Chapter - 5

DATA FLOW TESTING

Data Flow Testing Basics

- 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.
- Von Neumann machines
- 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.

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.

Data Object State and Usage

- Data objects can be created, killed and used.
- They can be used in two distinct ways:
  - In a calculation
  - As a part of a control flow predicate
- The following symbols denote these possibilities.
  - d- defined, created, initialized, etc.,
  - k- killed, undefined, released.
  - u- used for some thing.
    - c- used in calculations
    - p- used in a predicate

Defined

- 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

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known.
- 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.

Usage

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

Data Flow Anomalies-continued.,

- 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 Anomalies-continued.,

- 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.

Data Flow Anomalies-continued.,

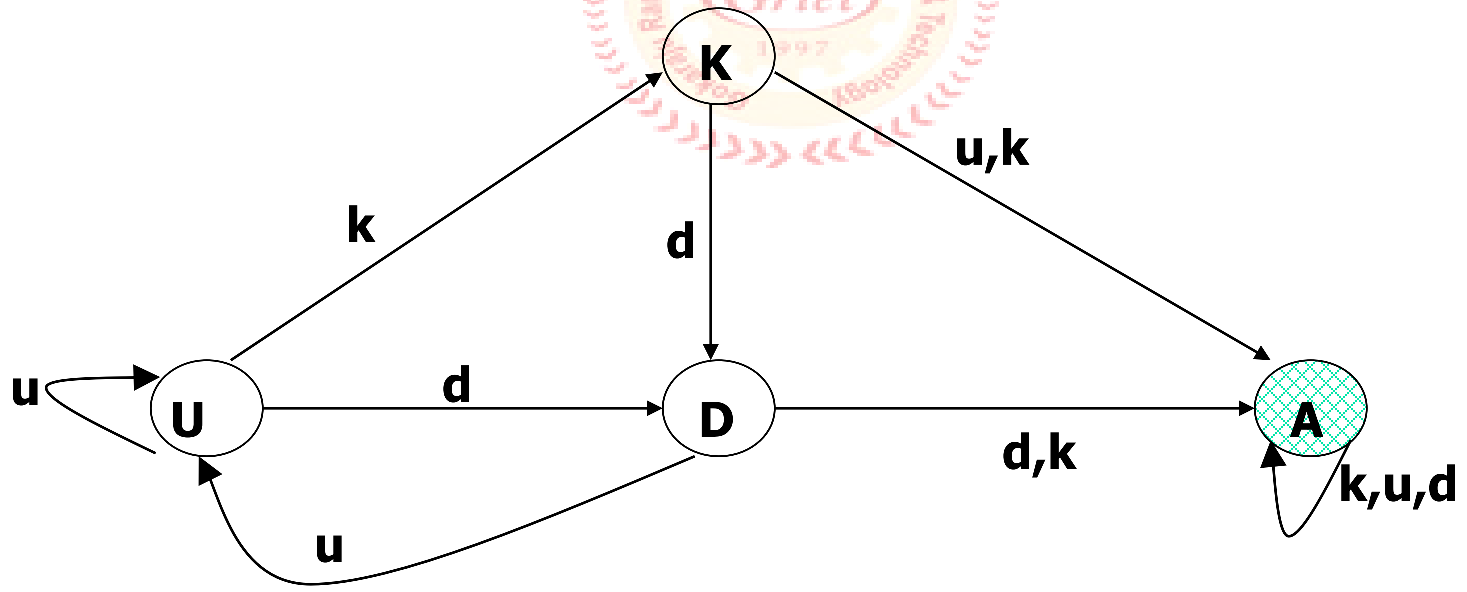
- -k: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
- -d: okay. This is just the first definition along this path.
- -u: possibly anomalous. Not anomalous if the variable is global and has been 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, does not 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 State Graph

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



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.

Static Vs Dynamic Anomaly Detection-continued.,

- If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it does not 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 isn’t static analysis enough?

- there are many things for which current notions of static analysis are **inadequate they are:**
  - Dead Variables.
  - Arrays.
  - Records and pointers.
  - Dynamic subroutine or function name in a call.
  - False anomalies.
  - Recoverable anomalies and alternate state graphs.
  - Concurrency, interrupts, system issues.
- Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods for data flow anomaly detection.

The Data Flow Model

- The data flow model is based on the program's control flow graph.
- Here we annotate each link with symbols or sequences of symbols 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

- Here are the modeling rules:
- 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 to complete the graph. The entry nodes are dummy nodes placed at entry statements for the same reason.
- The outlink of simple statements are weighted by the proper sequence of data flow actions for that statement.
- Predicate nodes are weighted with the p- use on every outlink, appropriate to that outlink.

