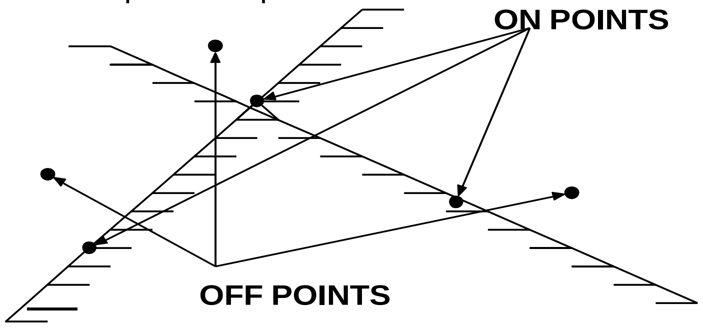
- ➢ Domains are defined by their boundaries. So domain testing concentrates test points on boundaries or near boundaries.
- ➢ Find what wrong with boundaries, and then define a test strategy.
- ➢ Because every boundary uses at least two different domains, test points used to check one domain can also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

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

(a) General:

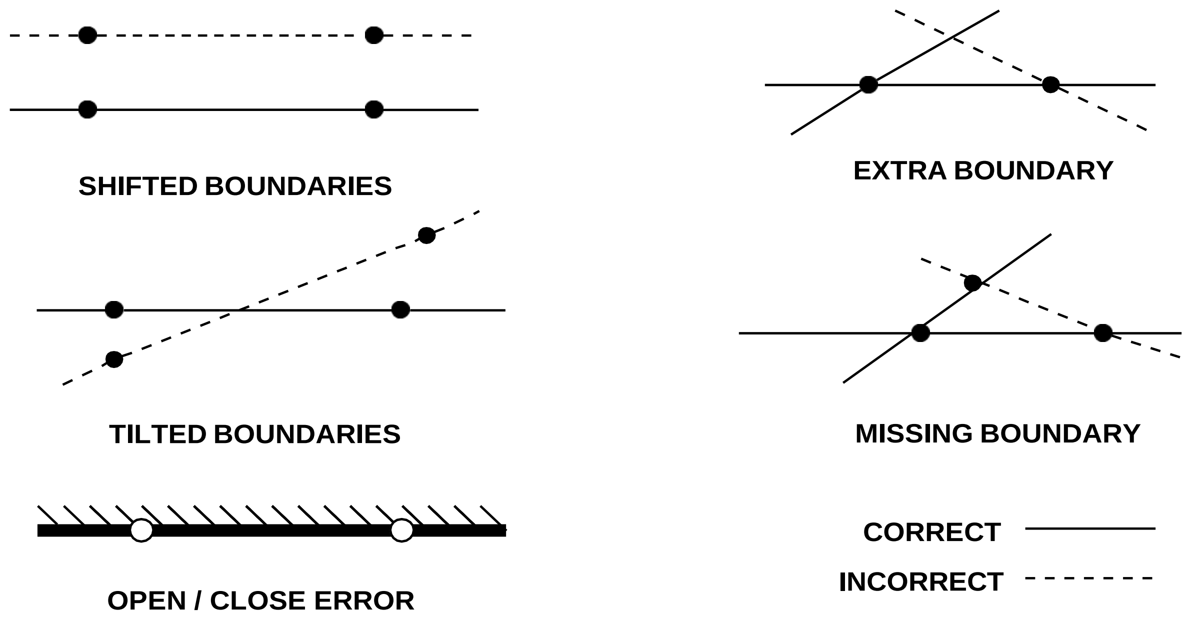


- ❖ An interior point is a point in a domain. It can be defined as a point which specifies certain distance covered by some other points in the same domain.
- ❖ This distance is known as epsilon neighborhood.
- ❖ A boundary point is on the boundary that is a point with in a specific epsilon neighborhood.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



