

UNIT II

TRANSACTION FLOW TESTING AND DATA FLOW TESTING

Transaction Flow Testing:-transaction flows, transaction flow testing techniques.

Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.

Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

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:

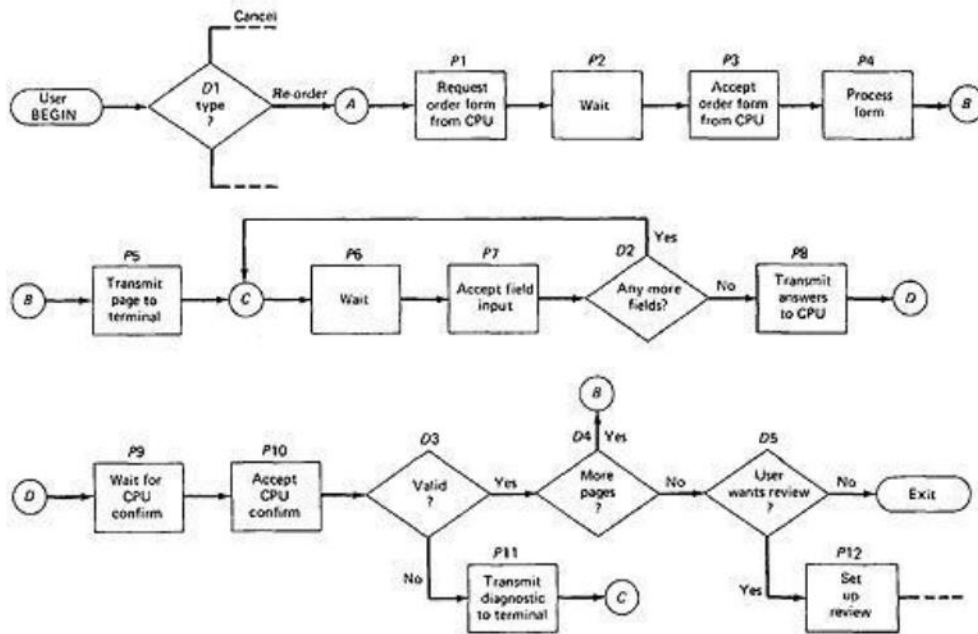


Figure 3.1: An Example of a Transaction Flow

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.

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 its identity. (See Figure 3.2 (b))

Mitosis: Here the parent transaction is destroyed and two new transactions are created. (See Figure 3.2 (c))

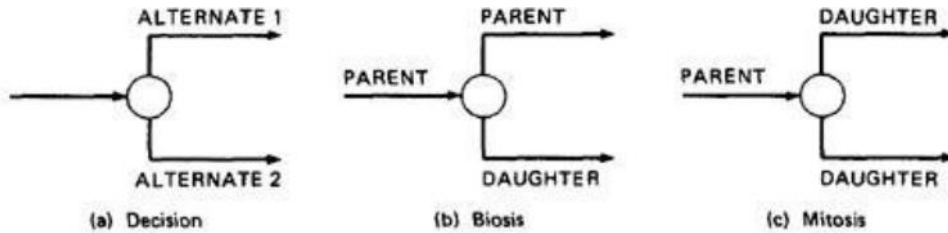


Figure 3.2: Nodes with multiple outlinks

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

(c))

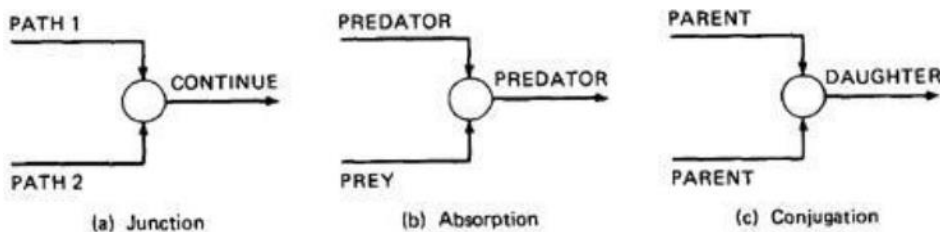


Figure 3.3: Transaction Flow Junctions and Mergers

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.

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.

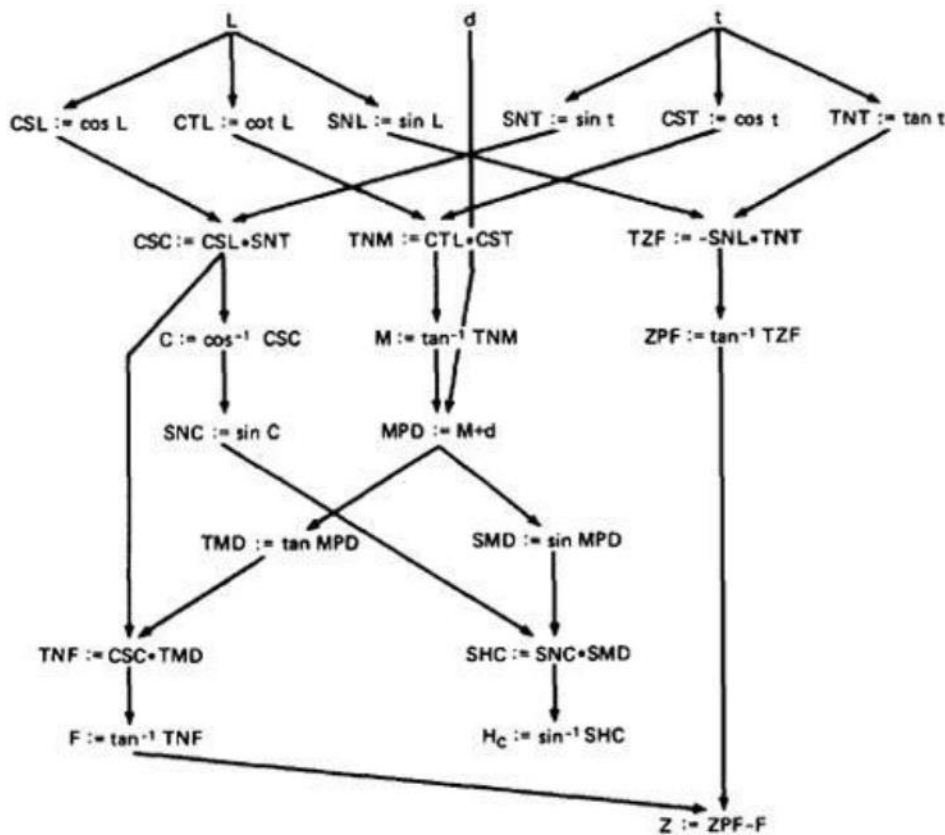


Figure 3.4: Example of a data flow graph

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:

Defined: d - defined, created, initialized etc

Killed or undefined: k - killed, undefined, released etc

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.

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

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.

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.

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:

-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.

DATA FLOW ANOMALY STATE GRAPH:

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

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.

