UNIT-II

Transaction flow Testing and Data flow Testing

1.1 Transaction flows:-

A transaction is defined as a set of statements or a unit of work handled by a system user. It is actually a collection of operations that are required for handling a particular transaction.

Each transaction is usually associated with an entry point (or a starting point) and an exit point (or a terminating point). The execution of a transaction begins at the entry point and ends at an exit point thereby producing some results.

Every transaction related to an on-line information system consists of 12 steps.

1. At the 'Starting point', it accepts the input.
2. Validates the input.
3. After validating, an acknowledgement is sent back to the user.
4. Input processing is performed.

5. If necessary, searches a file.

6. User processes the request.

7. Accepts the input.

8. Validates the input.

9. Requests are further processed.

10. Updates the file.

11. The output is then transmitted.

12. The output is fed in the form of records and the transaction is cleaned up from the system.

The user processes these steps as a single transaction. For on-line processing many kinds of operations are required.

For example, in an automatic teller machine several operations take place such as withdrawing money, depositing transferring money, paying bills etc. The same operations can be done by other transactions also i.e for savings account checking account etc.

1.1.2 Transaction flows and how they occurs:-

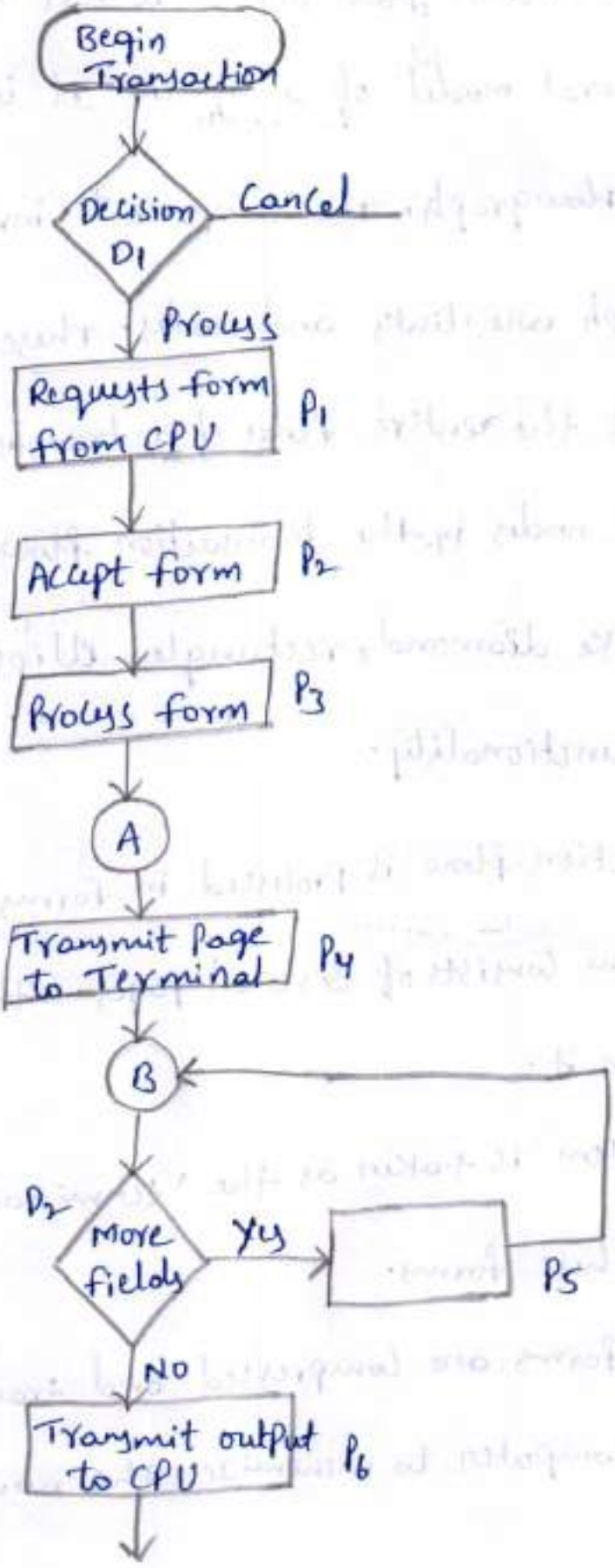
The transaction flow occurs to test the structural and behavioural model of a system. It is similar to the control flowgraph. The components involved in transaction flowgraph are links and nodes. These nodes and links represent the entire flow of a transaction.

