Module 3

Dr. Sandip Mal

Module 3

- **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.

DATA FLOW TESTING

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

Data 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)

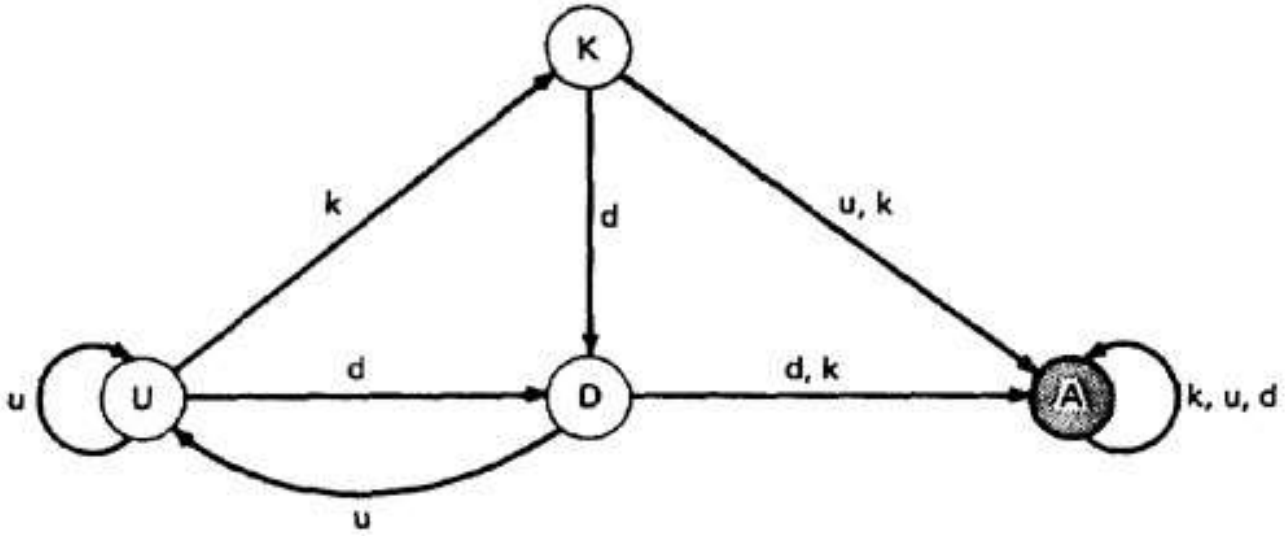
- **Defined (d):**
  - An object is defined explicitly when it appears in a data declaration.
  - Or implicitly when it appears on the left hand side of the assignment.
  - It is also to be used to mean that a file has been opened.
  - A dynamically allocated object has been allocated.
  - Something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

DATA FLOW ANOMALIES

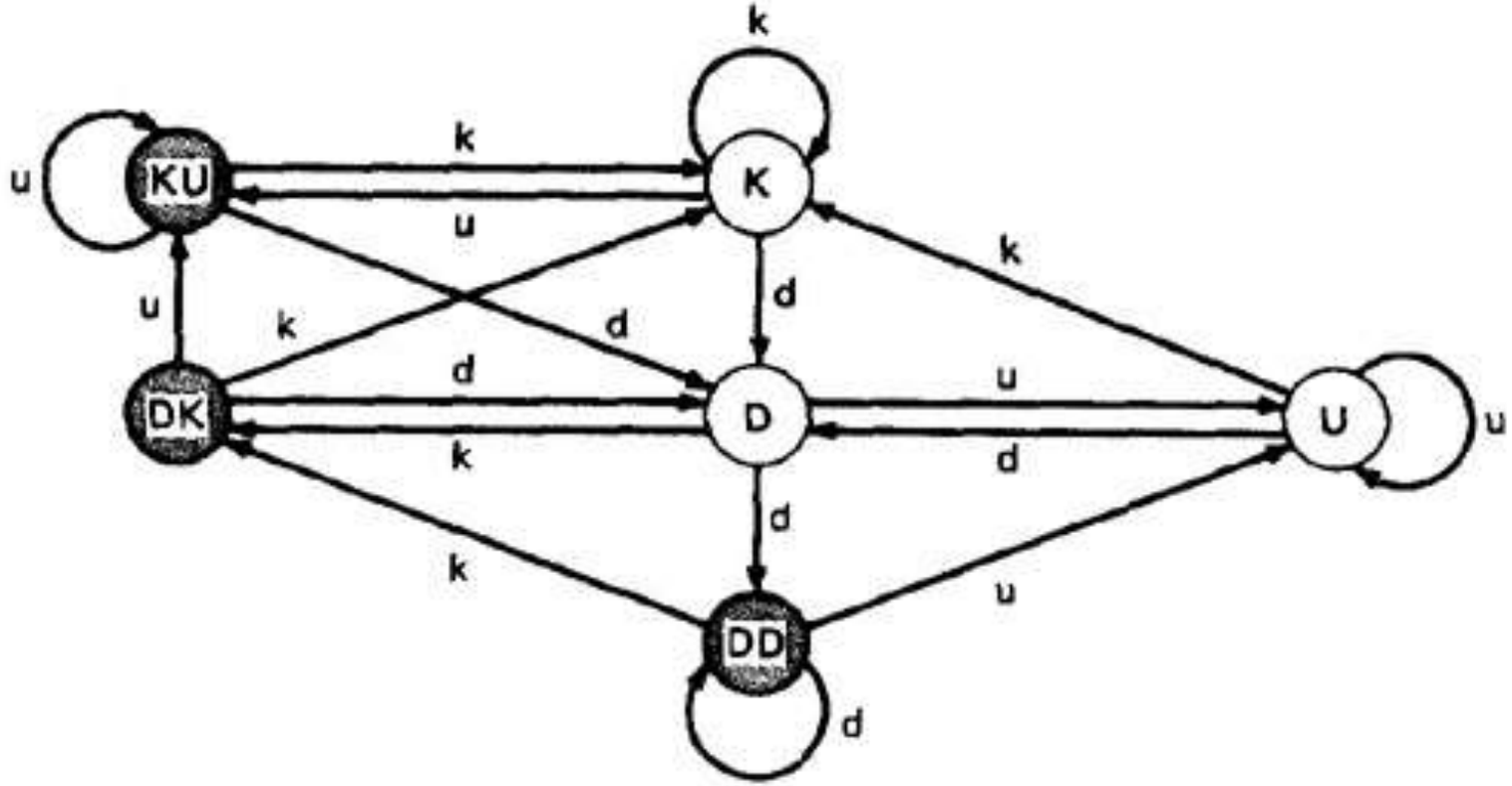
- An anomaly is denoted by a two-character sequence of actions.
- For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.
- What is an anomaly is depend on the application.
- There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.
  - **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 does not exist.
  - **ud** :- usually not a bug because the language permits reassignment at almost any time.
  - **uk** :- normal situation.
  - **uu** :- normal situation.

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, does not exist
- **D** :- defined but not yet used for anything
- **U** :- has been used for computation or in predicate
- **A** :- anomalous
- **Unforgiving Data - Flow Anomaly Flow Graph:** Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.