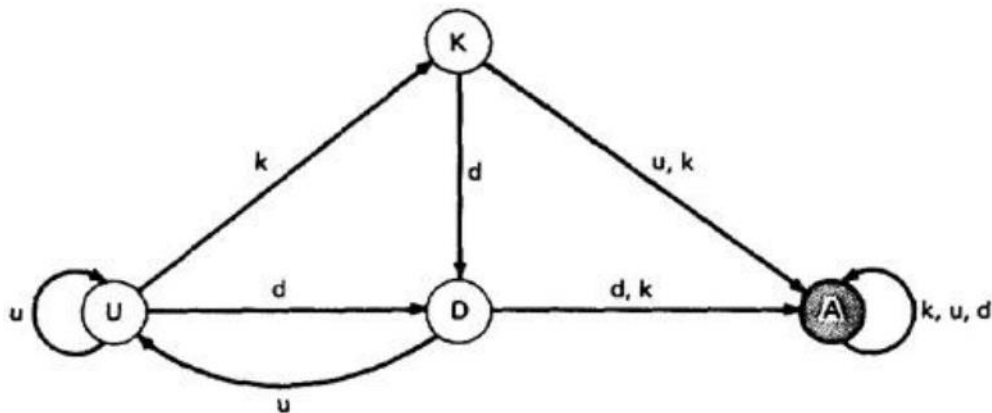


Figure 3.5: Unforgiving Data Flow Anomaly State Graph

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

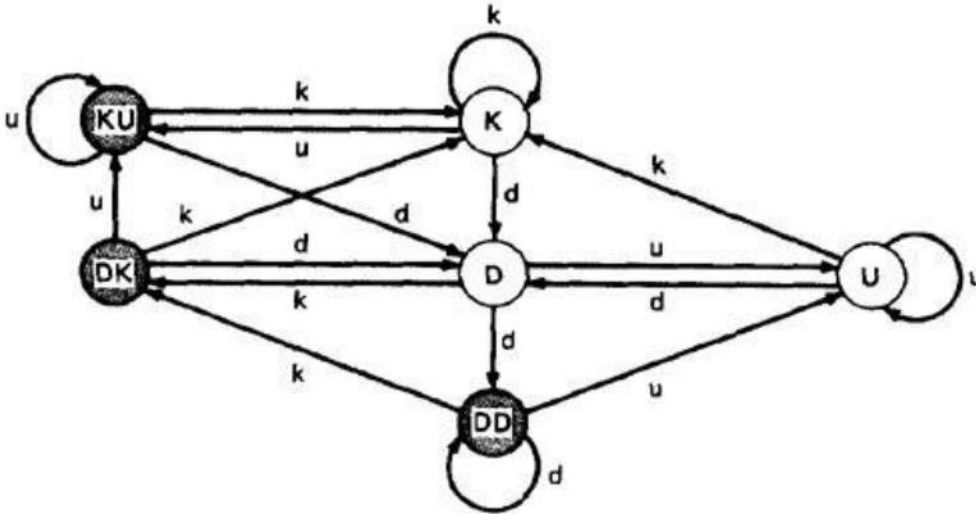


Figure 3.6: Forgiving Data Flow Anomaly State Graph

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 belong 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:

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.

Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.

The 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.

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

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.

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.

