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))

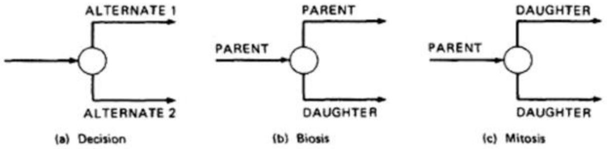


Figure 3.2: Nodes with multiple outlinks

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

0. 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))
1. Absorption: In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))
2. 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))

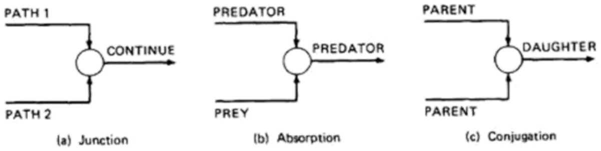


Figure 3.3: Transaction Flow Junctions and Mergers

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:

- o Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- o Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- o The system's design documentation should contain an overview section that

details the main transaction flows.

- o 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.

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

BUG ASSUMPTION:

- The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that

data objects are not available when they should be, or silly things are being done to data objects.

- Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
- Although we'll be doing data-flow testing, we won't be using data flowgraphs as such. Rather, we'll use an ordinary control flowgraph annotated to show what happens to the data objects of interest at the moment.

DATA FLOW GRAPHS:

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

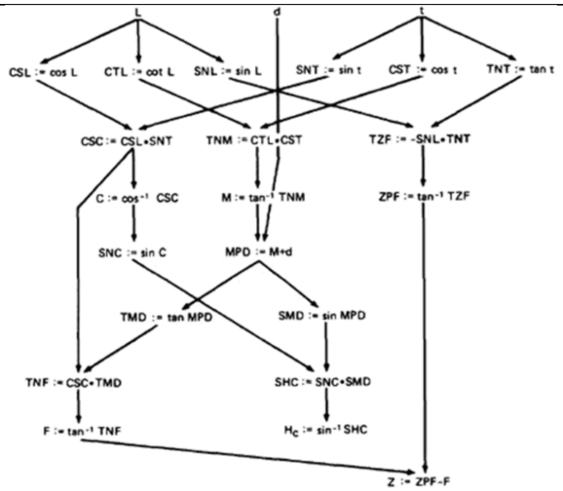


Figure 3.4: Example of a data flow graph

- We will use an control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the

data flows we want.

- Data Object State and Usage:
  - Data Objects can be created, killed and used.
  - They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
  - The following symbols denote these possibilities:
    1. Defined: d - defined, created, initialized etc
    2. Killed or undefined: k - killed, undefined, released etc
    3. Usage: u - used for something (c - used in Calculations, p - used in a predicate)
  - 1. Defined (d):
    - An object is defined explicitly when it appears in a data declaration.
    - Or implicitly when it appears on the left hand side of the assignment.
    - It is also to be used to mean that a file has been opened.
    - A dynamically allocated object has been allocated.
    - Something is pushed on to the stack.

A record written.

2. Killed or Undefined (k):

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known with certitude (with aboslute certainity / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A

3. Usage (u):

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

- It is used in a Predicate (p) when it appears directly in a predicate.

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.

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

0. -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.
1. -d :- okay. This is just the first definition along this path.
2. -u :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
3. k- :- not anomalous. The last thing done on this path was to kill the variable.
4. d- :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
5. 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:

0. K :- undefined, previously killed, doesnot exist

1. D :- defined but not yet used for anything
2. U :- has been used for computation or in predicate
3. 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.

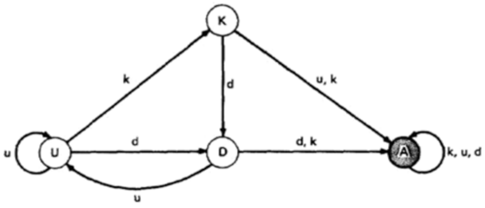


Figure 3.5: Unforgiving Data Flow Anomaly State Graph

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.