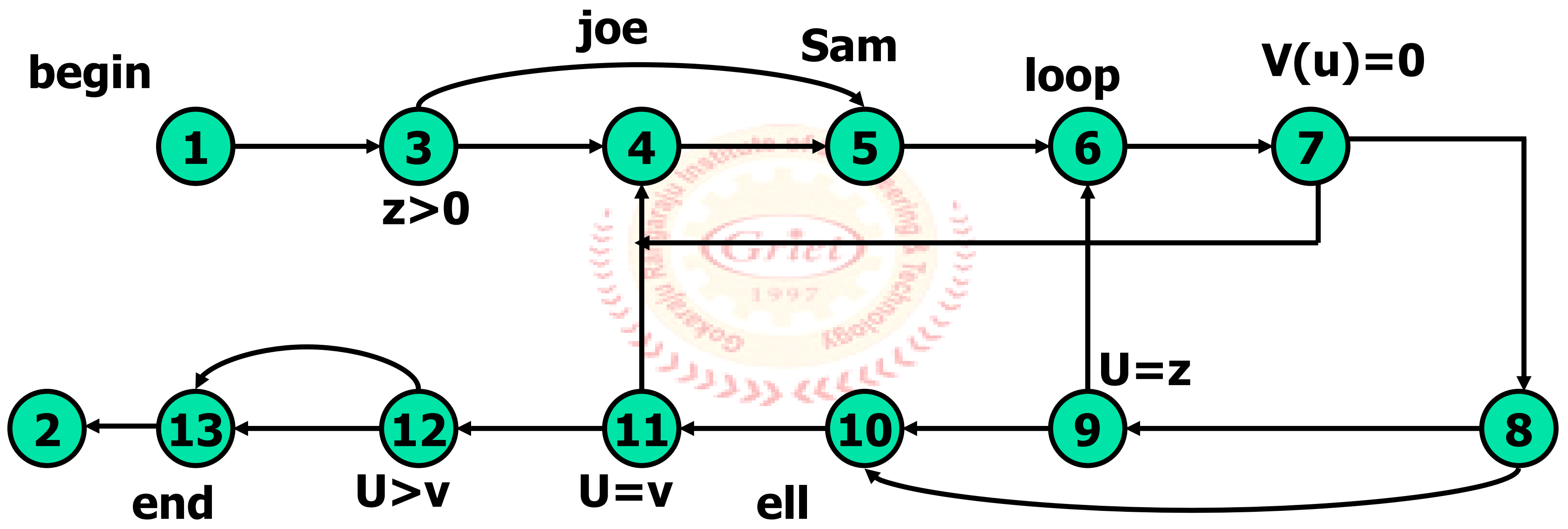
Components of the Model- continued.,

- Every sequence of simple statements 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.

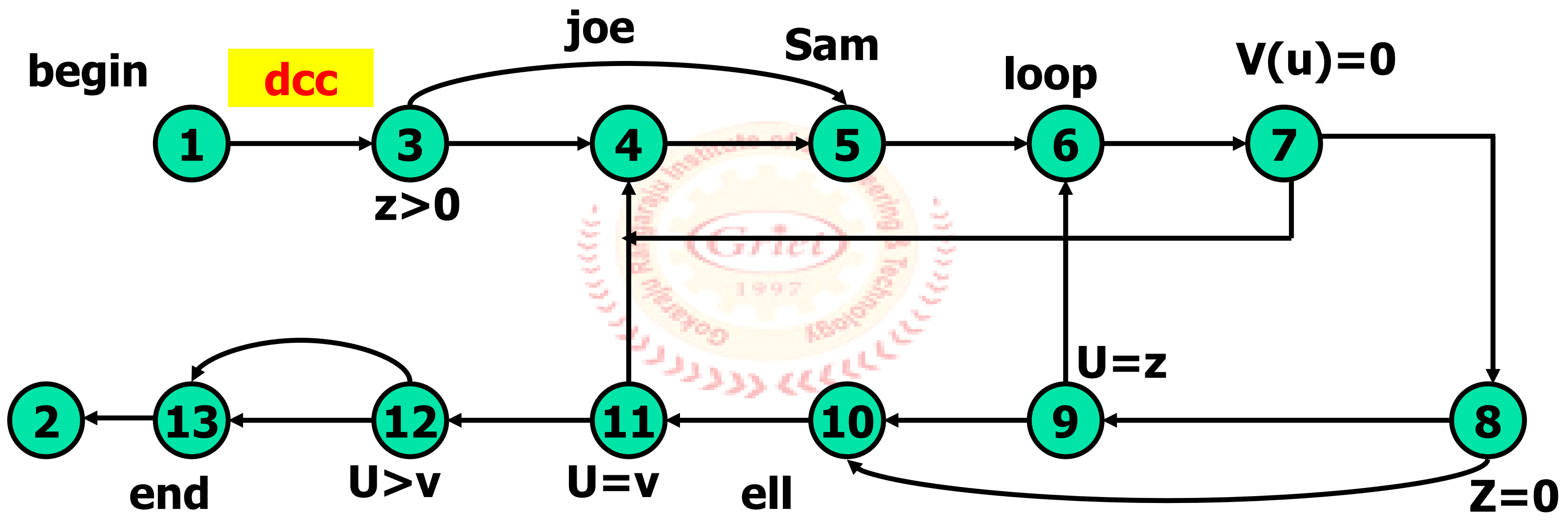
Example-Data Flow Graph

```
      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
```