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

V(U-1) := V(U+1) + U(V-1)
ELL: V(U+U(V)) := U + V
IF U = V GOTO JOE
IF U > V THEN U := Z
Z := U
END
```

* A contrived horror

Figure 3.7: Program Example (PDL)

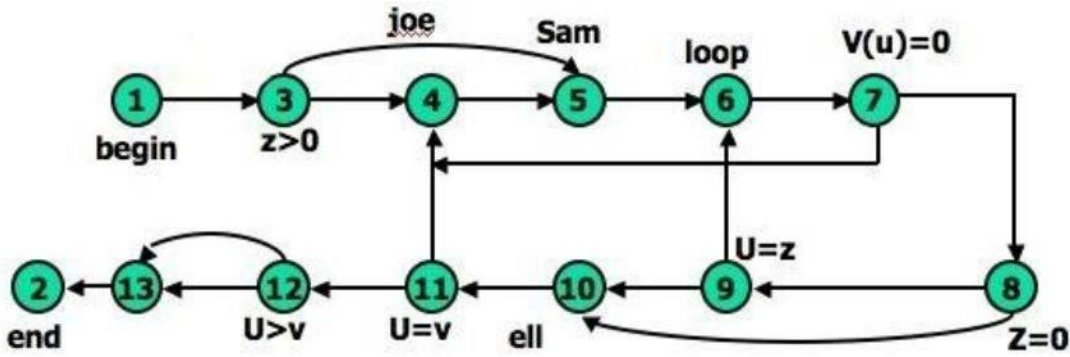


Figure 3.8: Unannotated flow graph for example program in Figure 3.7

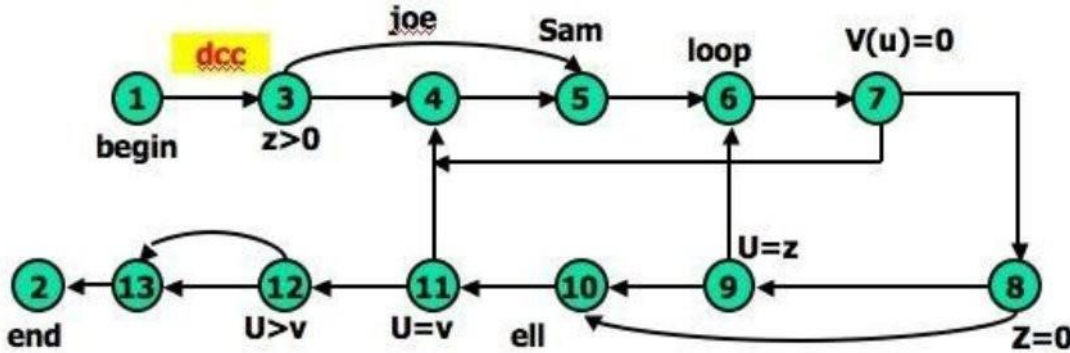


Figure 3.9: Control flow graph annotated for X and Y data flows.

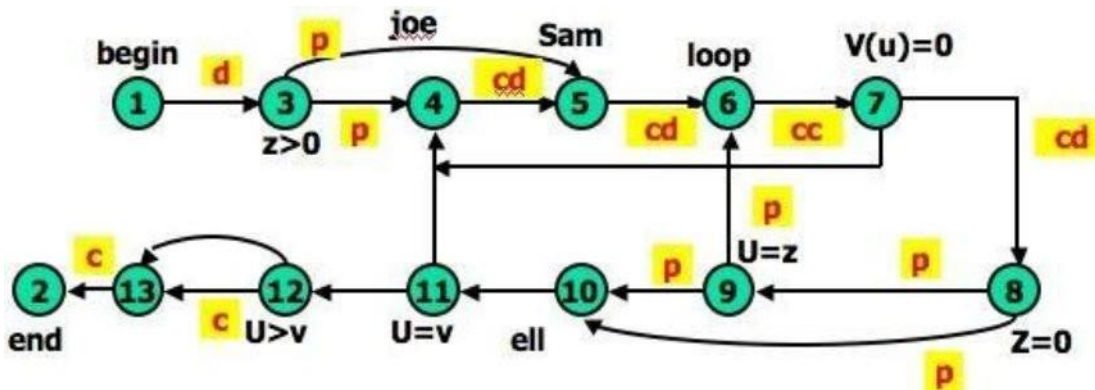


Figure 3.10: Control flow graph annotated for Z data flow.

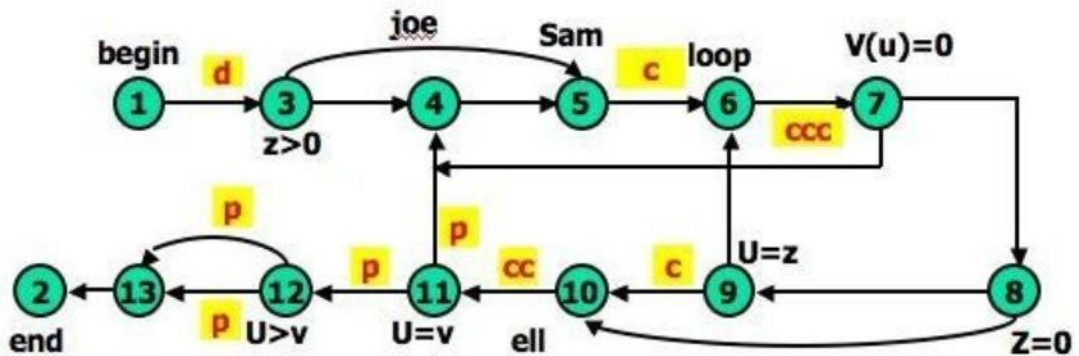


Figure 3.11: Control flow graph annotated for V data flow.

STRATEGIES OF 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:

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. If 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).

Loop-Free Path Segment is a path segment for which every node in it is visited at most 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.

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.

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 Strategy (AU): The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.

Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

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 at least 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.

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 3.12 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

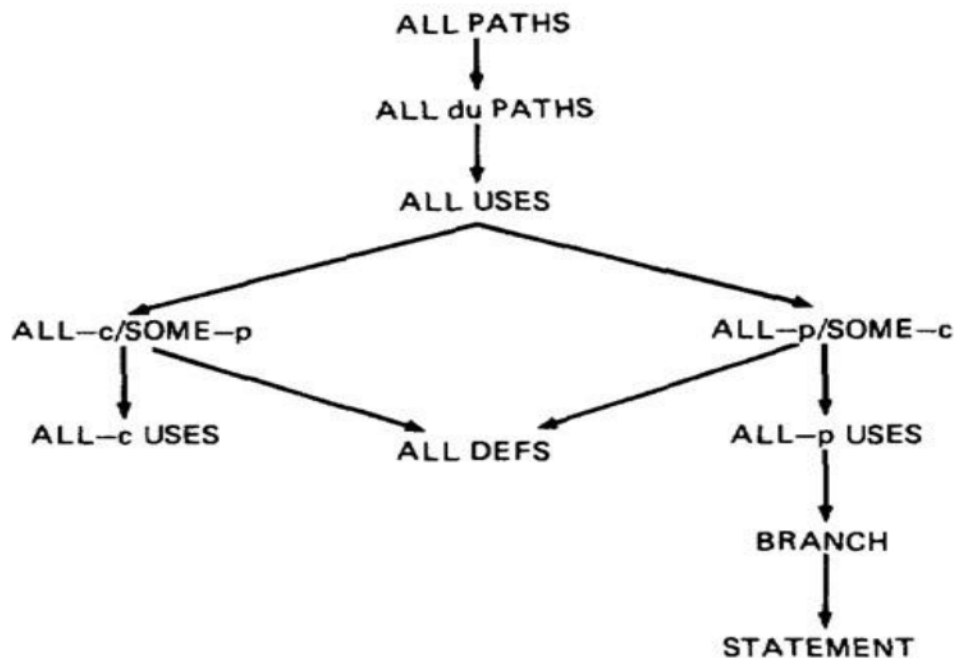


Figure 3.12: Relative Strength of Structural Test Strategies.

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.

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.

DOMAINS AND PATHS:

INTRODUCTION:

Domain: In mathematics, domain is a set of possible values of an independent variable or the variables of a function.

Programs as input data classifiers: domain testing attempts to determine whether the classification is or is not correct.

Domain testing can be based on specifications or equivalent implementation information.

If domain testing is based on specifications, it is a functional test technique.

If domain testing is based implementation details, it is a structural test technique.

For example, you're doing domain testing when you check extreme values of an input variable.

All inputs to a program can be considered as if they are numbers. For example, a character string can be treated as a number by concatenating bits and looking at them as if they were a binary integer. This is the view in domain testing, which is why this strategy has a mathematical flavor.

THE MODEL: The following figure is a schematic representation of domain testing.

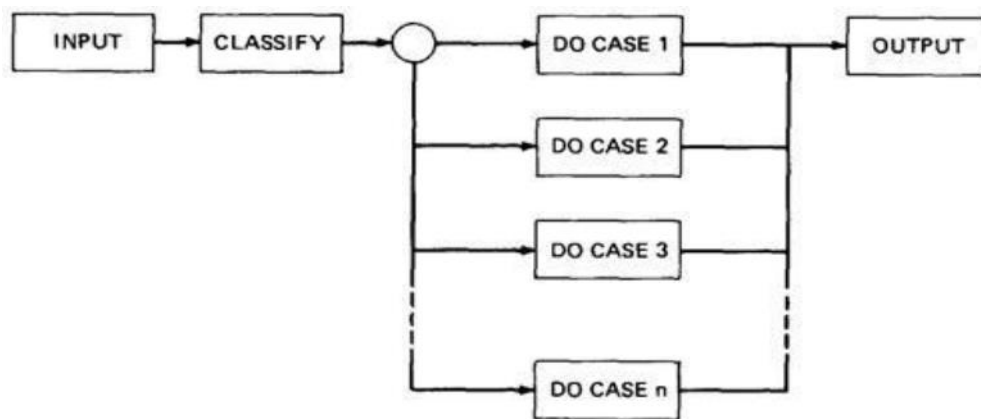


Figure 4.1: Schematic Representation of Domain Testing.

Before doing whatever it does, a routine must classify the input and set it moving on the right path.