The nodes in the transaction flowgraph have different shapes like diamond, rectangle, ellipse, circle to show their functionality.

A transaction flow is processed in 'forms'.
Each form consists of several pages with records and fields in it.

* A system is taken as the 'terminal controller' to process these forms.
* Long forms are compressed and transmitted by the central computer to minimize the number of records in it.

The following flowgraph illustrate the processing of a transaction using forms.



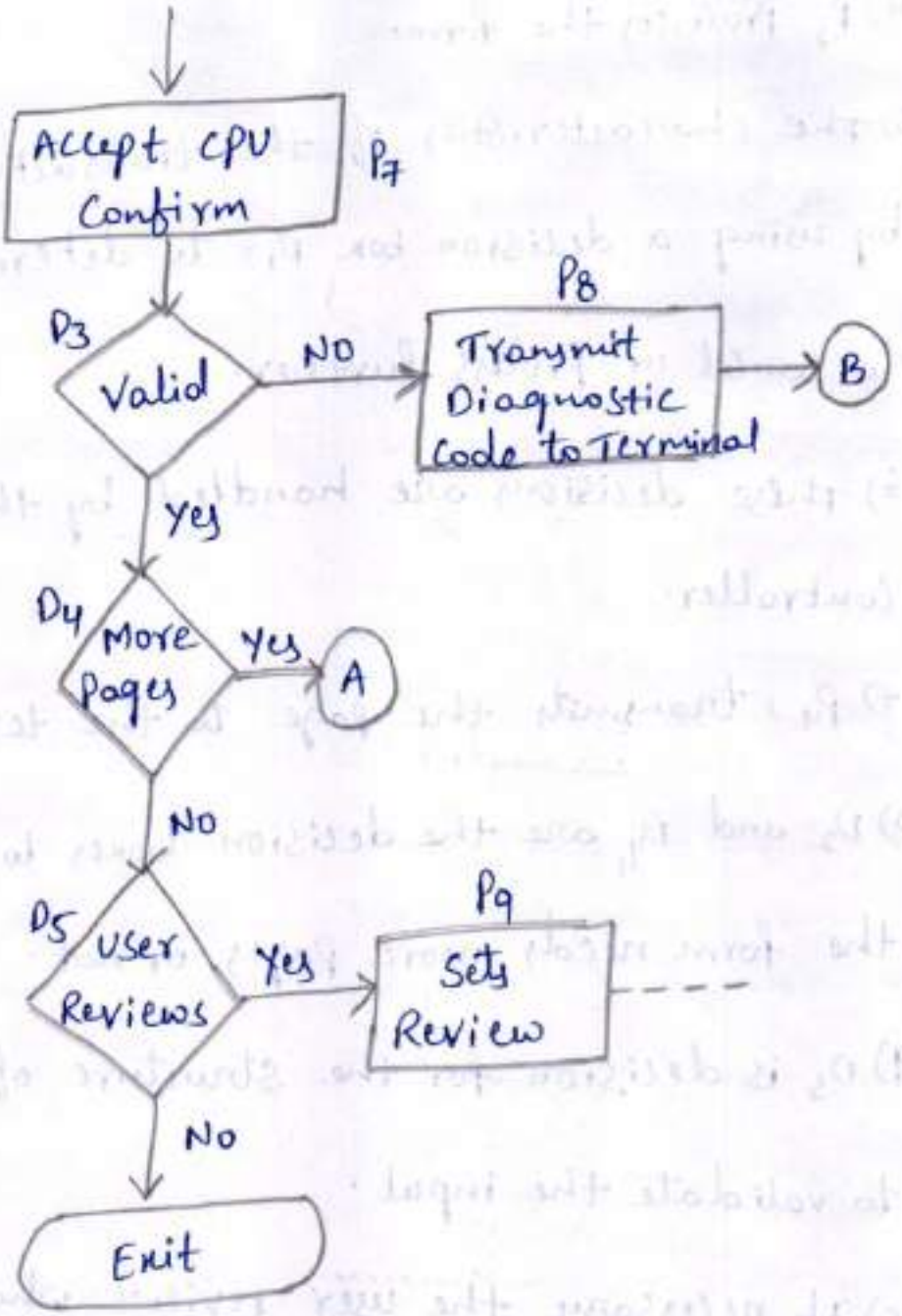


fig: Transaction flowgraph

1) Above flowgraph shows the flow of transaction.

2) When the transaction is to be initiated, the process P_1 requests forms from CPU.

3) The central computer accepts the form in the process P_2.

4) P_3 Processes the form.

5) The characteristics of the transactions are shown by using a decision box D_1, to determine whether to cancel or process further.

6) These decisions are handled by the terminal controller.

7) P_4, transmits the page to the terminal.

8) D_2 and D_4 are the decision boxes to know whether the form needs more pages or not.