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



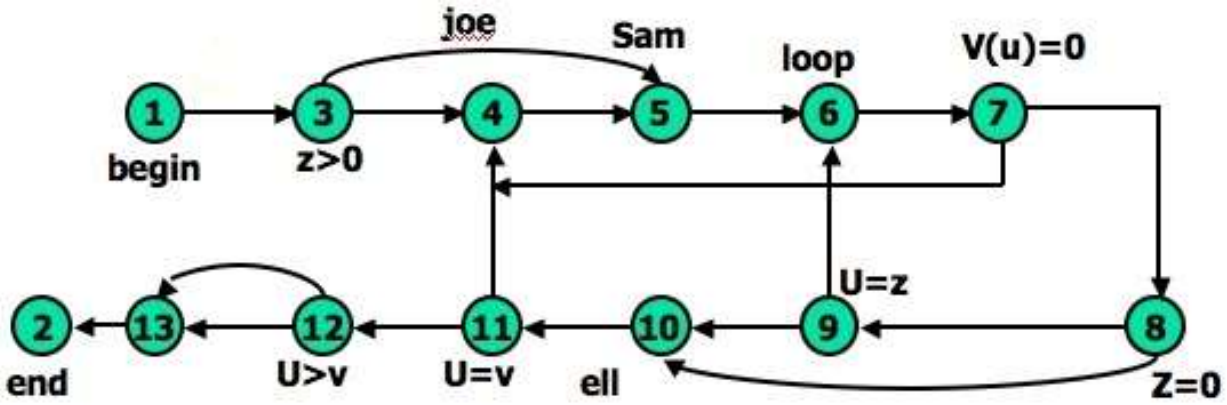
Example

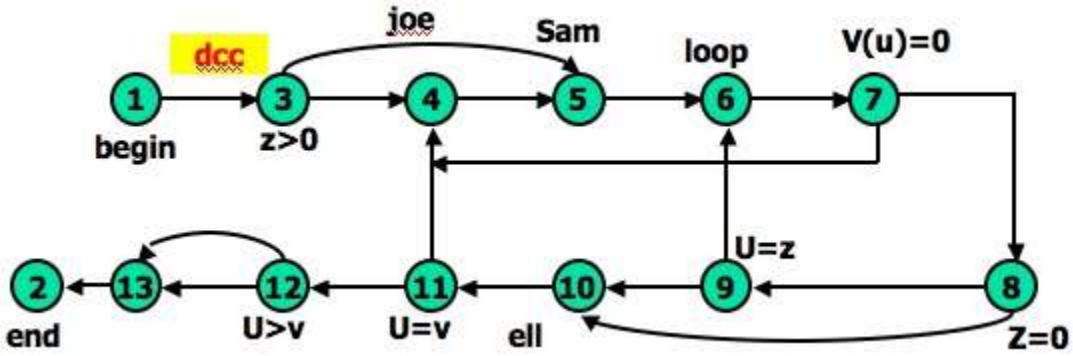
CODE* (PDL)

```
      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

Control flow graph





Control flow graph annotated for X and Y data flows.

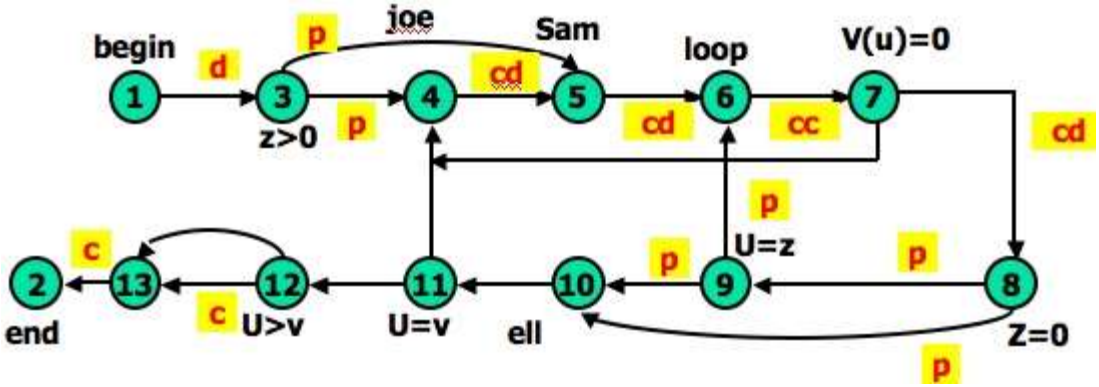


Figure: Control flow graph annotated for Z data flow.

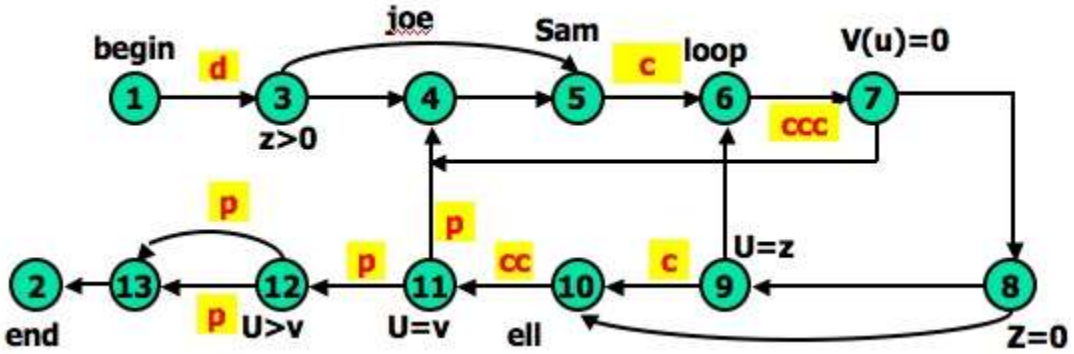
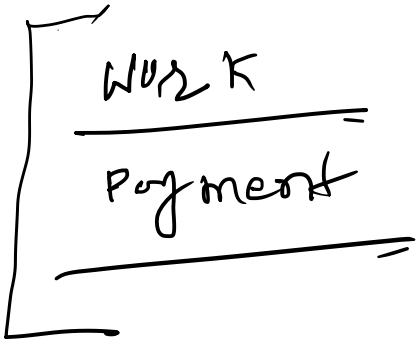


Figure : Control flow graph annotated for V data flow

```
main()
{ int work;
  double Payment = 0;
  scanf("%d", &work);
  if (work > 0) {
        Payment = 40;
        if (work <= 30)
           Payment = Payment + (work - 25) * 0.5;
        else {
           Payment = payment + 50 + (work - 30) * 0.1;
           if (payment >= 3000)
                 payment = payment * 0.9;
        }
    }
}
```

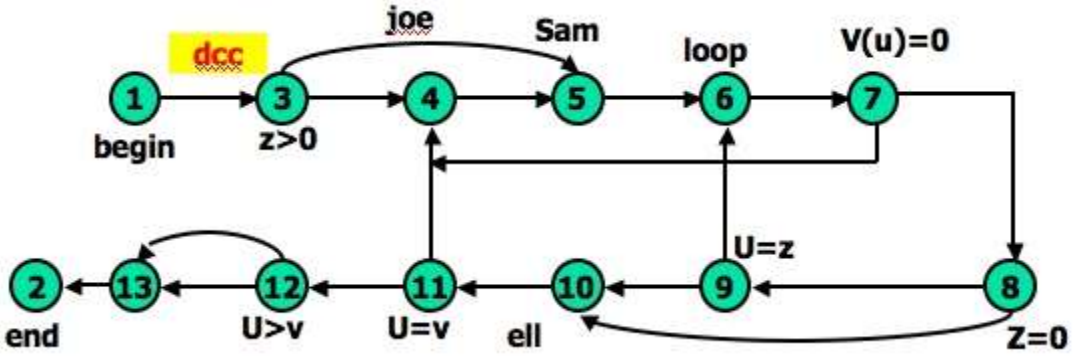


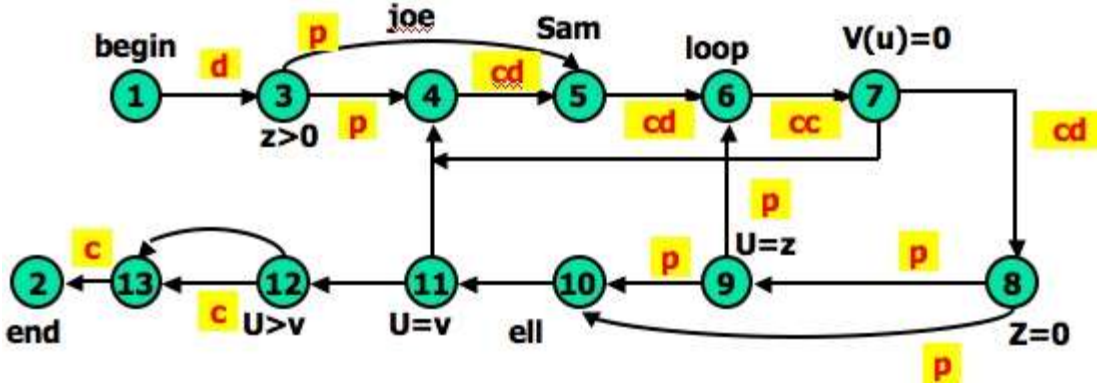
STRATEGIES OF DATA FLOW TESTING

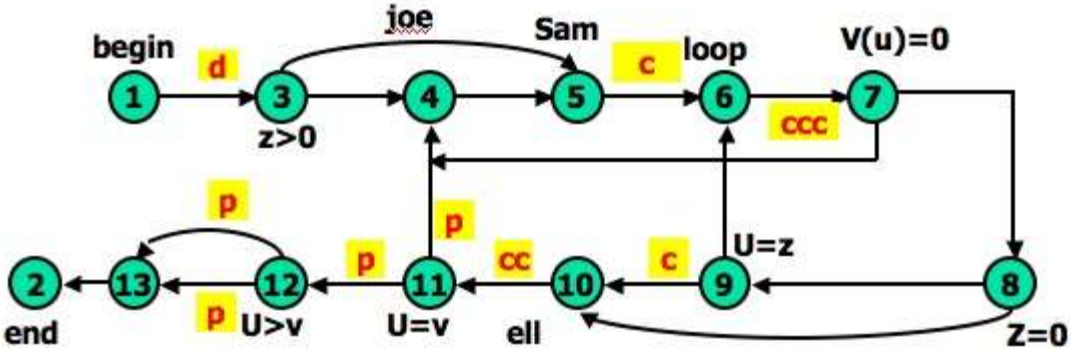
- Data Flow Testing Strategies are structural strategies.
- In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