An invalid input (e.g., value too big) is just a special processing case called 'reject'.

The input then passes to a hypothetical subroutine rather than on calculations.

In domain testing, we focus on the classification aspect of the routine rather than on the calculations.

Structural knowledge is not needed for this model - only a consistent, complete specification of input values for each case.

We can infer that for each case there must be at least one path to process that case.

A DOMAIN IS A SET:

An input domain is a set.

If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.

Domain testing does not work well with arbitrary discrete sets of data objects.

Domain for a loop-free program corresponds to a set of numbers defined over the input vector.

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, then predicate interpretation must be based on actual paths through the routine - that is, based on the implementation control flow graph.

Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flow graph for the routine; but usually, as is the nature of specifications, no interpretation is needed because the domains are specified directly.

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 or if the domain is defined by the union of two or more domains.

Domains are defined their boundaries. Domain boundaries are also where most domain bugs occur.

For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.

For example, in the statement IF $x > 0$ THEN ALPHA ELSE BETA we know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s).

A domain may have one or more boundaries - no matter how many variables define it. For example, if the predicate is $x^2 + y^2 < 16$, the domain is the inside of a circle of radius 4 about the origin. Similarly, we could define a spherical domain with one boundary but in three variables.

Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.

A DOMAIN CLOSURE:

A domain boundary is **closed** with respect to a domain if the points on the boundary belong to the domain.

If the boundary points belong to some other domain, the boundary is said to be

open.

Figure 4.2 shows three situations for a one-dimensional domain - i.e., a domain defined over one input variable; call it x

The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, $x \geq 0$ when $x > 0$ was intended

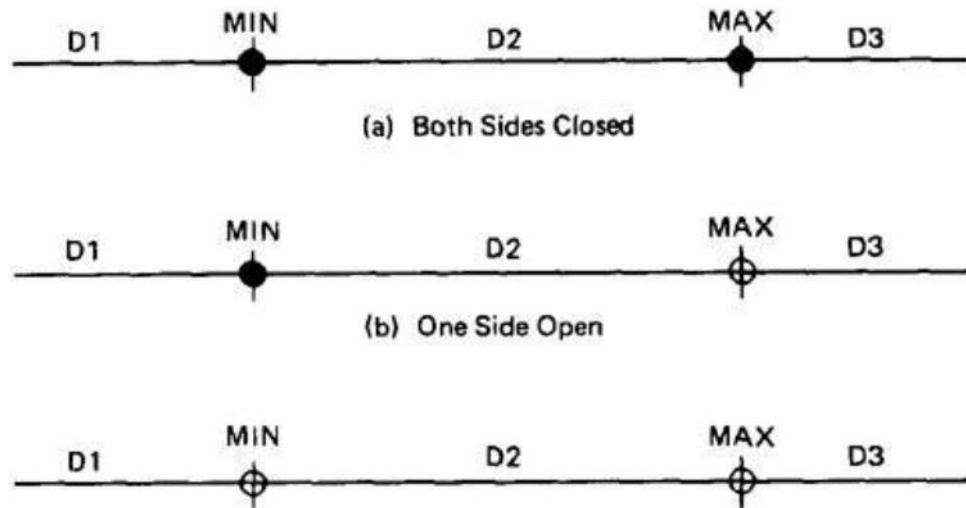


Figure 4.2: Open and Closed Domains.

DOMAIN DIMENSIONALITY:

Every input variable adds one dimension to the domain.

One variable defines domains on a number line.

Two variables define planar domains.

Three variables define solid domains.

Every new predicate slices through previously defined domains and cuts them in half.

Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space.

Thus, planes are cut by lines and points, volumes by planes, lines and points and n-spaces by hyperplanes.

BUG ASSUMPTION:

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

An incorrectly implemented domain means that boundaries are wrong, which may in turn mean that control flow predicates are wrong.

Many different bugs can result in domain errors. Some of them are:

Domain Errors:

Double Zero Representation: In computers or Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.

Floating point zero check: A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself or multiplied by zero. So the floating point zero check to be done against an epsilon value.

Contradictory domains: An implemented domain can never be ambiguous or contradictory, but a specified domain can. A contradictory domain specification means that at least two supposedly distinct domains overlap.

Ambiguous domains: Ambiguous domains means that union of the domains is incomplete. That is there are missing domains or holes in the specified domains. Not specifying what happens to points on the domain boundary is a common ambiguity.

Over specified Domains: his domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.

Boundary Errors: Errors caused in and around the boundary of a domain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.

Closure Reversal: A common bug. The predicate is defined in terms of

\geq . The programmer chooses to implement the logical complement and incorrectly uses

\leq for the new predicate; i.e., $x \geq 0$ is incorrectly negated as $x \leq 0$, thereby shifting boundary values to adjacent domains.

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

RESTRICTIONS TO DOMAIN TESTING: Domain testing has restrictions, as do other testing techniques. Some of them include:

Co-incidental Correctness: Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. If we're plagued by coincidental correctness we may misjudge an incorrect boundary. Note that this implies weakness for domain testing when dealing with routines that have binary outcomes (i.e., TRUE/FALSE)

Representative Outcome: Domain testing is an example of **partition testing**. Partition-testing strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain.

If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct. Most test techniques, functional or structural, fall under partition testing and therefore make this representative outcome assumption. For example, x^2 and 2^x are equal for $x = 2$, but the functions are different. The functional differences between adjacent domains are usually simple, such as $x + 7$ versus $x + 9$, rather than x^2 versus 2^x .

Simple Domain Boundaries and Compound Predicates: Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, $x \geq 0$ AND $x < 17$, just specifies two domain boundaries by one compound predicate. As an example of a compound predicate that specifies one boundary, consider: $x = 0$ AND $y \geq 7$ AND $y \leq 14$. This predicate specifies one boundary equation ($x = 0$) but alternates closure, putting it in one or the other domain depending on whether $y < 7$ or $y > 14$. Treat compound predicates with respect because they're more complicated than they seem.