9) D_3 is decision for the structure of the form, to validate the input.

10) If necessary, the user reviews whole system in Process P_9.

11) The central computer then transmits a diagnostic code back to the terminal controller in P_8.

12) After reviewing, the transaction flow is closed and exit operation is performed.

Applications of Transaction flows:-

i) Transaction flows are essential for defining requirement of complicated systems.

ii) Transaction flows are especially used in on-line system and ATM system.

iii) Transaction flows are also used in large systems such as an air traffic control or an airline reservation system.

1.1.3 Complications:-

Transaction flows don't have a good structured design for code. The parts of transaction flows result in problems like error conditions, failures, malfunctions, error recovery actions etc. These errors are unstructured.

A transaction can give birth to others and can also merge with others in many of the systems. From the time they are created to the time they are completed transaction flows have a unique identity.

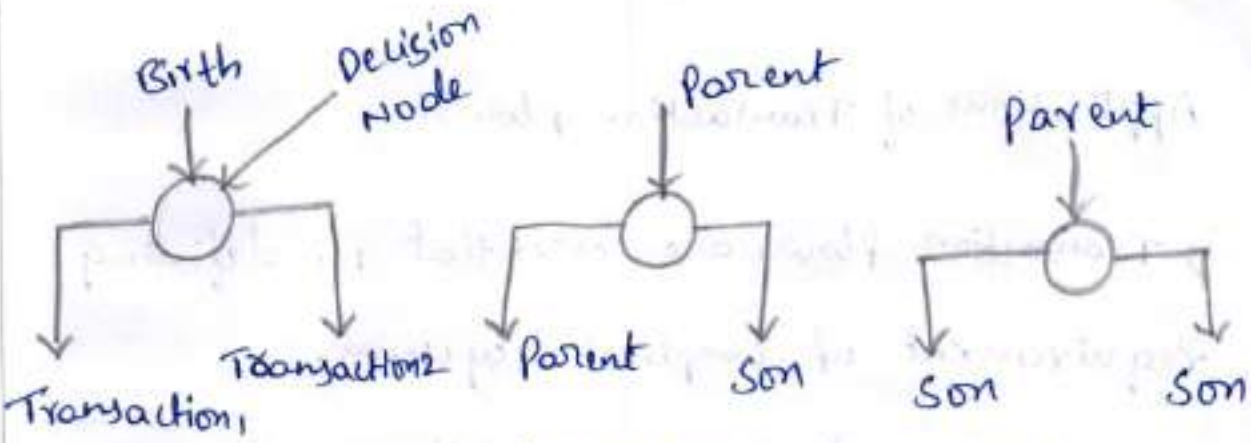


fig: Decision Nodes with Multiple out links.