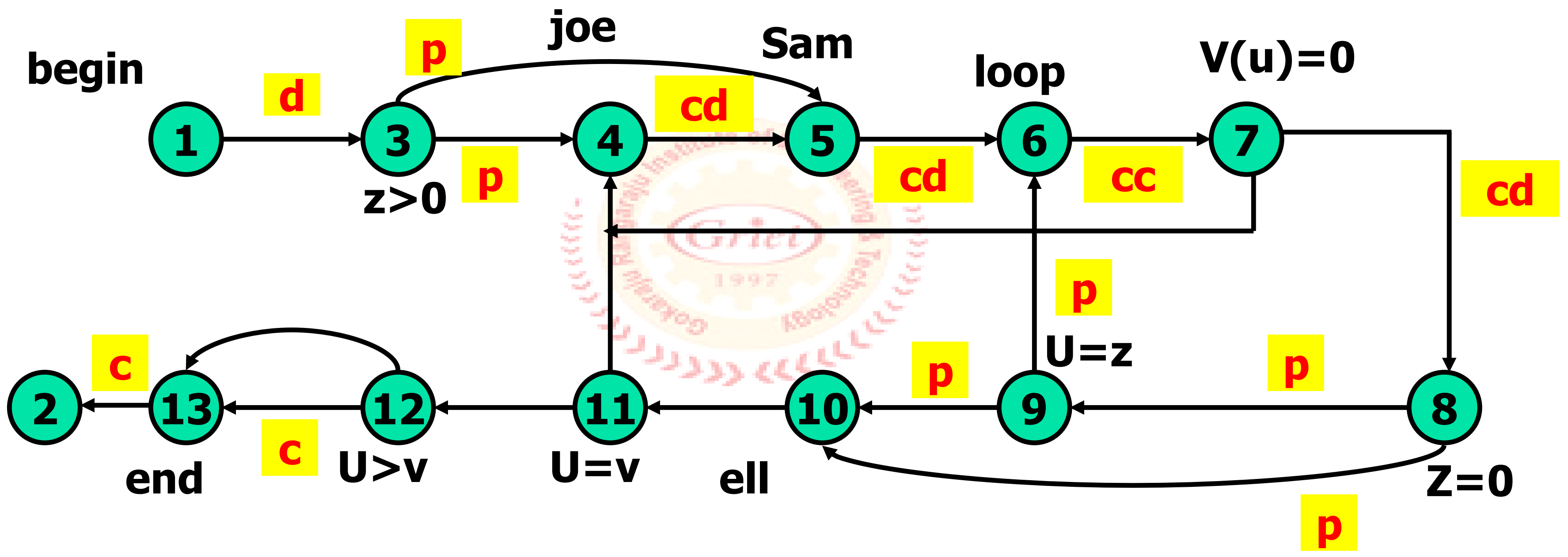
Un annotated Control Flow Graph



Control Flow Graph Annotated for X and Y Data Flows



Control Flow Graph Annotated for Z Data Flow



Data Flow Testing Strategies

- Data Flow Testing Strategies are structural strategies.
- Data Flow Testing Strategies require data flow link weights.
- Data Flow Testing Strategies are based on selecting test path segments (sub paths) that satisfy some characteristic of data flows for all data objects.

Definition Clear Path Segment

- Definition Clear Path Segment with respect to a variable X is a connected sequence of links such that X is defined on the first link and not redefined or killed on any sub sequent link of that path segment.
- The fact that there is a definition clear sub path between two nodes does not imply that all sub paths between those nodes are definition clear.

Loop Free Path Segment

- A loop free path segment is a path segment for which every node is visited at most once.

A simple path segment

- A simple path segment is a path segment in which at most one node is visited twice.

du path

- 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 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.

The data flow testing strategies

- Various types of data flow testing strategies in decreasing order of their effectiveness are:
  - All du paths Strategy
  - All uses Strategy
  - All p-uses/some c-uses Strategy
  - All c-uses/some p-uses Strategy
  - All definitions Strategy
  - All predicates uses, all computational uses Strategy

All du paths Strategy

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

All Uses Strategy

- 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.