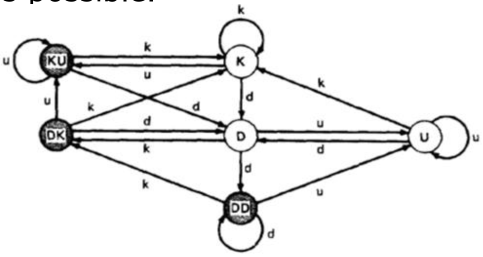


Figure 3.6: Forgiving Data Flow Anomaly State Graph

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

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 flowgraph..

Here we annotate each link with symbols (for example, d, k, u, c, 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.

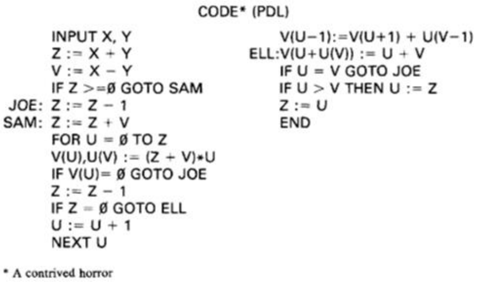
Components of the model:

0. 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.
1. 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.
2. 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.
3. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
4. 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.

5. 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.
6. 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 >=Ø GOTO SAM
JOE:  Z := Z − 1
SAM:  Z := Z + V
      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
    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)

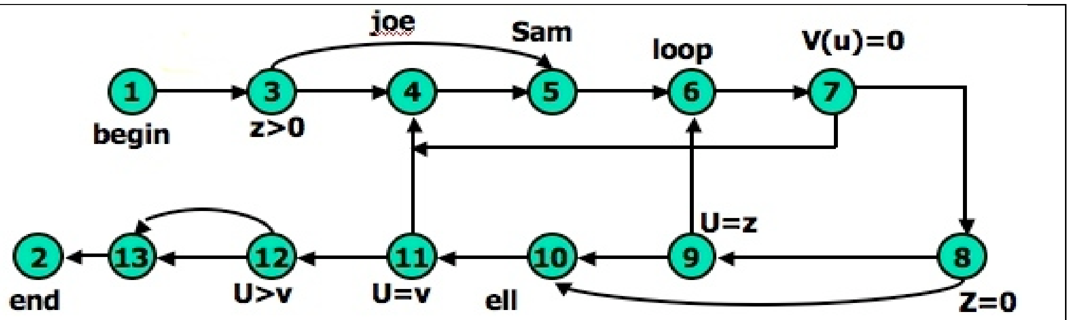


Figure 3.8: Unannotated flowgraph for example program in Figure 3.7

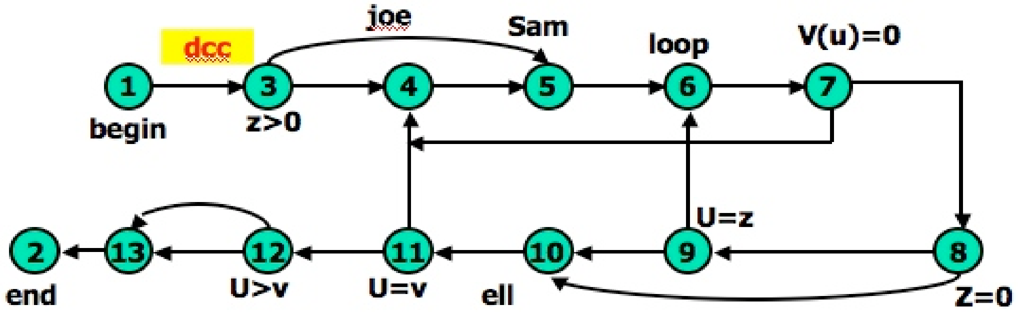


Figure 3.9: Control flowgraph annotated for X and Y data flows.