i) Birth of New Transactions:-

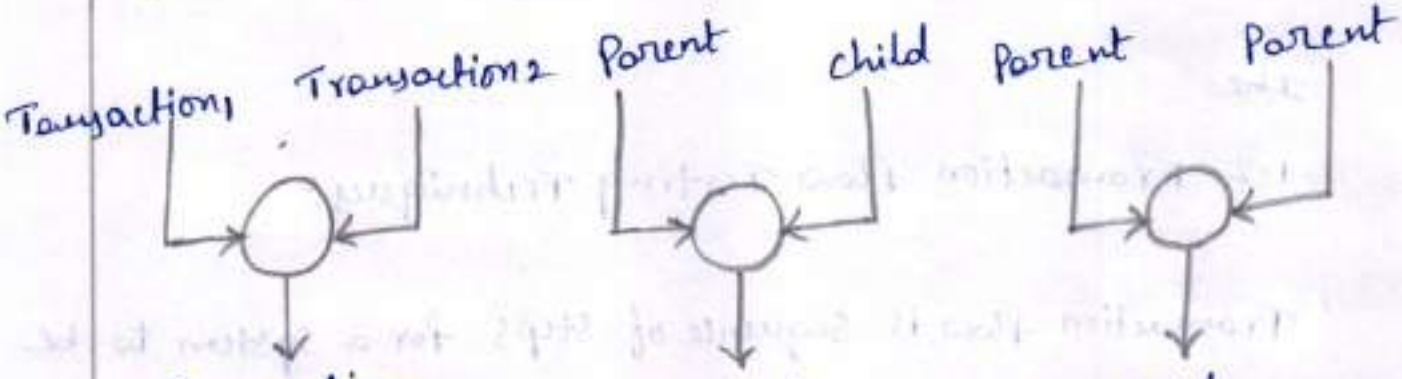
figure (a), (b), (c) shows three different representations of decision nodes. In fig (a), a transaction (Birth) has been created. The inbound transaction (Birth) at decision node gives birth to two new transactions. The two transactions, transaction 1 and transaction 2 has a different or same identity.

In fig (b) the parent transaction gives birth to two new transactions. One transaction has the same identity as 'parent', the other transaction results in a different identity 'Son'. This situation is called as 'Biosis'.

In fig(c) the parent transaction is destroyed and two new transactions (son) are created of the same identity. this situation is termed as 'Mitosis'.

ii) Transaction Merging :-

Merging is as troublesome as transaction flow splitting. The two transactions are merged at decision node giving a new transaction with the same or different identity.



a) Junction b) Absorption c) Conjugation

fig: Transaction flow junctions and mergers

In fig(a) transaction 1, transaction 2 merge at a junction resulting in a single 'transaction'.

fig(b) shows a different merging. The 'Parent' transaction and 'child' transaction gets merged at decision node. The parent absorbs its identity and therefore, results in a new (Parent) transaction.

fig(c) the two (parent) transactions of similar identity get merged at decision node resulting in conjugation. Therefore, this conjugation results in 'child' transaction.

~~The~~

1.2 Transaction flow Testing Techniques :-

Transaction flow is sequence of steps for a system to be tested. The transaction flow begins at the preliminary design and continues as the project progresses.

The following testing strategies has been taken.

1) Inspections :- Inspections are the most cost effective quality Processes that holds for testing. The Process include different phases such as "design phase, coding, testing, desk checking and debugging".

Inspection is a methodology that humans can do and the computers can't. Some of the examples are,

i) checking the syntax errors:-

There wouldn't be any syntax errors after inspection if humans do syntax checking.

ii) Referring the program:- The language processor cross check the undeclared variables, uninitialized values and the tables that are not been referred in the program.

2) Reviews:- Reviews are used to check the semantics (meaning) of a document. The quality of the documents can be examined and assured by eliminating the defects and mistakes from the documents. These inconsistencies can be found after the completion of document.

i) Review Document Information:-

The reviewed document must be used to decide if a particular statement is correct or wrong.

3) Walkthroughs:-

A walkthrough is an informal review method compared to the other types of reviews. It is a way of finding defects or problems or anomalies in the written documentation.