All P-uses/some c-uses Strategy

- All p-uses/some c-uses (APU+C) strategy is defined as follows: 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.

All C-uses/some p-uses Strategy

- 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.

All Definitions Strategy

- 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.

All predicate-uses, All Computational uses 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.

Ordering the Strategies

$$\begin{tikzcd}
 & & \text{All paths} \arrow[d] & & \\
 & & \text{All du paths} \arrow[d] & & \\
 & & \text{All uses} \arrow[dl] \arrow[dr] & & \\
 & \text{All c/some p} \arrow[dr] \arrow[ddl] & & \text{all p/some c} \arrow[dl] \arrow[ddr] & \\
 & & \text{all defs} & & \\
\text{All c uses} & & & & \text{All p uses} \arrow[d] \\
 & & & & \text{Branch} \arrow[d] \\
 & & & & \text{statement}
\end{tikzcd}$$

Slicing and Dicing

- A program slice is a part of a program defined with respect to a given variable X and a statement i: it is the set of all statements that could 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 variable 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.

Chapter - 6

DOMAIN TESTING

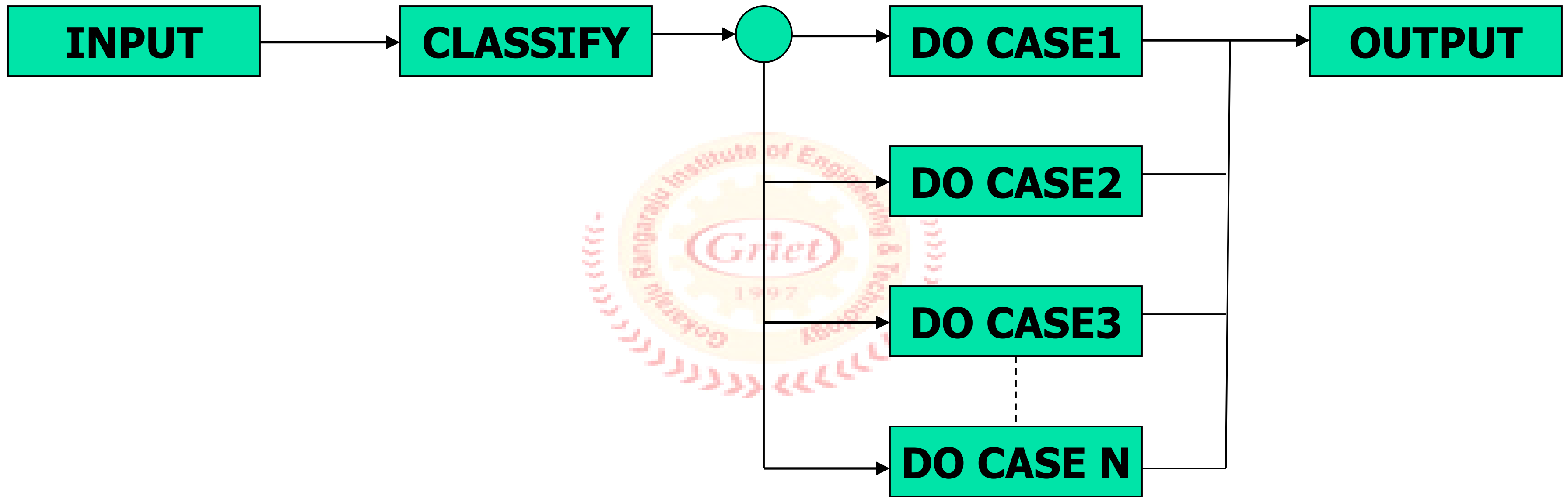
Domain Testing

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

Domains and Paths

- 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 on implementation details, it is a structural test technique.
- For example, you're doing domain testing when you check extreme values of an input variable.
- Before doing whatever it does, a routine must classify the input and set it moving on the right path.
- In domain testing we focus on the classification aspect of the routine rather than on the calculations.

Schematic representation of Domain Testing



A domain is a set

- An input domain is a set.
- If the source language supports set definitions less testing is needed because the compiler does much of it for us.
- Domain testing does not work well with arbitrary discrete sets of data objects.

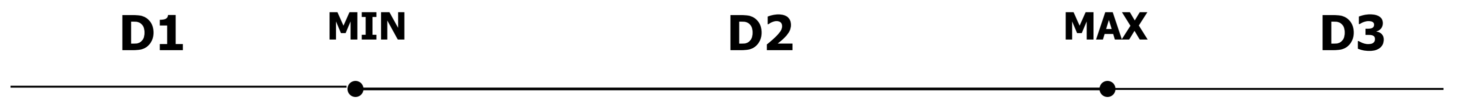
Domains, Paths, and Predicates

- In domain testing, predicates are assumed to be interpreted in terms of input vector variable.
- 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 by their boundaries.
- For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.
- 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.