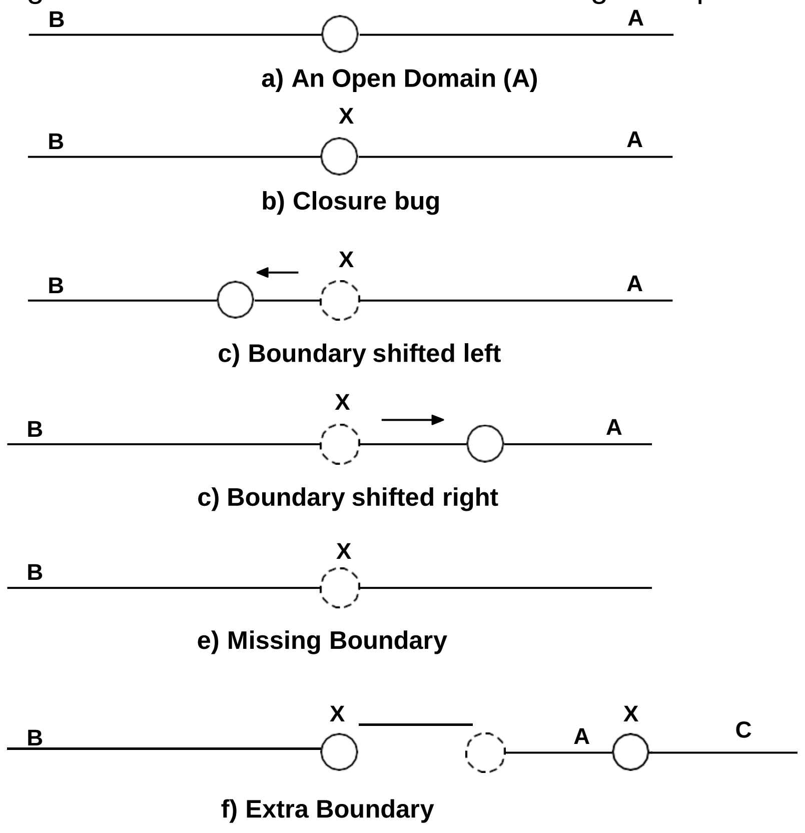
- ❖ An on point is a point on the boundary. An off point is outside the boundary.
- ❖ If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.

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

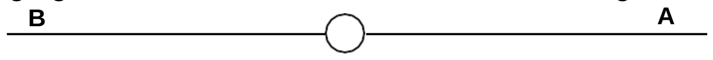


(b) Testing One-Dimensional Domains:

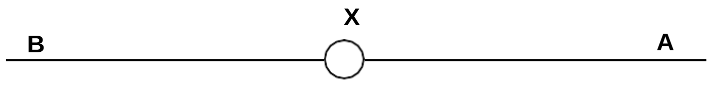
- ❖ The following figure shows one dimensional domain bugs for open boundaries.



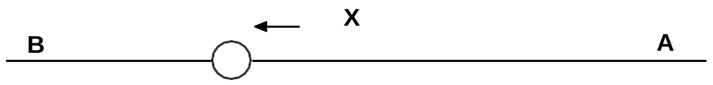
- ❖ In the above figure a) we assume that the boundary was to open for A.
- ❖ In figure b) one test point (marked X) on the boundary detects the bug.
- ❖ In figure c) a boundary shifts to left.
- ❖ In figure d) a boundary shifts to right.
- ❖ In figure e) there is a missing boundary. In figure f) there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



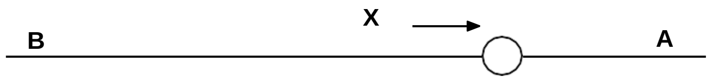
a) A closed Domain (A)



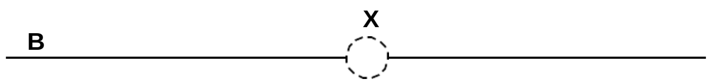
b) Closure bug



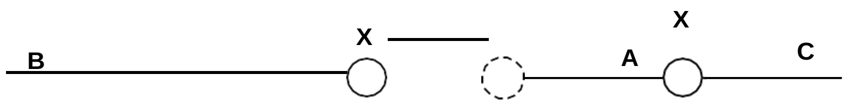
c) Boundary shifted left



c) Boundary shifted right



e) Missing Boundary



f) Extra Boundary

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

(c) Testing Two-Dimensional Domains:

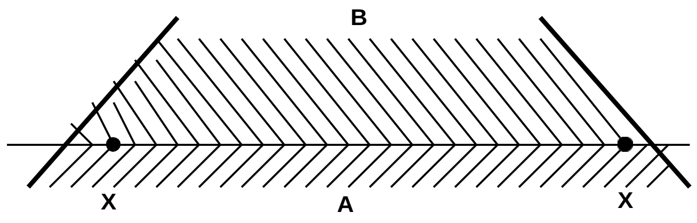
- ➢ The following figure shows domain boundary bugs for two dimensional domains.
- ➢ A and B are adjacent domains, and the boundary is closed with respect to A and the boundary is opened with respect to B.

(i) Closure Bug:

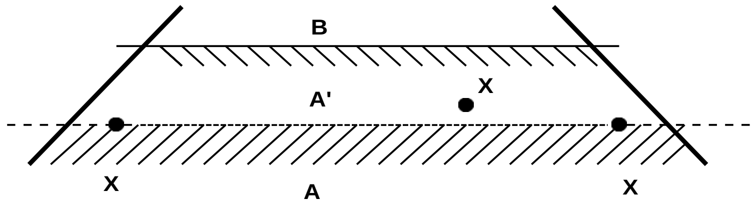
- ❖ The figure (a) shows a wrong closure, that is caused by using a wrong operator for example, x>=k was used when x > k was intended.
- ❖ The two on points detect this bug.

(ii) Shifted Boundary:

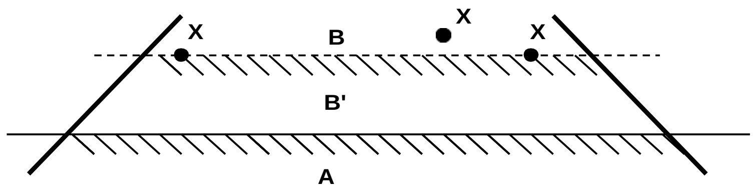
- ❖ In figure (b) the bug is shifted up, which converts part of domain B into A'.
- ❖ This is caused by incorrect constant in a predicate for example x + y >= 17 was used when x + y > = 7 was intended. Similarly figure (c) shows a shift down.