1.3 Basics of Data flow Testing:-

1.3.1 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.

1.3.2 Data flow Machines - Von Neumann Machines:-

Von Neumann Machines:- The computers used today are Von Neumann machines. The low cost computational and memory elements have made machines that can break time. The Von Neumann architecture is a design model that uses a Central Processing unit and a Strong Structure.

This model holds instructions and data in it. The input and output are instructed into the Arithmetic Logic Unit (ALU) to perform operations. Each instruction in this architecture executed one by one (instruction by instruction) Therefore this model has a sequential flow of steps.

1) Fetch instruction.
2) Interpret instruction.
3) Fetch operands
4) Execute instruction.
5) Store results in register.
6) Increment program Counter.
7) Repeat the prolys from step1.

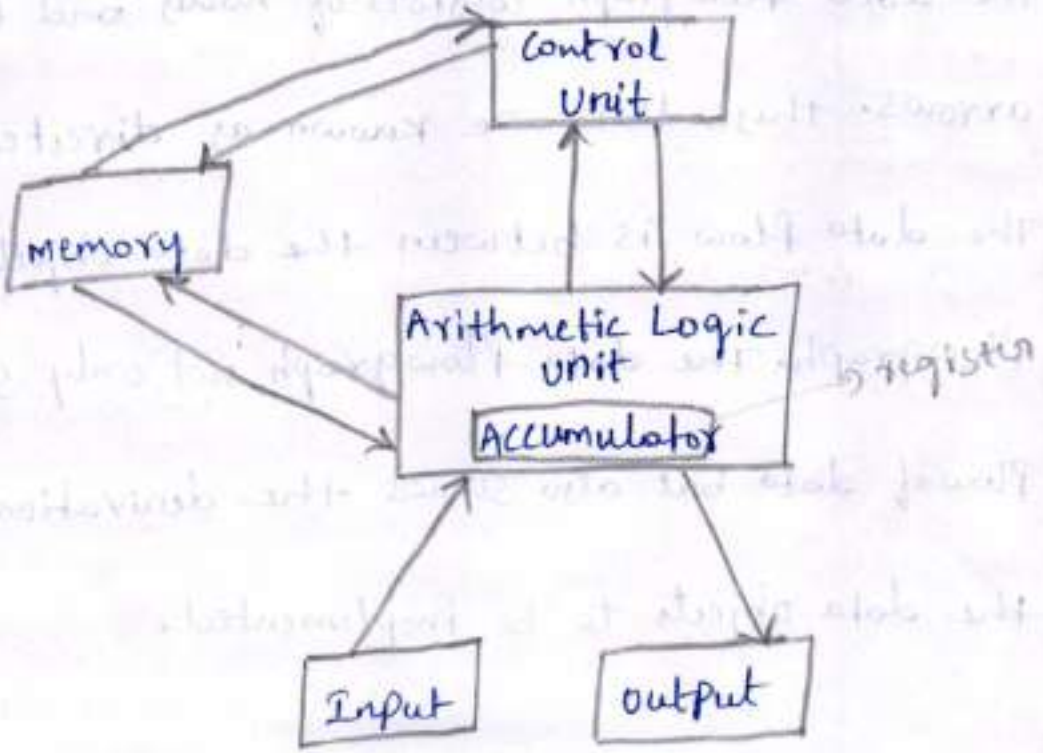


Fig: General Architecture of a Von Neumann Machine.

Multi Instruction Multi Data (MIMD) Machines :-

MIMD machines are massively parallel machines. They fetch several instructions in parallel. Therefore they have several mechanisms for executing the above steps 1-7. MIMD machines can also perform arithmetic or logical operations simultaneously. for a MIMD machine, the instructions are produced in sequential flow. The parallel machine is MIMD machine with multiple processors and sequential machine is Von Neumann machine with only one processor.

1.3.3 Data Flowgraphs :-

The data flowgraph consists of nodes and links having arrows. These links are known as directed links. The data flow is between the data objects in the data flowgraph. The data flowgraph not only shows the flow of data but also shows the derivation between the data objects to be implemented.