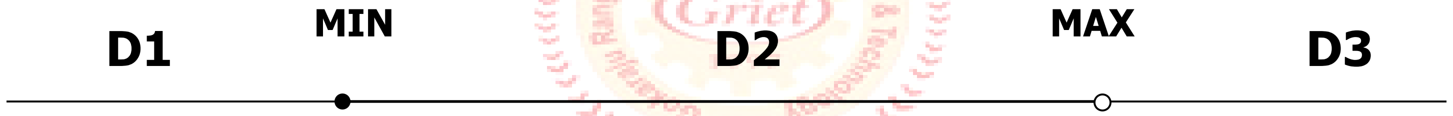
Domain Closure

- 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.

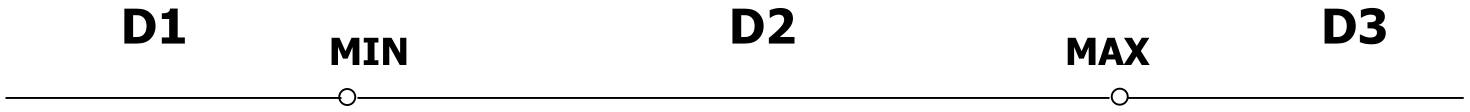
Open and Closed Domains



BOTH SIDES CLOSED



ONE SIDE OPEN



BOTH SIDES OPEN

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 hyper planes.

The 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.

Domain Errors

- Double zero representation: In computer 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 it self or multiplied by zero. So the floating point zero check to be done against a epsilon value.
- Contradictory domains: 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.

Domain Errors-continued.,

- Over specified domains: the domain can be overloaded with so many contradictions that the result is null domain.
- Boundary errors: Errors caused in and around the boundary of a domain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.
- Closure reversal: Some times the logical complementation can be done improperly. For example, complementation of <= can be wrongly implemented as >= instead of >.
- Faulty logic: compound predicates are subject to faulty logic transformations and improper simplification. If the predicate define domain boundaries, all kinds of domain bugs can result from

Nice Domains

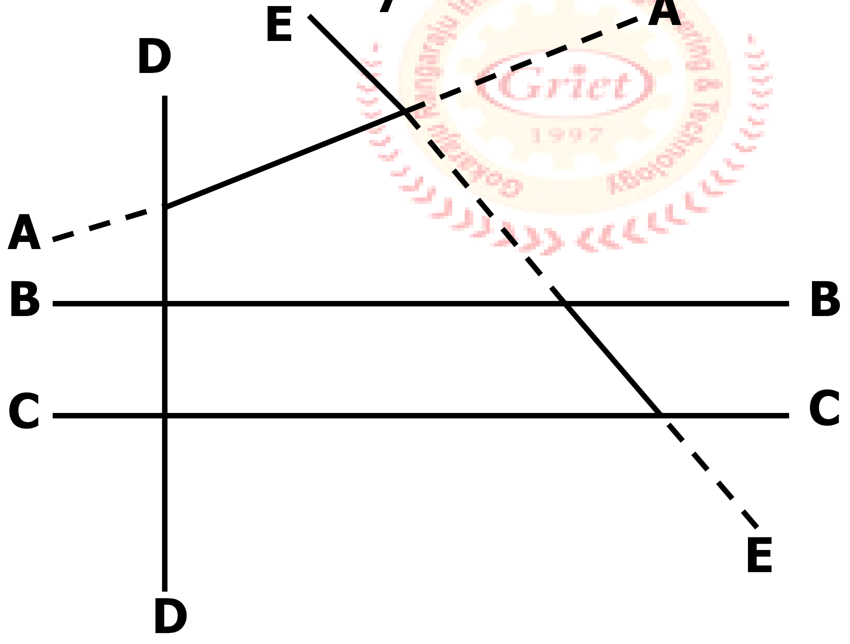
- Some important properties of nice domains are:
- Linear, complete, systematic, orthogonal, consistently closed, simply connected and convex.
- To the extent that domains have these properties domain testing is easy as testing gets.
- The bug frequency is lesser for nice domain than for ugly domains.

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.



Systematic Boundaries

- By systematic boundary means that boundary inequalities related by a simple function such as a constant.
- Example:

  $f_1(x)>=k_1$ or $f_1(x)>=g(1,c)$

  $f_2(x)>=k_2$ or $f_2(x)>=g(2,c)$

  ……………………………………………

  $f_i(x)>=k_i$ or $f_i(x)>=g(i,c)$
- Where fi is an arbitrary 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.