- In other words, data flow strategies require data-flow link weights (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

- **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. paths in Figure 2 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3, 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 atmost once. For Example, path (4,5,6,7,8,10) in Figure 3 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 (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.


joe
Sam
dcc
loop
V(u)=0
1
3
4
5
6
7
begin
z>0
U=z
2
13
12
11
10
9
8
end
U>v
U=v
ell
Z=0


begin
d
p
joe
Sam
cd
loop
V(u)=0
1
3
4
5
6
7
z>0
p
cd
cc
cd
p
U=z
c
p
p
2
13
12
11
10
9
8
c
end
U>v
U=v
ell
Z=0
p

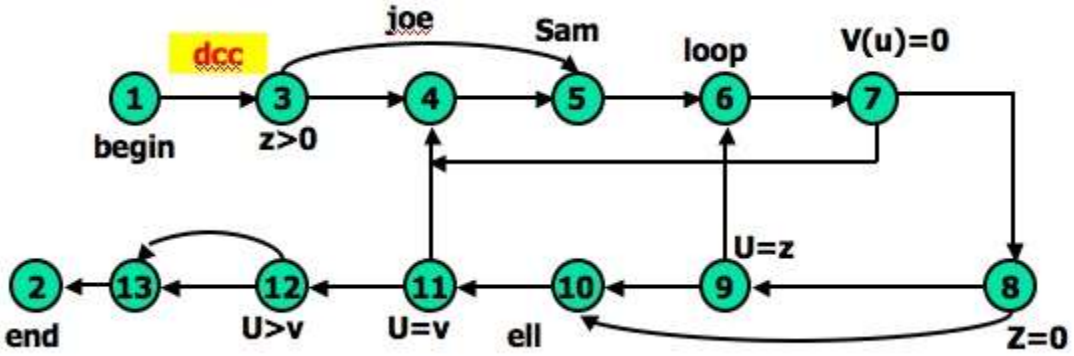

begin
d
joe
Sam
c
loop
V(u)=0
1
3
4
5
6
7
z>0
ccc
p
p
p
U=z
2
13
12
11
10
9
8
p
p
cc
c
end
U>v
U=v
ell
Z=0

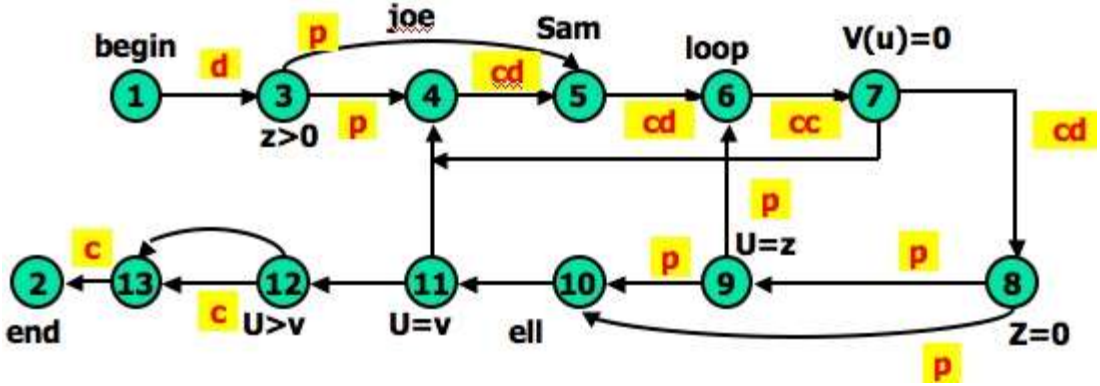
STRATEGIES

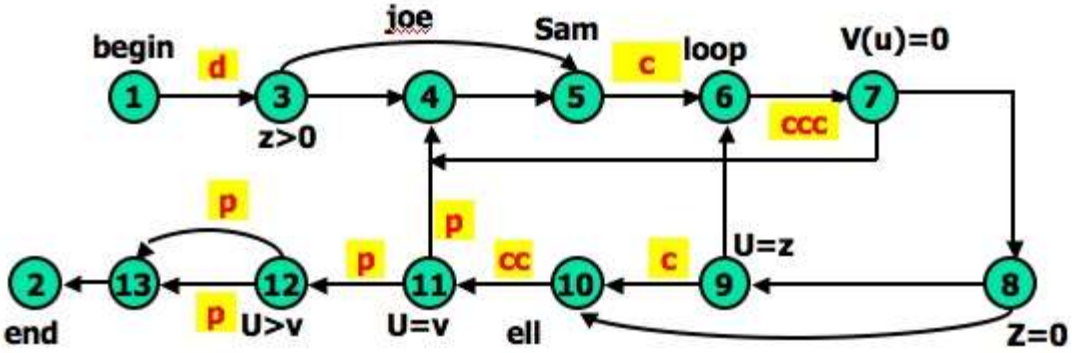
All - du Paths (ADUP): The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

For variable X and Y: In Figure 1, 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 2) 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).