1.3.4 Data object State and Usage:-

Data objects have three stats. They are,

1) Data objects can be created.
2) They can be killed.
3) They can be used.

Data objects can be used in two different ways,

i) In calculation part.
ii) In the control flowgraph.

To denote the data objects state and their use, various symbols have been assigned. They are as follows,

d - define object, create and initialize
K - Killed object, not defined object, release.
u - Use object for some purpose.
c - used in the calculation part.
p - used in a predicate for operation purpose.

Every symbol in data flowgraph has a meaning. Each symbol is described below.

1) The data flow anomaly starts in 'K' state.

2) An attempt is made to use an undefined variable. Hence, it goes in an anomalous (A) state.

3) The killed (K) state defines a variables d in defined (D) state.

4) If a variable is killed from a defined (D) state then it becomes anomalous.

5) The variable u is used in U state and is redefined d in D state.

6) Variable K get killed in K state.

~~138 Shooting Data flow~~

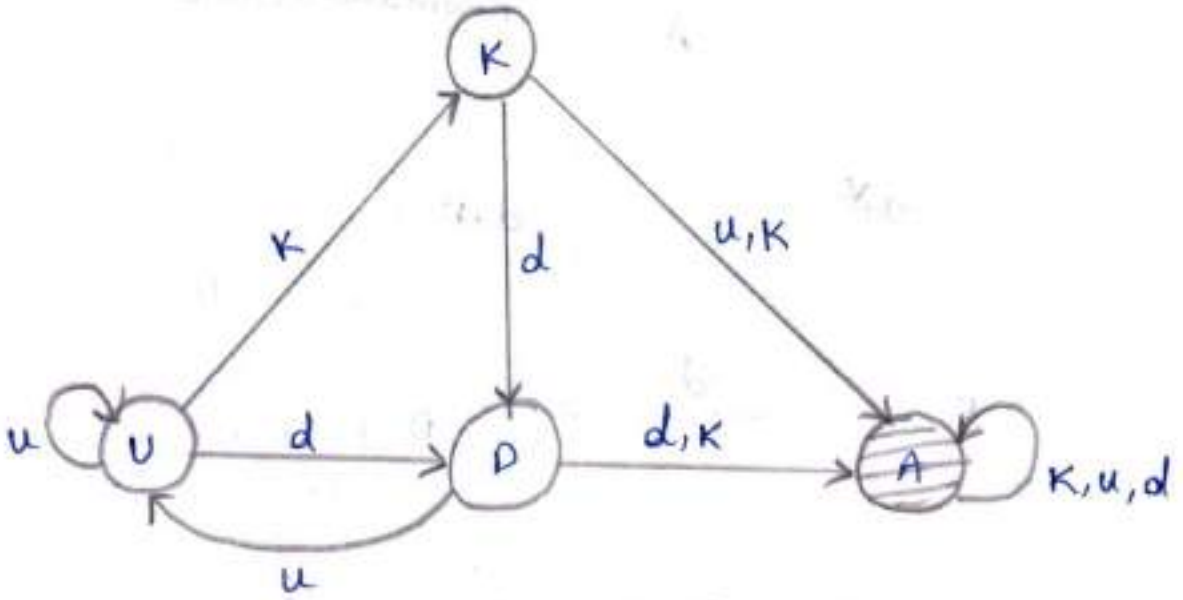


fig: Data flow Anomaly State graph

1.3.8 Showing Data flow Anomaly State graph by different Anomalies :-

A data flow Anomaly can be in one of four states.