Orthogonal Boundaries

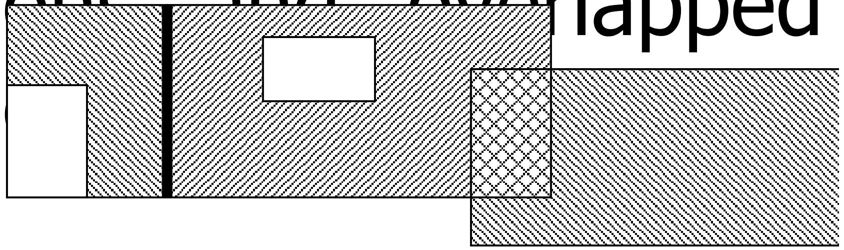
- If two boundary sets U and V are said to be orthogonal if every inequality in V is perpendicular to every inequality in U.

Ugly Domains

- Some domains are born ugly and uglified by bad specifications.
- Every simplification of ugly domains by programmers can be either good or bad.

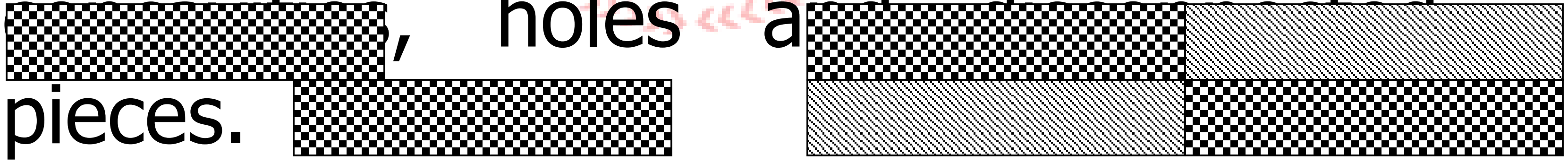
Ambiguities and Contradictions

- Domain ambiguities are holes in the input space.
- The holes may lie with in the domains or in cracks between domains.
- Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specificati

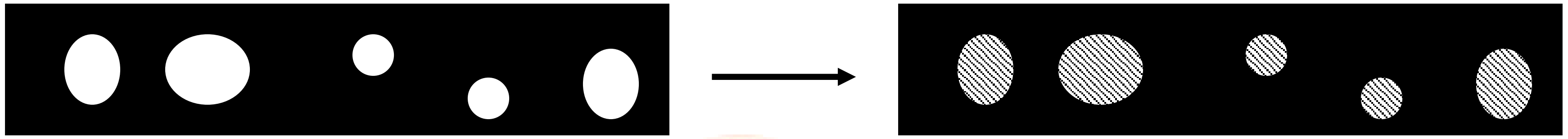


Simplifying the Topology

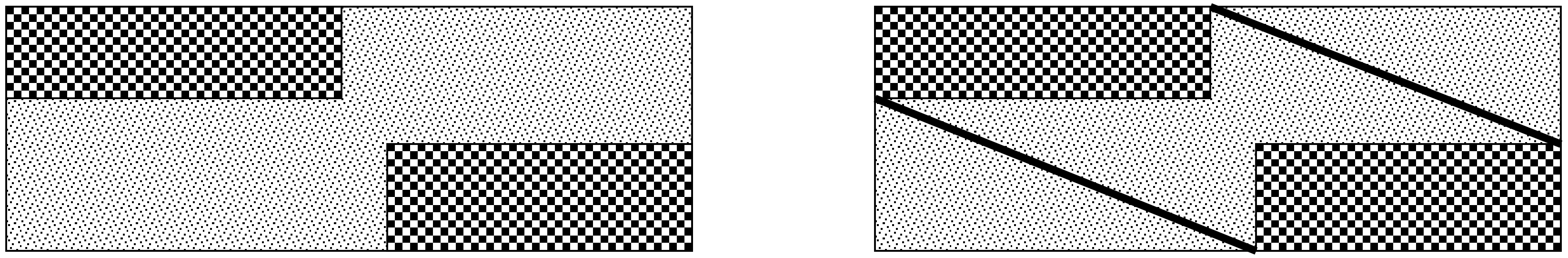
- The programmer's and tester's reaction to complex domains is the same- simplify.
- There are three generic cases: concavities, holes and disconnected pieces.


Making it convex

Simplifying the Topology



Filling in the holes



Simplifying the topology

Domain testing overview

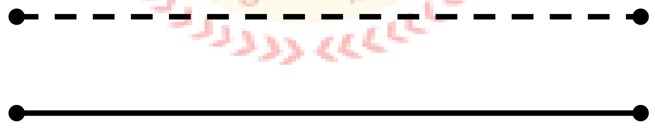
- Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
- Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all categorized kinds of boundary errors.
- Run the tests by post test analysis determine if any boundaries are faulty and if so, how.
- Run enough tests to verify every boundary of every domain.