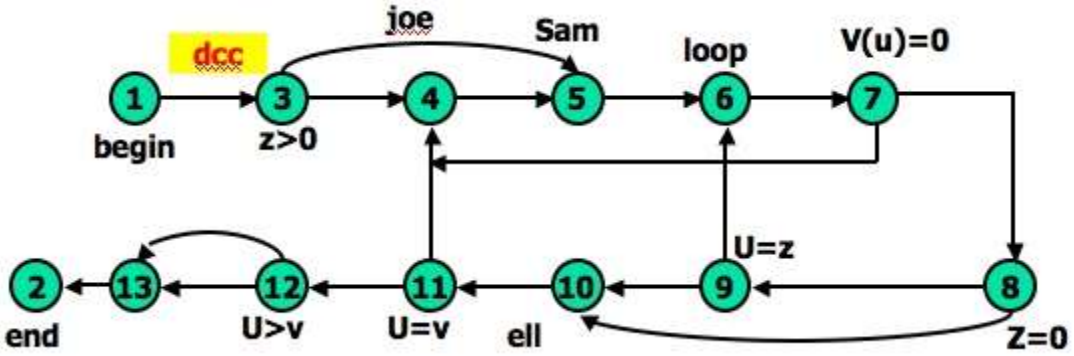

joe
Sam
dcc
loop
V(u)=0
1
3
4
5
6
7
begin
z>0
U=z
2
13
12
11
10
9
8
end
U>v
U=v
ell
Z=0

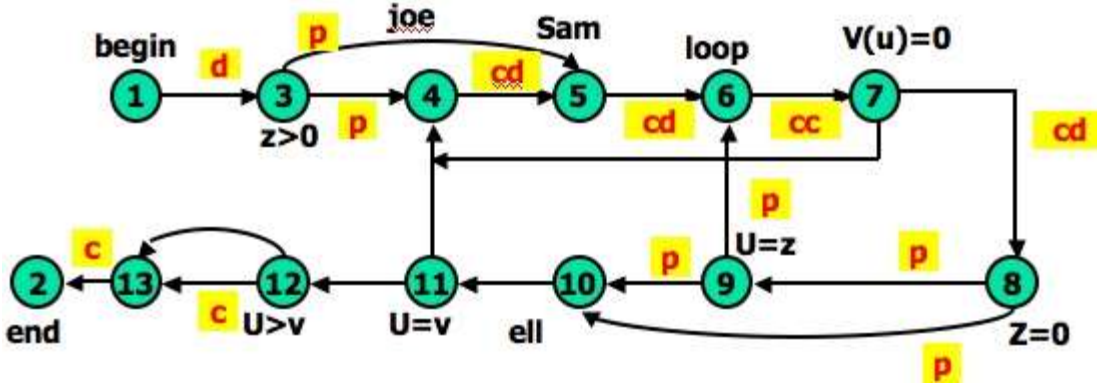

begin
d
p
joe
Sam
cd
loop
V(u)=0
1
3
4
5
6
7
z>0
p
cd
cc
cd
p
U=z
c
p
p
2
13
12
11
10
9
8
c
end
U>v
U=v
ell
Z=0
p

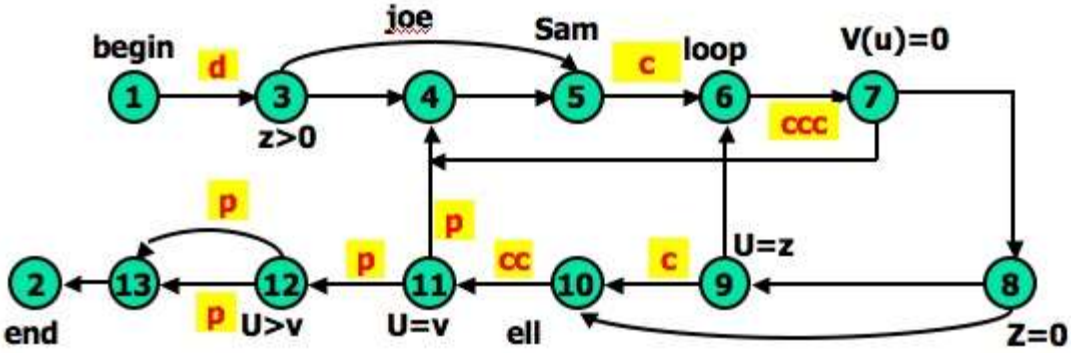

begin
d
joe
Sam
c
loop
V(u)=0
1
3
4
5
6
7
z>0
ccc
p
p
p
U=z
2
13
12
11
10
9
8
p
p
cc
c
end
U>v
U=v
ell
Z=0

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, 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 4

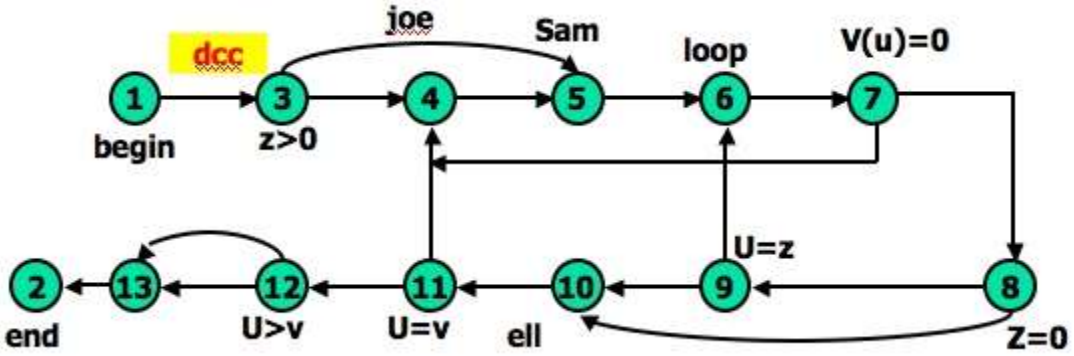

joe
Sam
dcc
loop
V(u)=0
1
3
4
5
6
7
begin
z>0
U=z
2
13
12
11
10
9
8
end
U>v
U=v
ell
Z=0

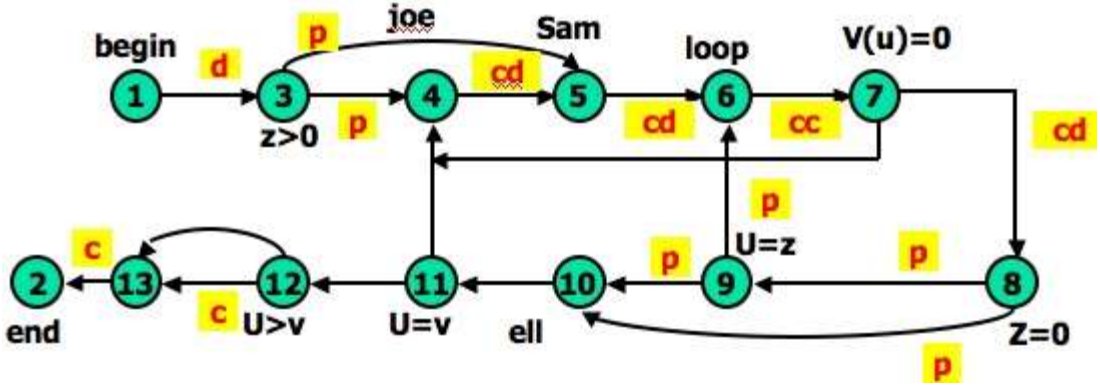

begin
d
p
joe
Sam
cd
loop
V(u)=0
1
3
4
5
6
7
z>0
p
cd
cc
cd
p
U=z
c
p
p
2
13
12
11
10
9
8
c
end
U>v