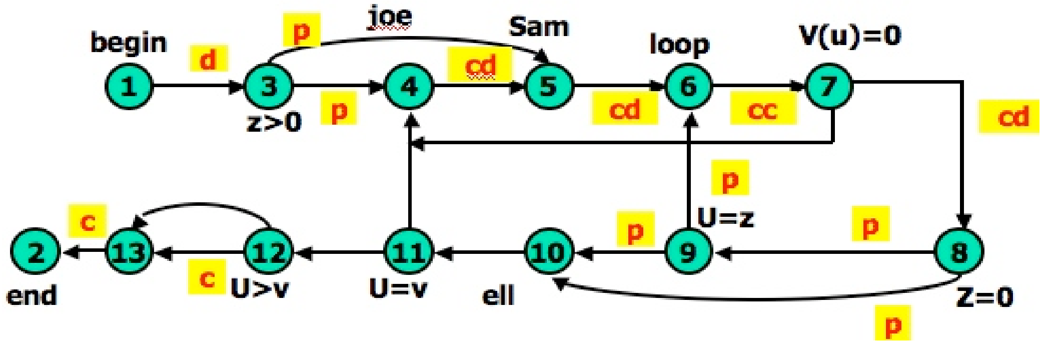


Figure 3.10: Control flowgraph annotated for Z data flow.

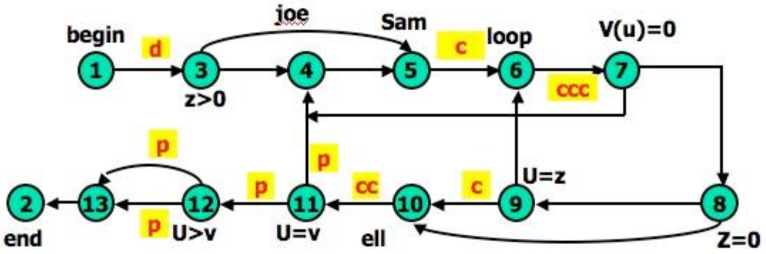


Figure 3.11: Control flowgraph annotated for V data flow.

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 subpaths that contain a d (or u, k, du, dk).
- A strategy X is stronger than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for weaker.

TERMINOLOGY:

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

STRATEGIES: The structural test strategies discussed below are based on the program's control flowgraph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

0. All - du Paths (ADUP): The all-du-paths (ADUP) strategy is

the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

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

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

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

The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

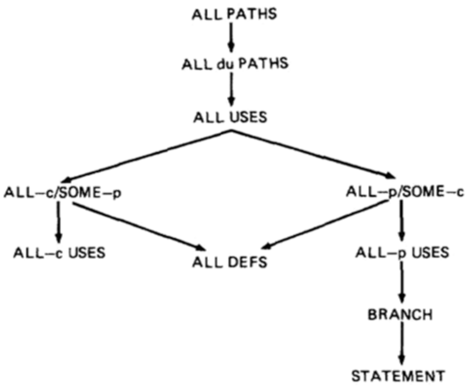


Figure 3.12: Relative Strength of Structural Test Strategies.

- 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

- 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. o Dynamic slicing is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

UNIT – III

Concepts

DOMAINS AND PATHS	A. 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.

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

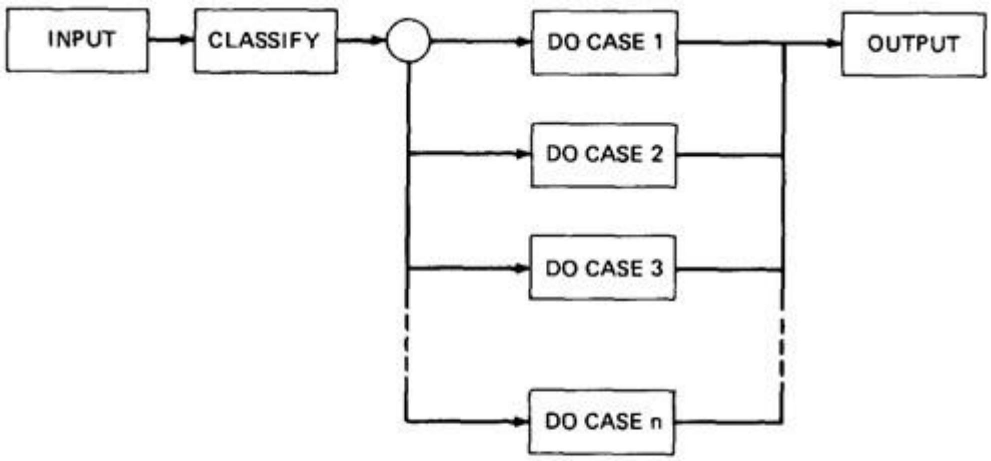


Figure 4.1: Schematic Representation of Domain Testing.

- 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 atleast one path to process that case.

C. 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.

D. 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.