Functional Homogeneity of Bugs: Whatever the bug is, it will not change the functional form of the boundary predicate. For example, if the predicate is $ax \geq b$, the bug will be in the value of a or b but it will not change the predicate to ax

$\geq b$, say.

Linear Vector Space: Most papers on domain testing, assume linear boundaries - not a bad assumption because in practice most boundary predicates are linear.

Loop Free Software: Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression (after interpretation), which means a possible domain boundary change.

NICE AND UGLY DOMAINS:

NICE DOMAINS:

Where do these domains come from?

Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction.

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.

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.

Some important properties of nice domains are: **Linear, Complete, Systematic, And Orthogonal, Consistently closed, Convex and simply connected.**

To the extent that domains have these properties domain testing is easy as testing gets. The bug frequency is lesser for nice domain than for ugly domains.

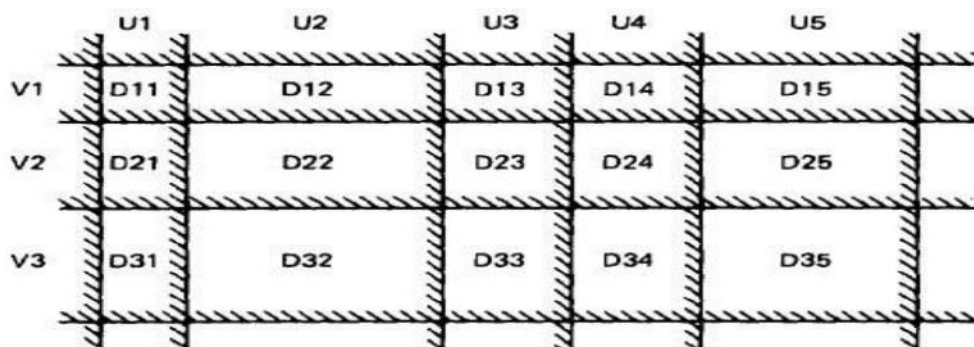


Figure 4.3: Nice Two-Dimensional Domains.

LINEAR AND NON LINEAR BOUNDARIES:

Nice domain boundaries are defined by linear inequalities or equations.

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 $n+1$ point to determine an n -dimensional hyper plane.

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

COMPLETE BOUNDARIES:

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

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

Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, 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.

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

If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

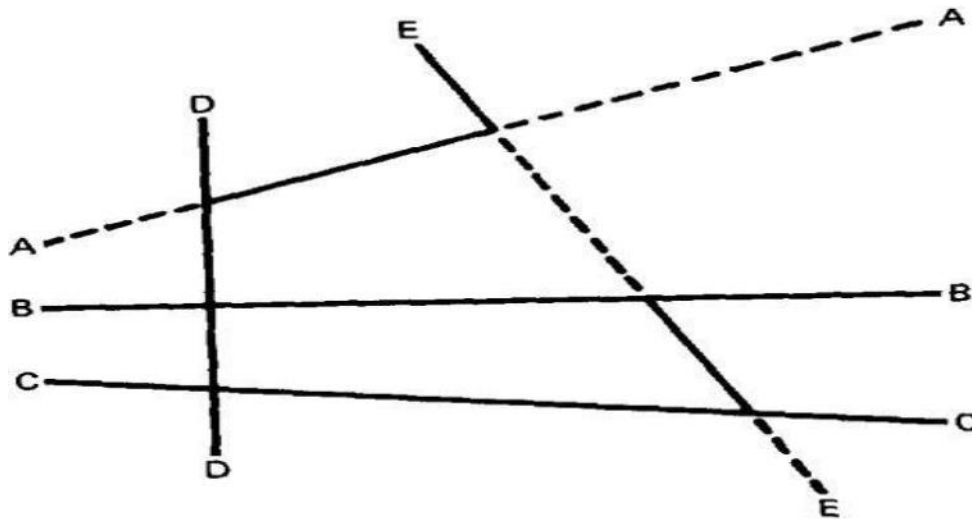


Figure 4.4: Incomplete Domain Boundaries.

SYSTEMATIC BOUNDARIES:

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

In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant.

$$\begin{array}{l}
 f_1(X) \geq k_1 \text{ or } f_1(X) \geq g(1,c) \\
 f_1(X) \geq k_2 \quad f_2(X) \geq g(2,c) \\
 \dots\dots\dots \quad \dots\dots\dots \\
 f_i(X) \geq k_i \quad f_i(X) \geq g(i,c)
 \end{array}$$

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 $k + ic$.

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).

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.

ORTHOGONAL BOUNDARIES:

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 .

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

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 $O(n^2)$.

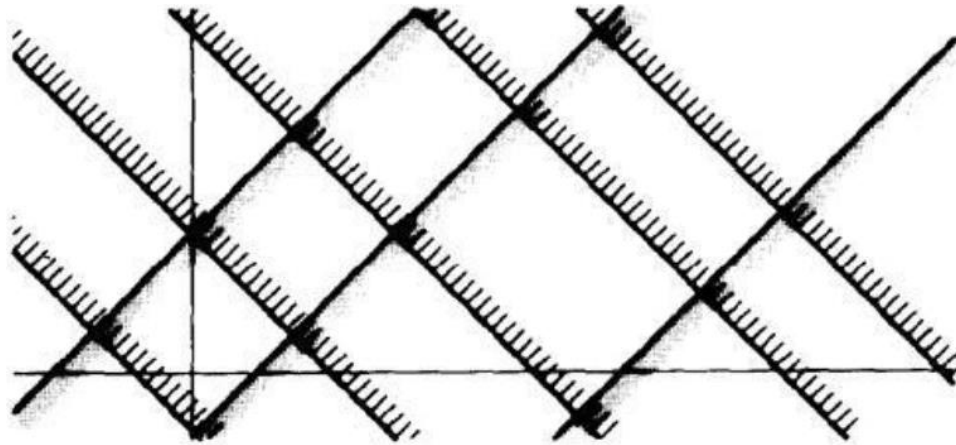


Figure 4.5: Tilted Boundaries.