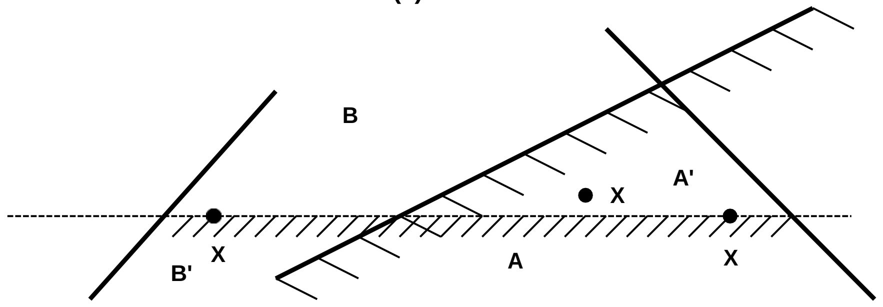
(a) Closure Bug



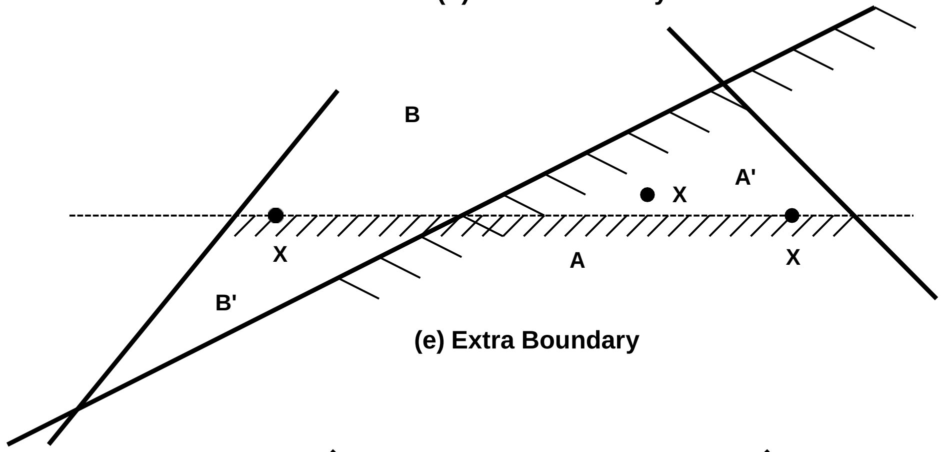
(b) Shifted Up



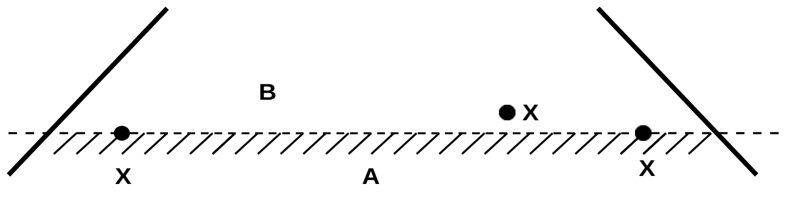
(c) Shifted Down



(d) Tilted Boundary



(e) Extra Boundary



(f) Missing Boundary

(iii) Tilted boundary:

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

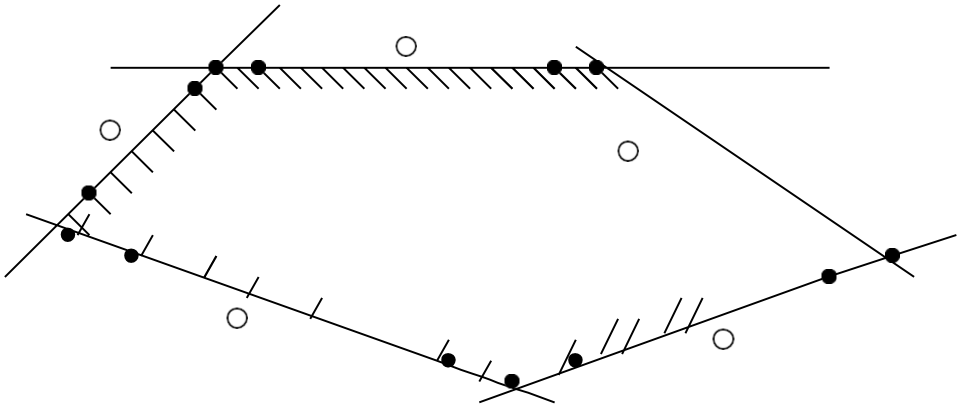
(iv) Extra Boundary:

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

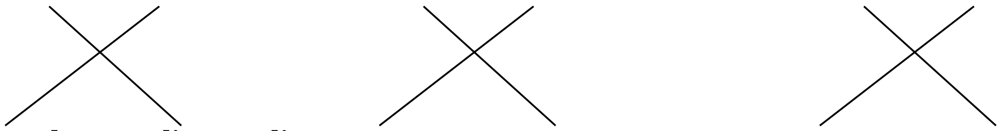
(v) Missing Boundary:

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

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

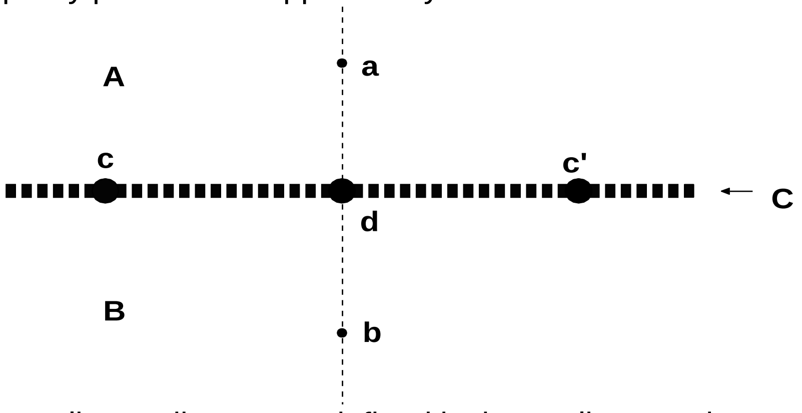


- ➢ There are two on points (closed circles) for each segment and one off point (open circle)
- ➢ Note that the selected test points are shared with adjacent domains.
- ➢ The on points for two adjacent boundary segments can also be shared.
- ➢ The shared on points is given below.



(d) Equality and Inequality Predicates:

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



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

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

(e) Random Testing:

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

(f) Testing n-Dimensional Domains:

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

(iii) Procedure:

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

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

(iv) Variations, Tools, Effectiveness:

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

(4) Domains and Interface Testing:

(i) General:

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

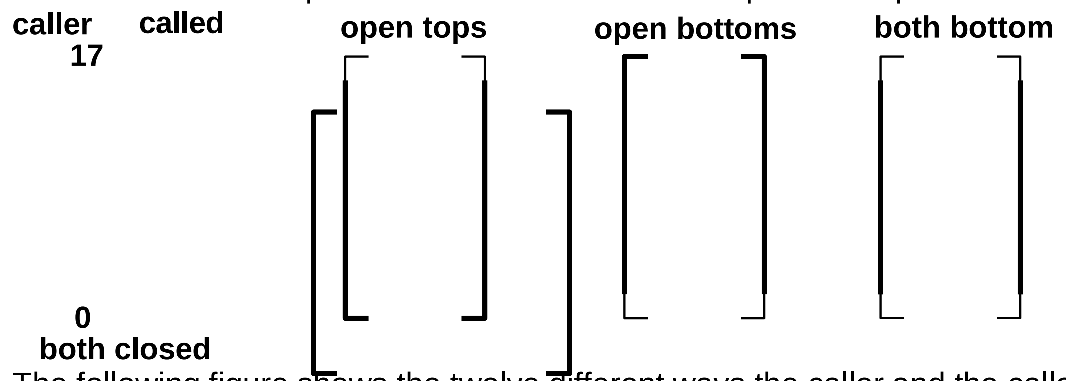
(ii) Domains and Range:

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

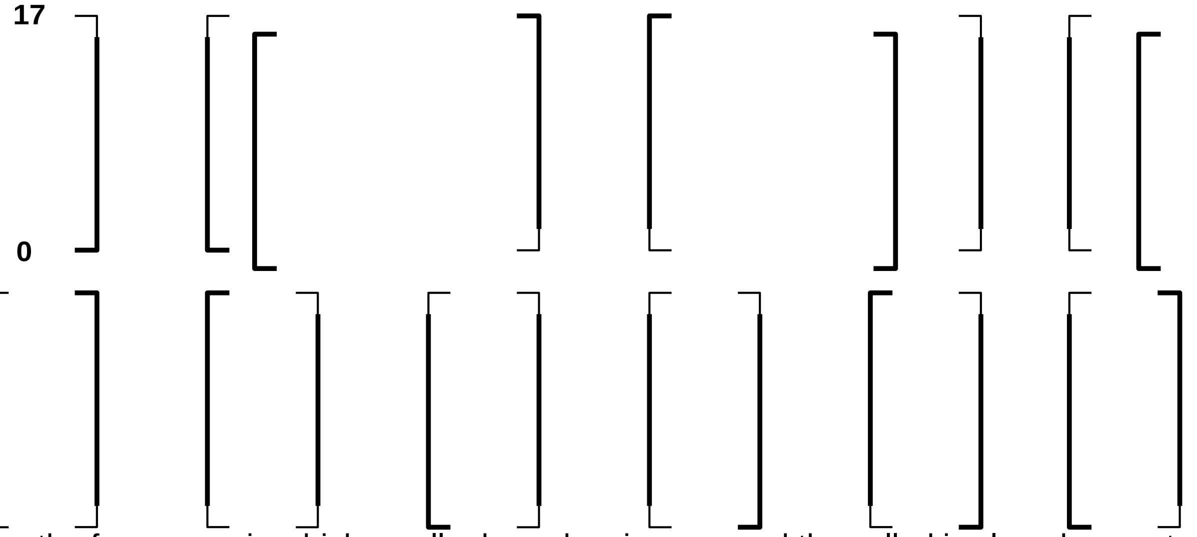
$$\begin{tikzcd}
\text{Caller domain} \arrow[r] & \text{Caller range} \\
\text{Caller range} \arrow[r] & \text{Called domain} \\
\text{Called domain} \arrow[r] & \text{Called range}
\end{tikzcd}$$