U=v
ell
Z=0
p

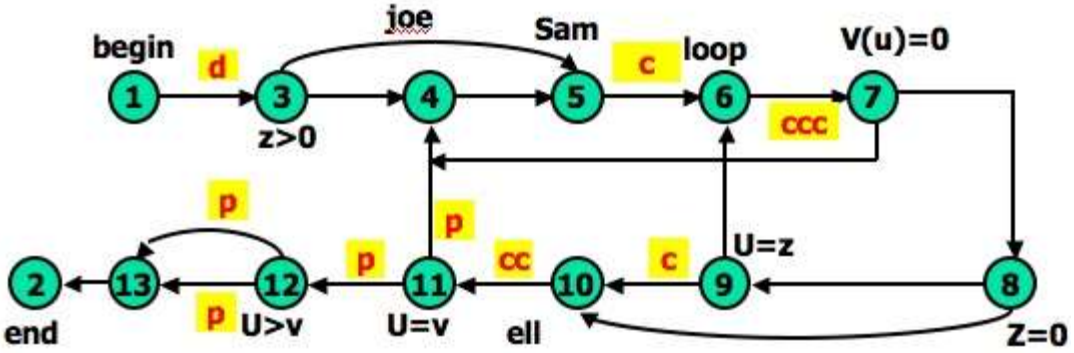

begin
d
joe
Sam
c
loop
V(u)=0
1
3
4
5
6
7
z>0
ccc
p
p
p
U=z
2
13
12
11
10
9
8
p
p
cc
c
end
U>v
U=v
ell
Z=0

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 2, 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.

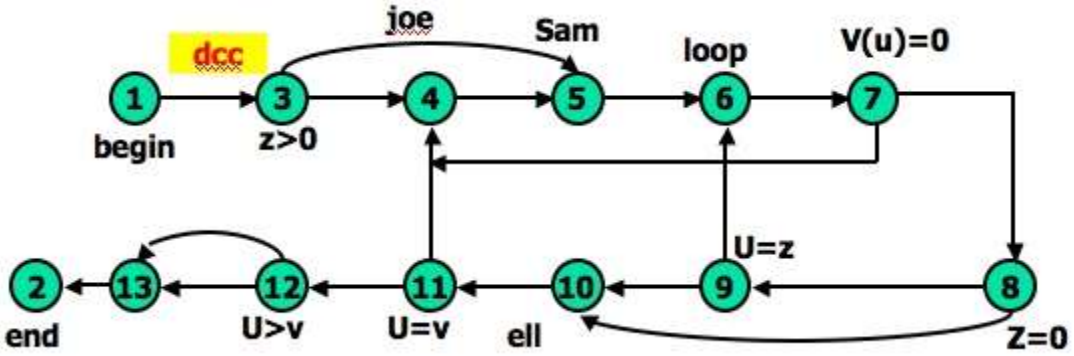

joe
Sam
dcc
loop
V(u)=0
1
3
4
5
6
7
begin
z>0
U=z
2
13
12
11
10
9
8
end
U>v
U=v
ell
Z=0

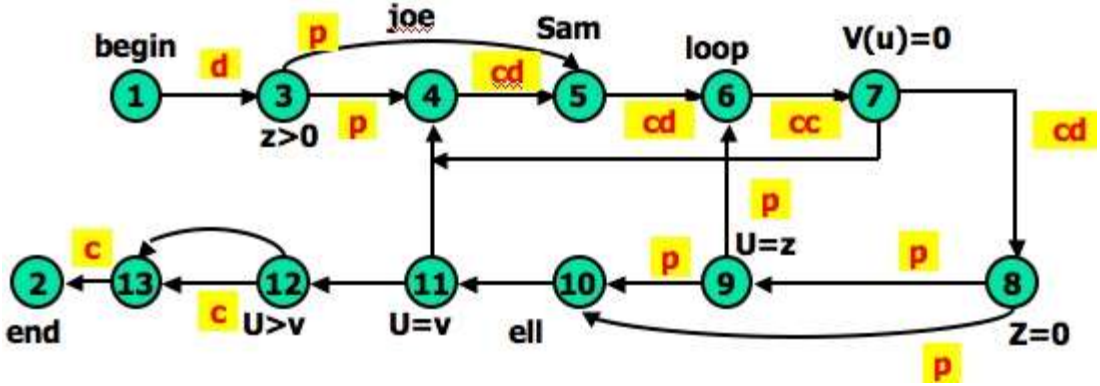

begin
d
p
joe
Sam
cd
loop
V(u)=0
1
3
4
5
6
7
z>0
p
cd
cc
cd
p
U=z
c
p
p
2
13
12
11
10
9
8
c
end
U>v
U=v
ell
Z=0
p

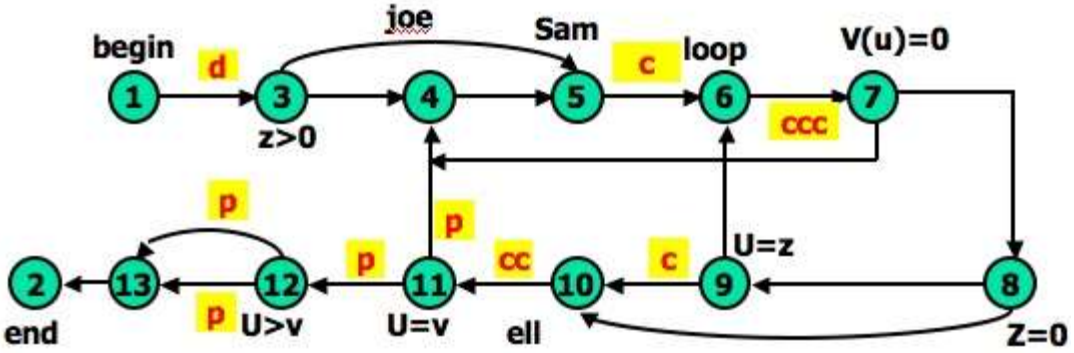

begin
d
joe
Sam
c
loop
V(u)=0
1
3
4
5
6
7
z>0
ccc
p
p
p
U=z
2
13
12
11
10
9
8
p
p
cc
c
end
U>v
U=v
ell
Z=0

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 2, 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.