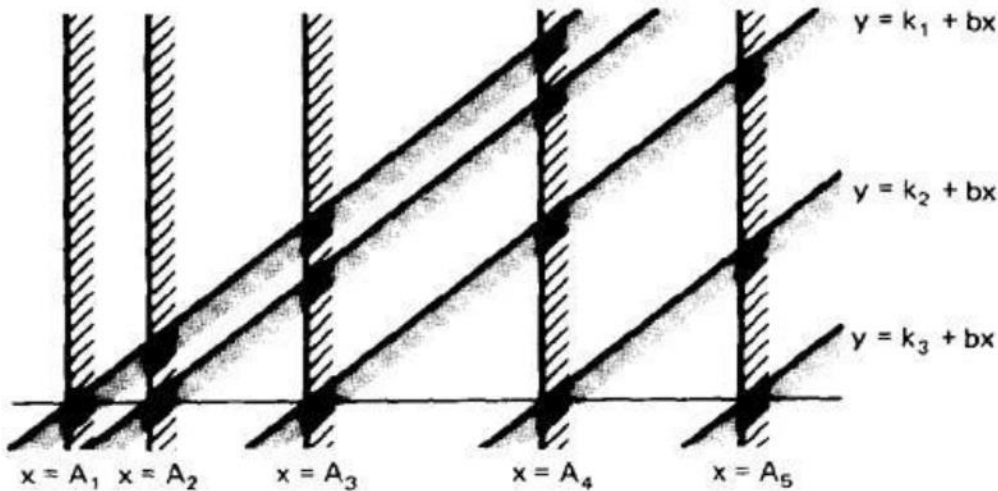


Figure 4.6: Linear, Non-orthogonal Domain Boundaries.

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).

CLOSURE CONSISTENCY:

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

The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies - e.g., the boundary lines belong to the domains on the right.

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.

CONVEX:

A geometric figure (in any number of dimensions) is convex if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within the figure.

Nice domains are convex; dirty domains aren't.

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.

SIMPLY CONNECTED:

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

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

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.

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.

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

UGLY DOMAINS:

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

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

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.

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 (e.g., the income tax code), such "simplifications" constitute bugs.

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

AMBIGUITIES AND CONTRADICTIONS:

Domain ambiguities are holes in the input space.

The holes may lie within the domains or in cracks between domains.

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

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

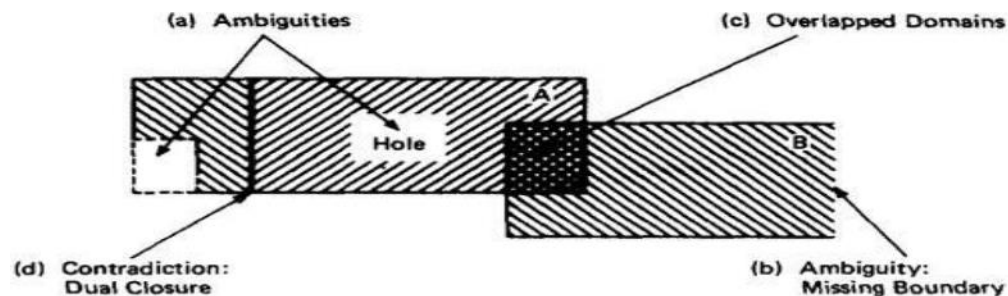


Figure 4.7: Domain Ambiguities and Contradictions.

SIMPLIFYING THE TOPOLOGY:

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

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

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

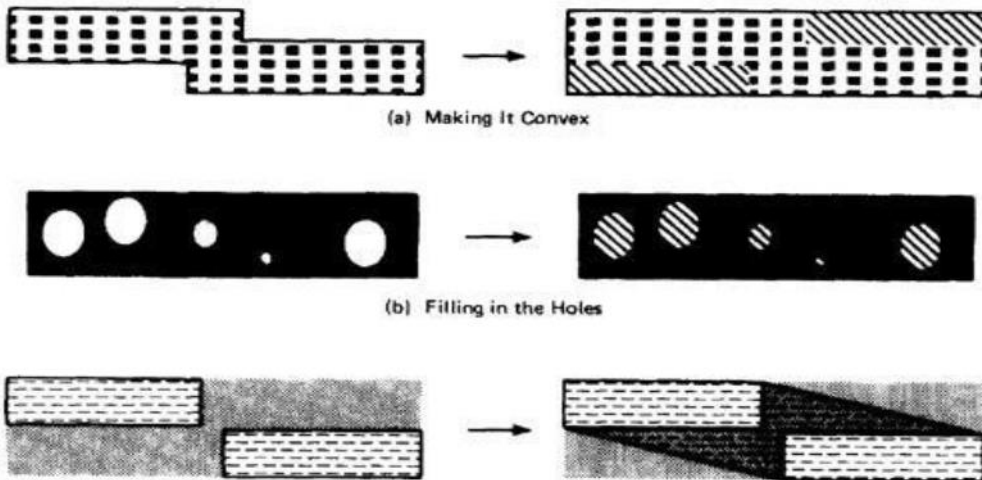


Figure 4.8: Simplifying the topology.

RECTIFYING BOUNDARY CLOSURES:

If domain boundaries are parallel but have closures that go every which way (left, right, left . . .) the natural reaction is to make closures go the same way (see Figure 4.9).

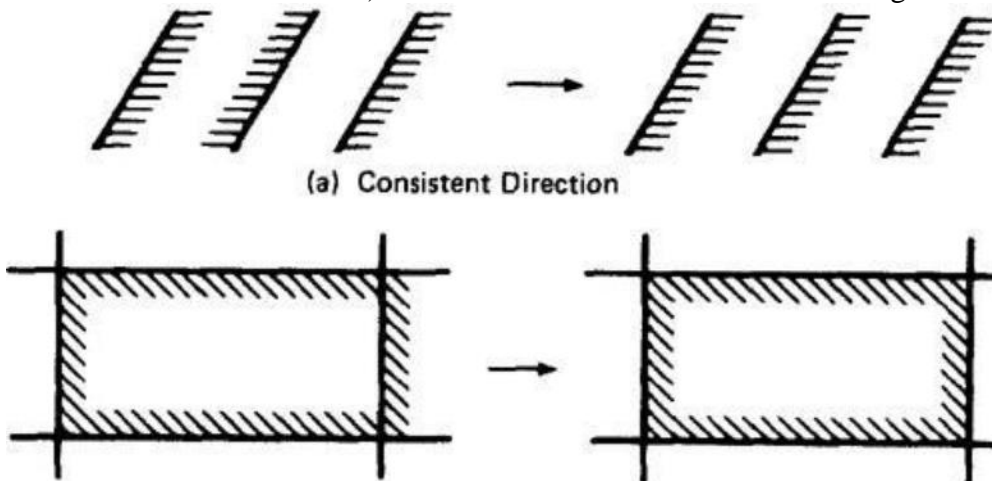


Figure 4.9: Forcing Closure Consistency.

DOMAIN TESTING:

DOMAIN TESTING STRATEGY: The domain-testing strategy is simple, although possibly tedious (slow).

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

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.

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.

Run the tests and by posttest analysis (the tedious part) determine if any

boundaries are faulty and if so, how.

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.

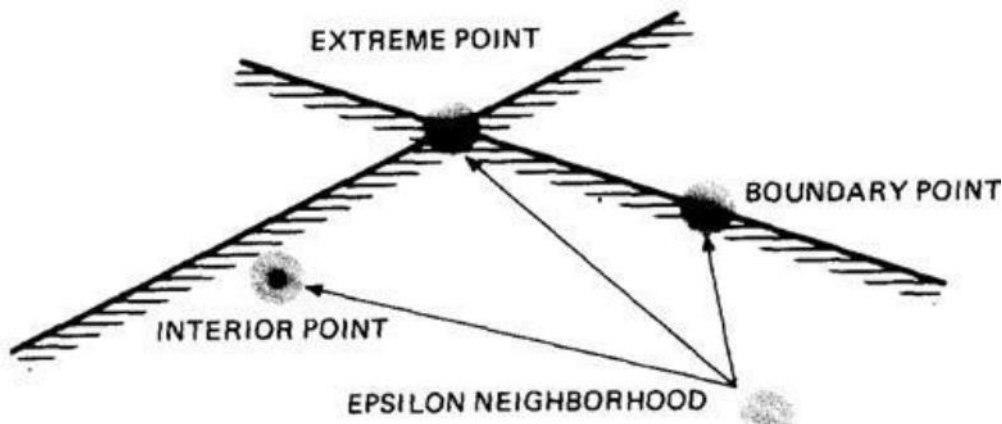


Figure 4.10: Interior, Boundary and Extreme points.

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.

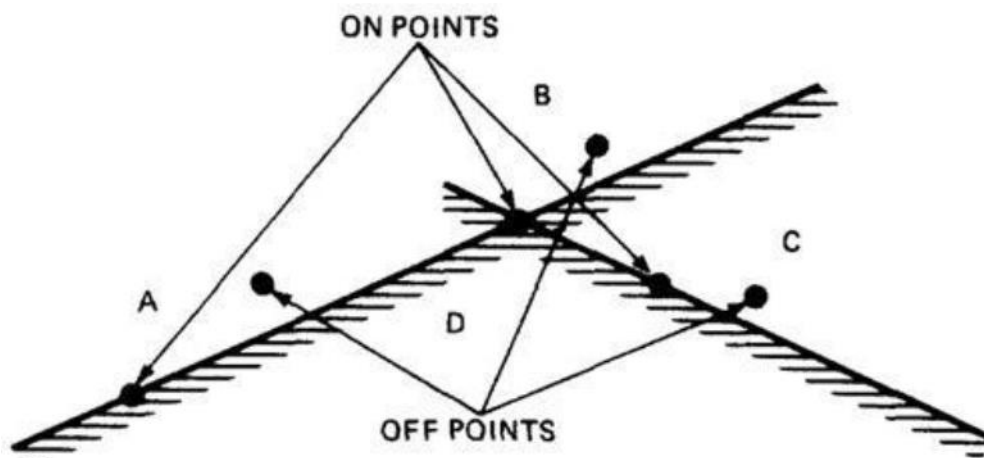