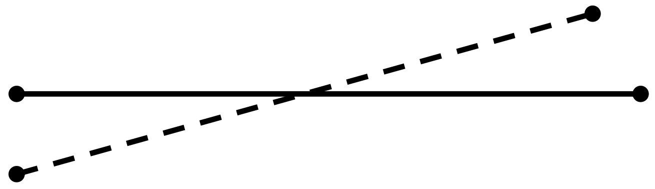
Domain bugs and how to test for them

- An interior point is a point in the domain such that all points with in an arbitrarily small distance 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 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 domain is open, an off point is a point near the boundary but in the domain being tested.

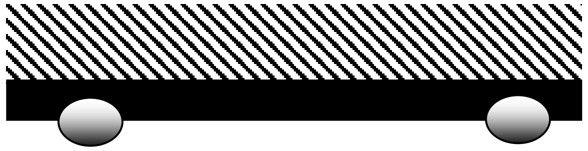
Generic Domain Bugs

- The generic domain Bugs are: shifted boundaries, tilted boundaries, open/closed errors, extra boundary and missing boundary.

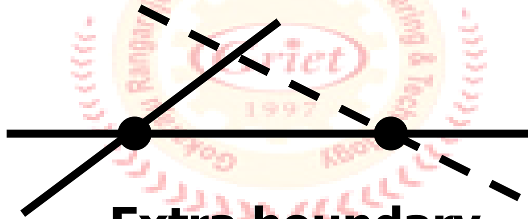

Shifted boundaries


Tilted boundaries

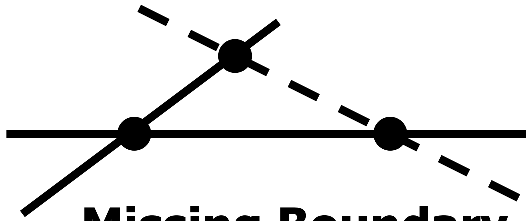
Generic Domain Bugs-continued.,



Open/closed error

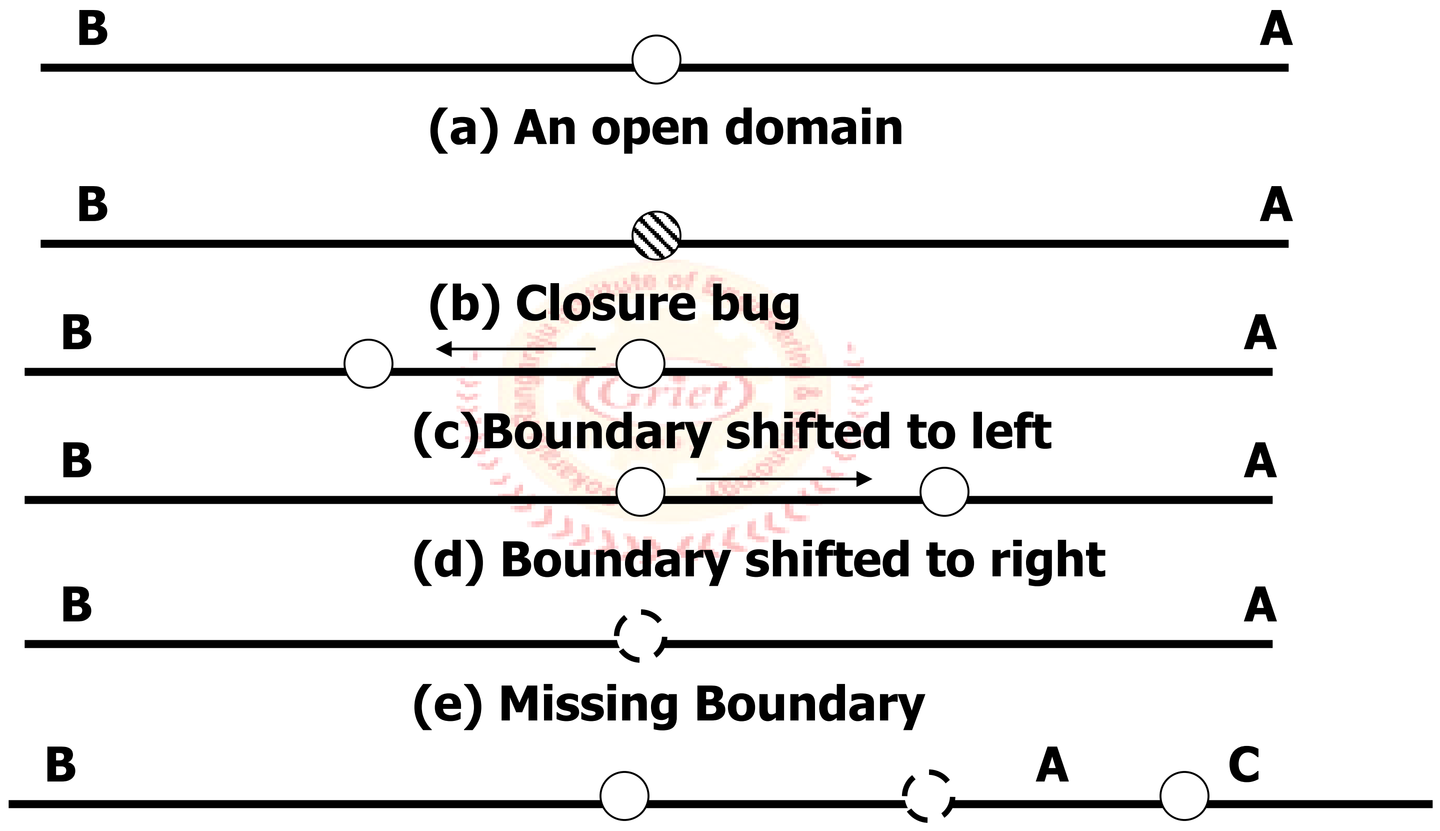


Extra boundary



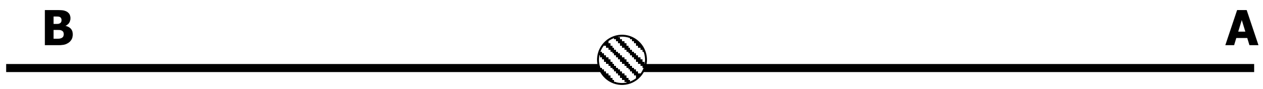
Missing Boundary

Testing one dimensional domains-open boundaries

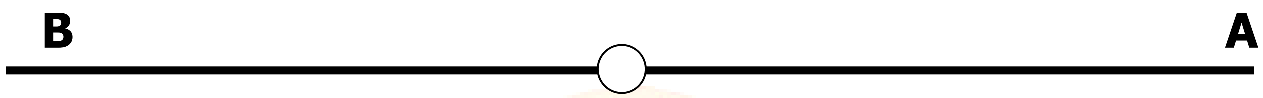


(f) Extra Boundary

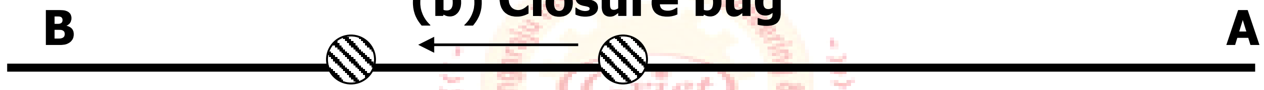