joe
Sam
dcc
loop
V(u)=0
1
3
4
5
6
7
begin
z>0
U=z
2
13
12
11
10
9
8
end
U>v
U=v
ell
Z=0

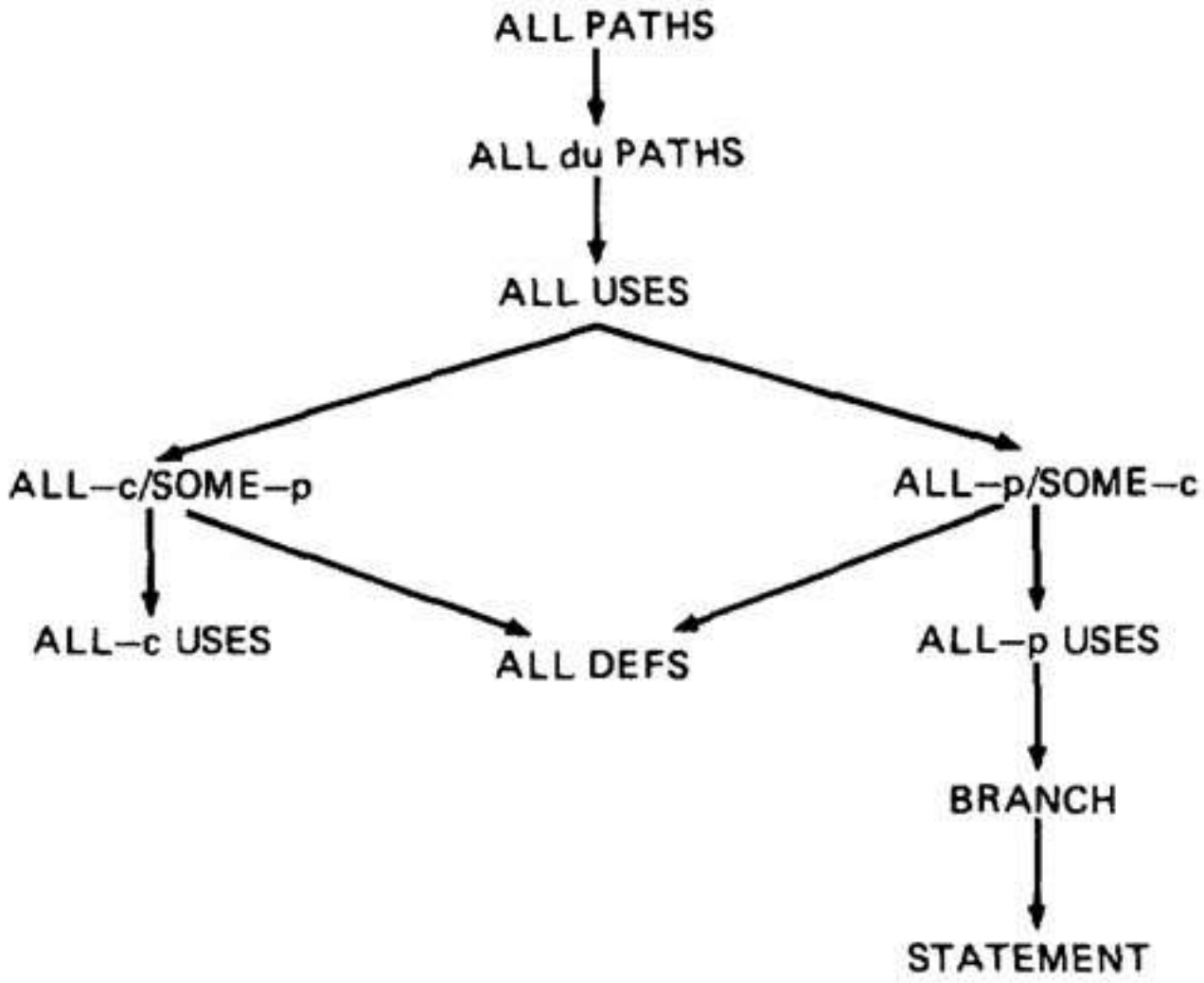

begin
d
p
joe
Sam
cd
loop
V(u)=0
1
3
4
5
6
7
z>0
p
cd
cc
cd
p
U=z
c
p
p
2
13
12
11
10
9
8
c
end
U>v
U=v
ell
Z=0
p


begin
d
joe
Sam
c
loop
V(u)=0
1
3
4
5
6
7
z>0
ccc
p
p
p
U=z
2
13
12
11
10
9
8
p
p
cc
c
end
U>v
U=v
ell
Z=0

- **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.

- **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.


ALL PATHS
ALL du PATHS
ALL USES
ALL–c/SOME–p
ALL–p/SOME–c
ALL–c USES
ALL DEFS
ALL–p USES
BRANCH
STATEMENT

DOMAIN TESTING

- **DOMAINS AND PATHS**
- 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).