Figure 4.11: On points and Off points.

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

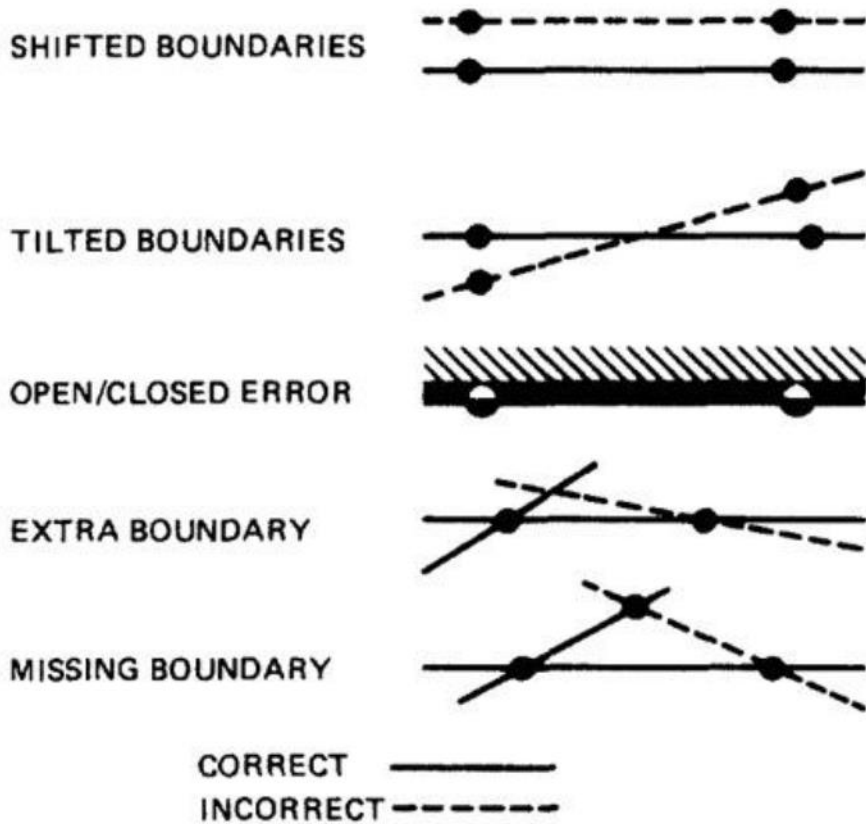


Figure 4.12: Generic Domain Bugs.

TESTING ONE DIMENSIONAL DOMAIN:

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 shows possible domain bugs for a one-dimensional open domain boundary.

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 \geq or $<$ to \leq (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.

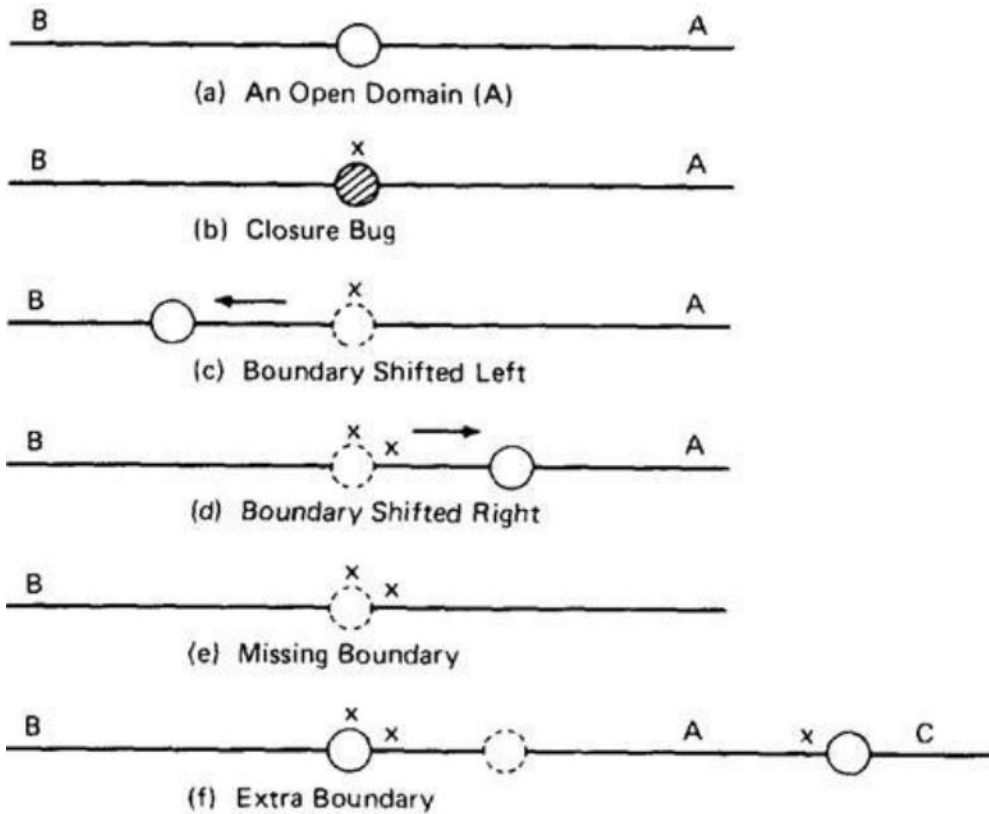


Figure 4.13: One Dimensional Domain Bugs, Open Boundaries.

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.

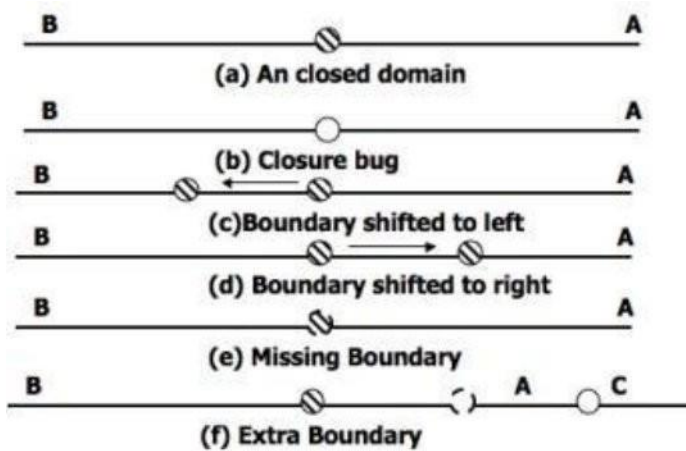


Figure 4.14: One Dimensional Domain Bugs, Closed Boundaries.

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.

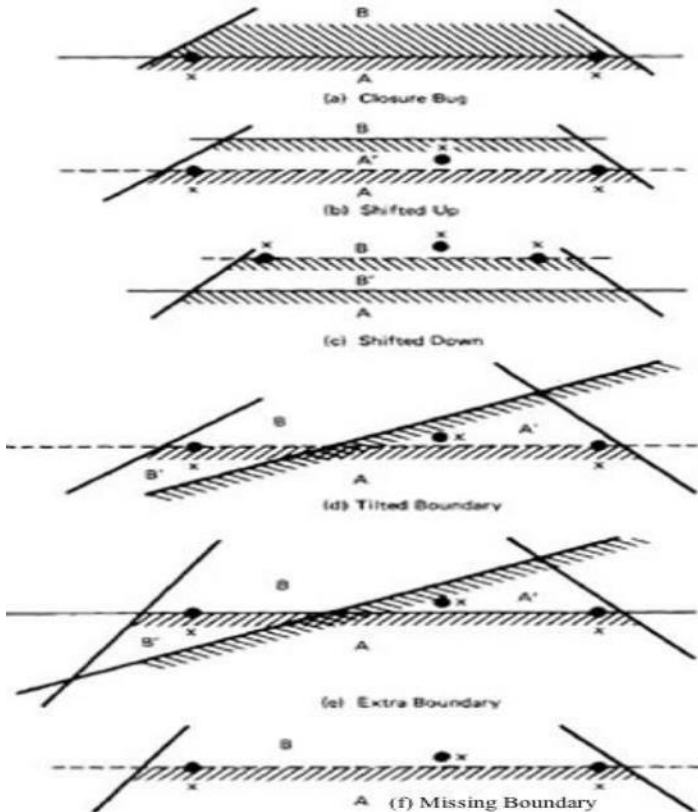


Figure 4.15: Two Dimensional Domain Bugs.

For Closed Boundaries:

Closure Bug: Figure 4.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, $x \geq k$ 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.

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 $x + y \geq 17$ when $x + y \geq 7$ 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.

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.

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.

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.

Identify input variables.

Identify variable which appear in domain defining predicates, such as control flow predicates.

Interpret all domain predicates in terms of input variables.

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.

DOMAIN AND INTERFACE TESTING

INTRODUCTION:

Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components.

Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across their interface.

Interface between any two components is considered as a subroutine call.

We're looking for bugs in that "call" when we do interface testing.

Let's assume that the call sequence is correct and that there are no type incompatibilities.

For a single variable, the domain span is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least):

compatible domain spans and compatible closures (Compatible but need not be Equal).

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.

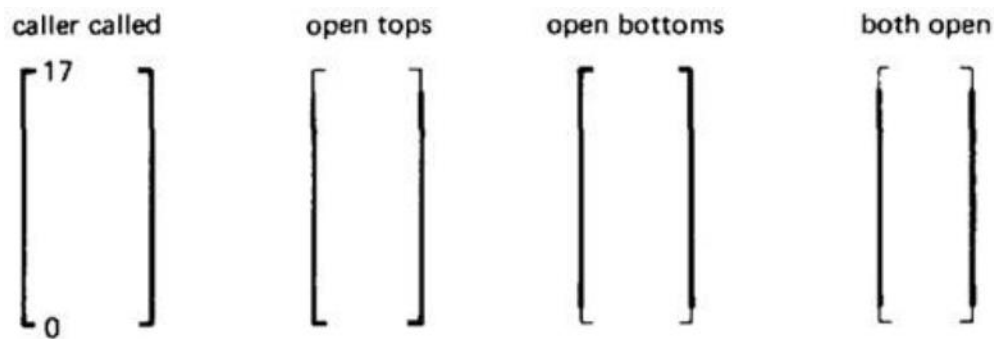


Figure 4.16: Range / Domain Closure Compatibility.

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.

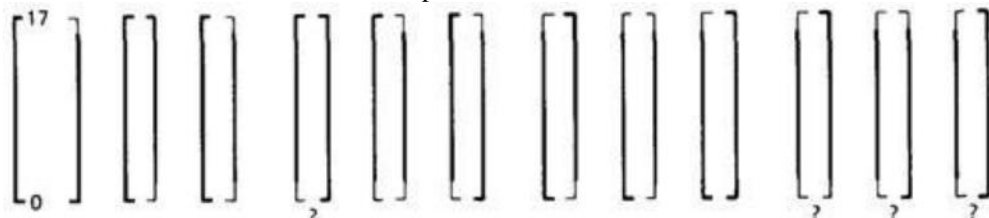


Figure 4.17: Equal-Span Range / Domain Compatibility Bugs.

SPAN COMPATIBILITY:

Figure 4.18 shows three possibly harmless span incompatibilities.

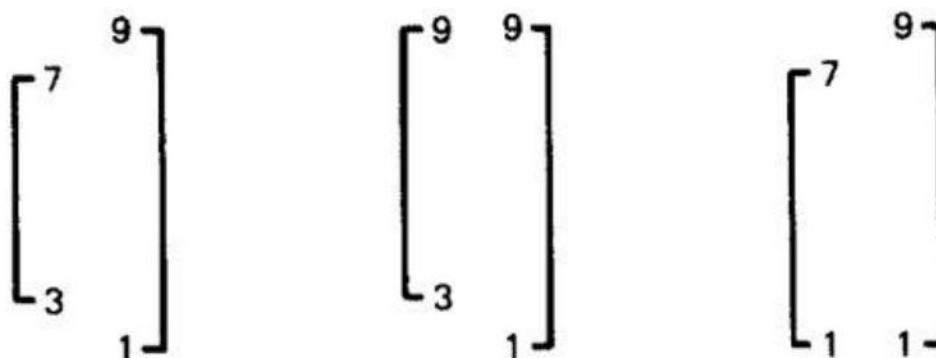


Figure 4.18: Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

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.

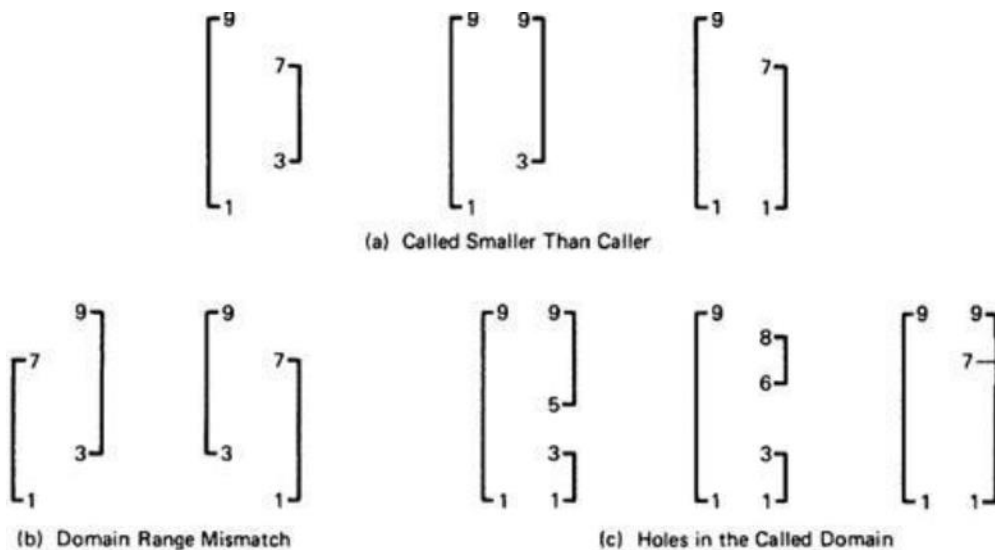


Figure 4.19: Buggy Range / Domain Mismatches

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.