Testing one dimensional domains-closed boundaries



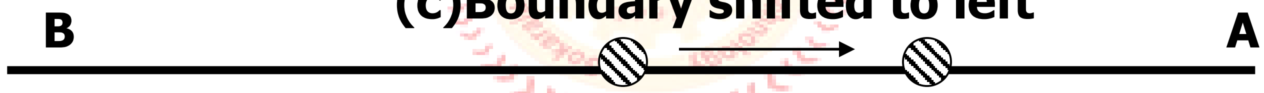
(a) An closed domain



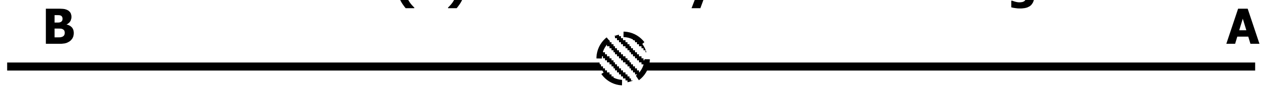
(b) Closure bug



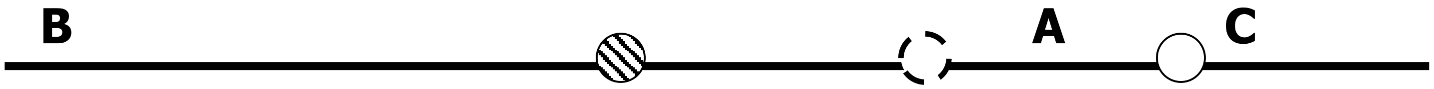
(c)Boundary shifted to left



(d) Boundary shifted to right



(e) Missing Boundary



(f) Extra Boundary

Procedure for Testing

- The procedure is conceptually is straight forward.
- It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.
- 1. identify input variables.
- 2. identify variable which appear in domain defining predicates, such as control flow predicates.
- 3. interpret all domain predicates in terms of input variables.
- 4. for p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. the result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. for example ABC+DEF+GHI….. Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connected domains.
- 5. Solve these inequalities to find all the extreme points of each domain using any of the linear programming methods.