(iii) Closure Compatibility:

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



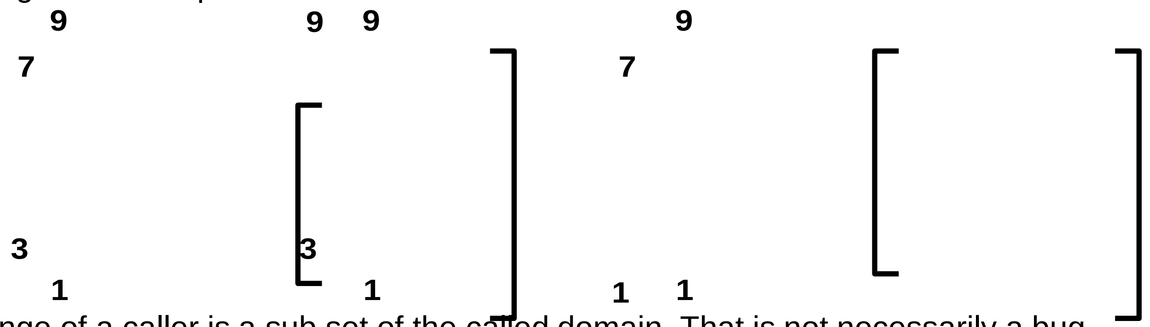
- The following figure shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs.



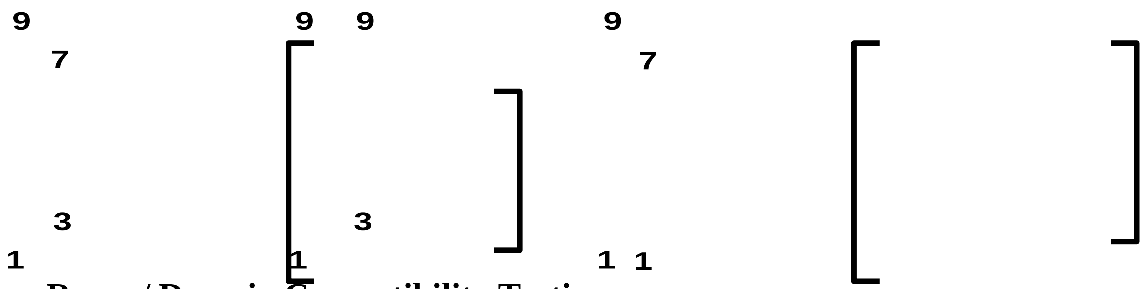
- Here the four cases in which a caller boundary is open and the called is closed are not buggy.

(iv) Span Compatibility:

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



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



(v) Interface Range/ Domain Compatibility Testing:

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

(vi) Finding the values:

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

(5) Domains and Testability:

(i) General:

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

(ii) Linearizing Transformations:

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

(i) Polynomials:

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

(ii) Logarithmic Transforms:

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

(iii) More general forms:

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

(iii) Coordinate Transformations:

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

(iv) A Canonical Program Form:

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

(v) Great Insights:

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