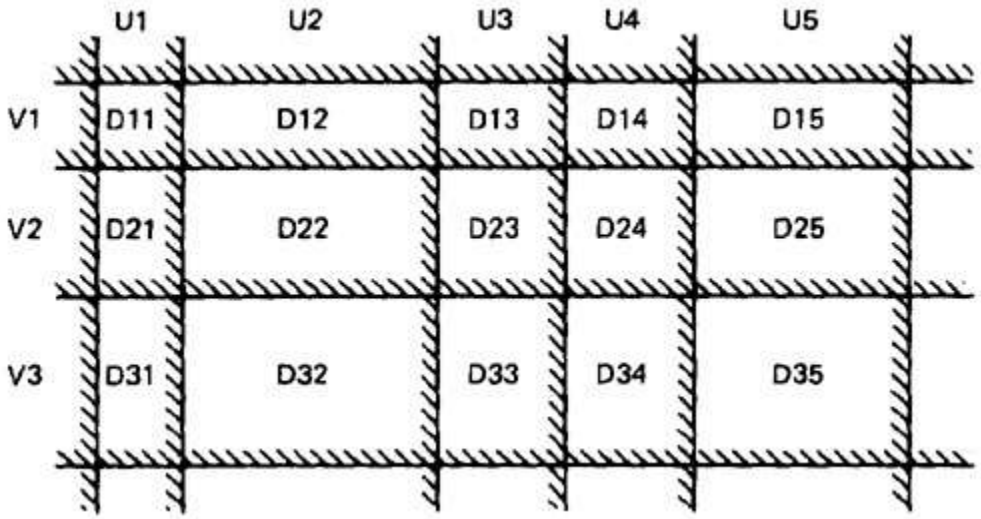
- A domain may have one or more boundaries - no matter how many variables define it.

  For example, if the predicate is x2 + y2 < 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.

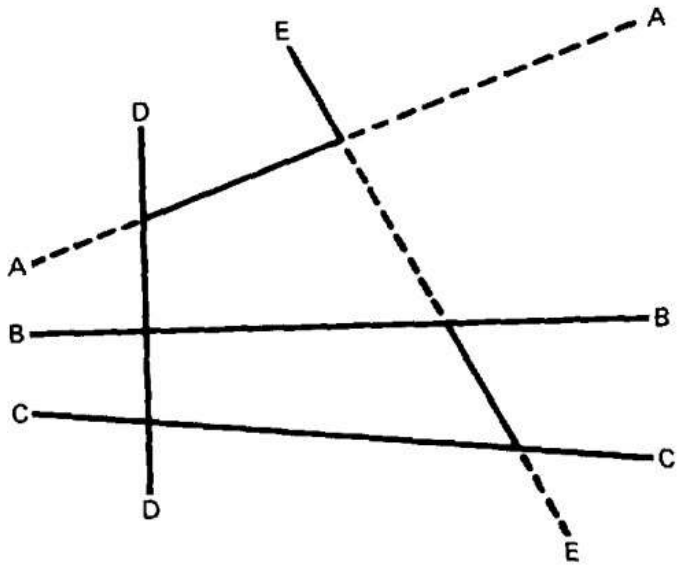
- Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.

NICE AND UGLY 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 points to determine a 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 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.



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. We want relations such as

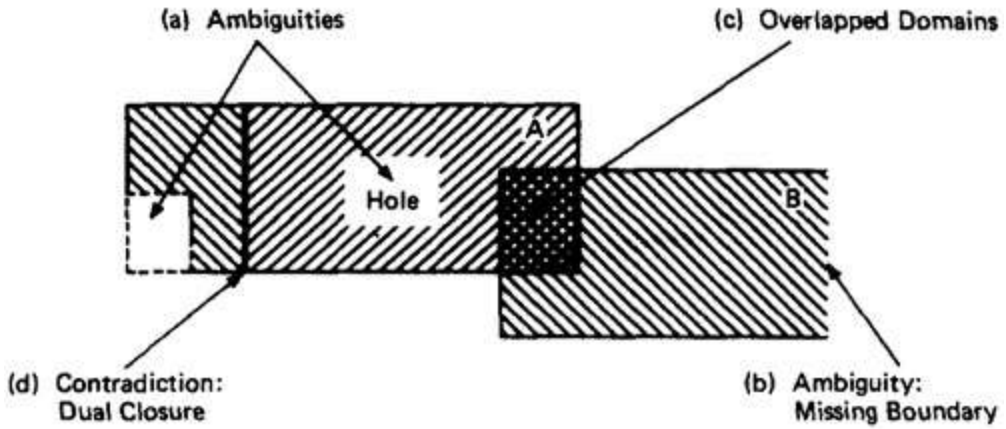
$f_1(X) >= k_1$	or $f_1(X) >= g(1,c)$
$f_1(X) >= k_2$	$f_2(X) >= g(2,c)$
.....	.....
$f_i(X) >= k_i$	$f_i(X) >= g(i,c)$

where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function over i and c that yields a constant, 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.

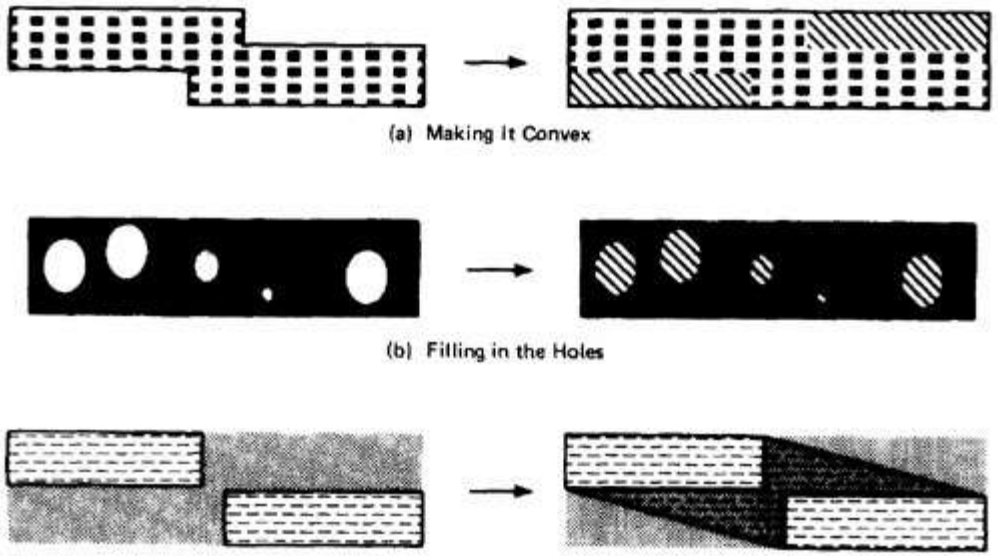
UGLY DOMAINS

- **AMBIGUITIES AND CONTRADICTIONS:** Domain ambiguities are holes in the input space.
- The holes may lie with in the domains or in cracks between domains.



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).

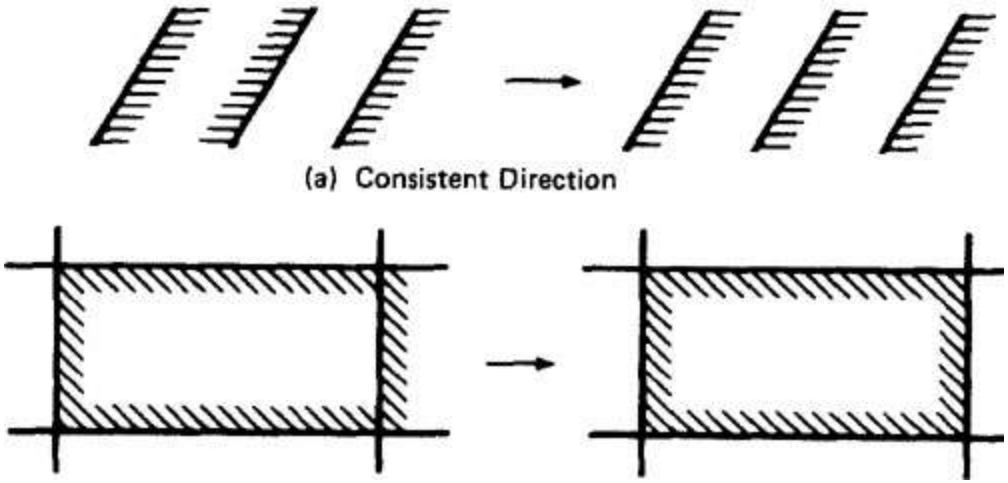


(a) Making it Convex

(b) Filling in the Holes

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)

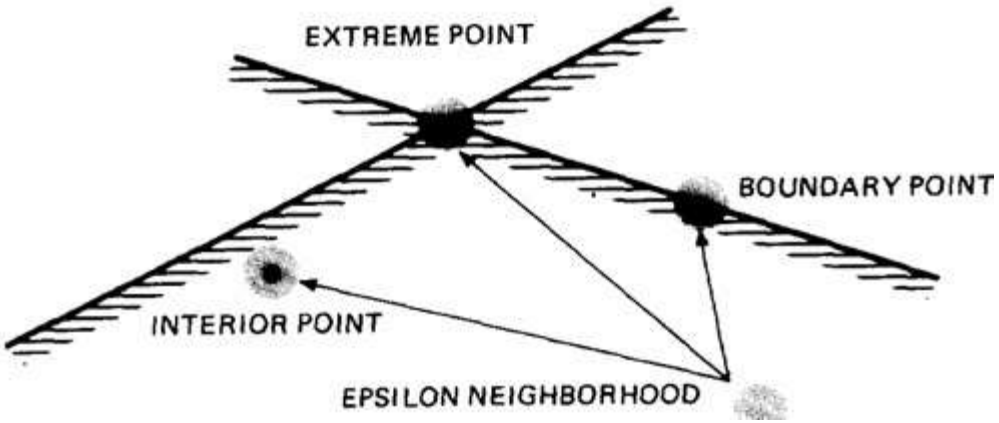


DOMAIN TESTING STRATEGY:

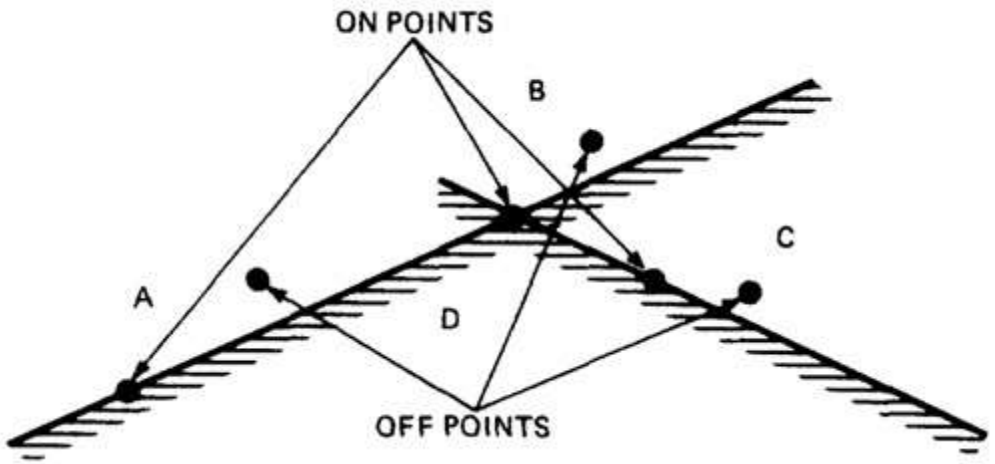
- Domains are defined by their boundaries. therefore, domain testing concentrates test points on or near boundaries.
- Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.
- Because every boundary 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.

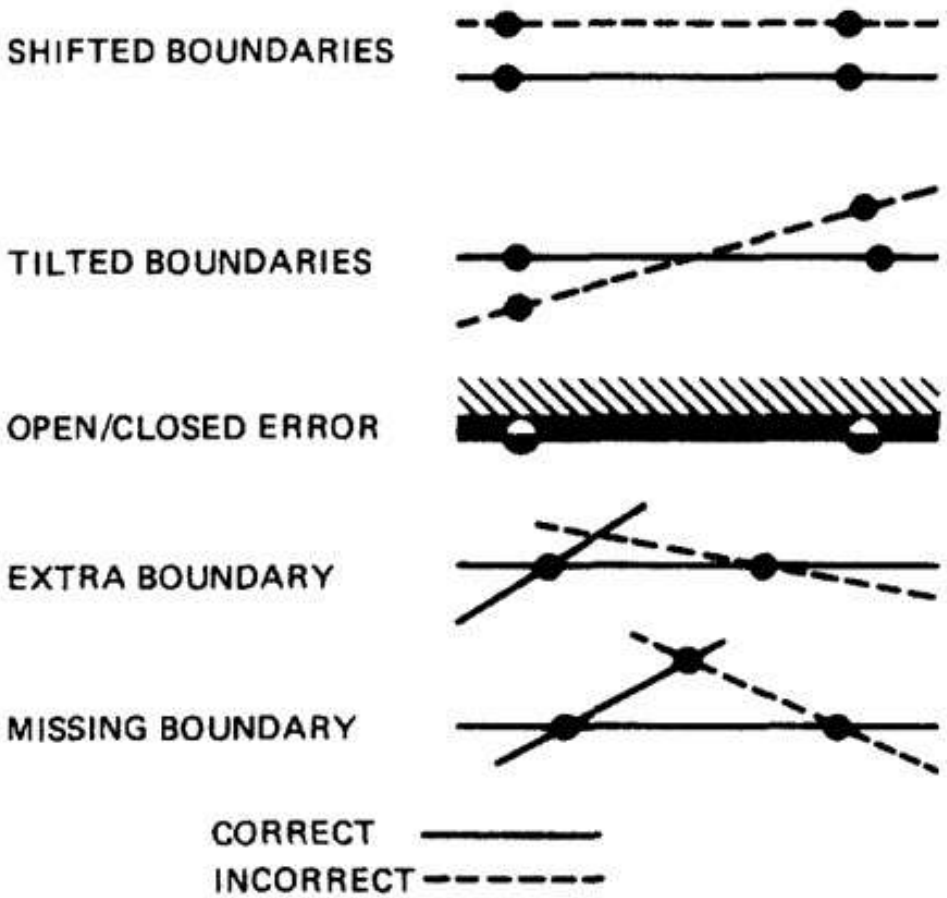
DOMAIN BUGS AND HOW TO TEST FOR THEM

- 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.



- 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.




SHIFTED BOUNDARIES
TILTED BOUNDARIES
OPEN/CLOSED ERROR
EXTRA BOUNDARY
MISSING BOUNDARY
CORRECT
INCORRECT

- **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

- **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)

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.