S.NO	Symbol	State
1)	D	Defined state
2)	R	Referred or read or used
3)	U	Undefined killed
4)	A	Anomalous

The variables used in the data flow anomaly state graphs are,

D - defined variable

R - read or used

U - undefined or killed

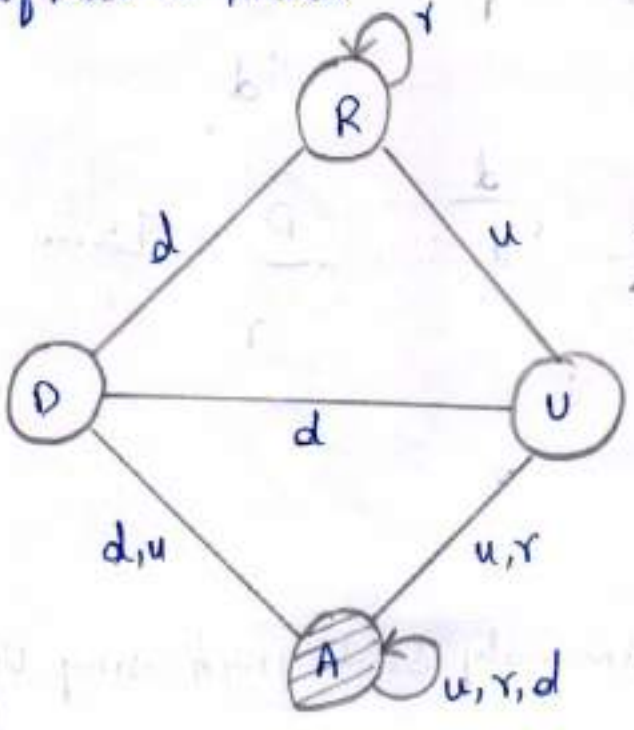


fig: Data flow Anomaly state graph 4 different states

The flow of the graph starts in D state. The defined variable d is killed and hence it goes in anomalous (A) state. An undefined variable u is being read (r) there (Above fig:). Hence it is anomalous A. The used variable x is redefined (d) in the state D and it is killed (u) in the state U.

The data flow anomaly state graph is also shown by the following different anomalies.

1) UR - anomaly
2) DU - anomaly
3) DD - anomaly

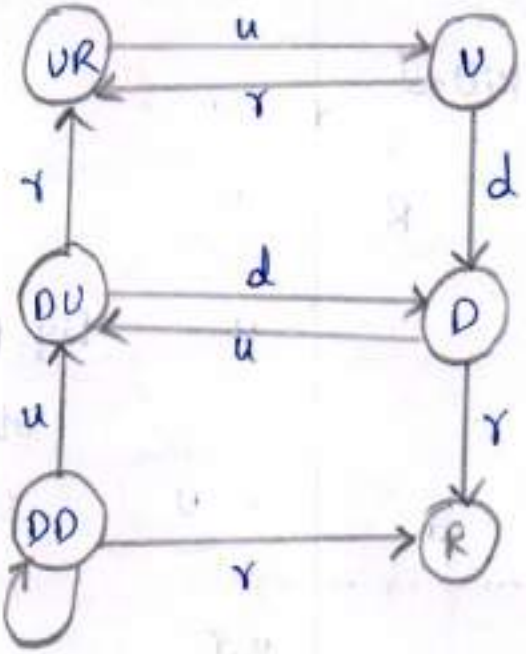


fig: Data flow Anomaly State graph using UR, DU, DD Anomalies

i) Assume that the variable starts in U state. i.e, it has been defined in D state and has been used in R state.

ii) If an undefined variable (u) is used or read, then it goes in state UR.

iii) If a defined variable (d) has been killed, then it goes in DU state.

iv) If a variable has been defined twice and has not been used or read, then it goes in DD state.

1.3.9 Static versus Dynamic Anomaly Detection:-

Static Anomaly Detection:- Static analysis is an analysis done at compile time. The source code is checked and the quality is improved by removing the bugs in the program. Syntax errors are detected in static analysis. To improve the quality of a document, the document is analyzed and checked by tool.

Dynamic Anomaly Detection:-

Dynamic analysis is done at run time. Dynamic analysis detects anomalous situations at run time with

Some of the data structures like arrays, pointers, records etc.

Arrays:- Arrays are dynamically calculated and to know whether the values are within the boundary range or out of boundary range.

Records:- In the case of records, files are created and the names of such files are dynamically known.

Subroutines:- Subroutines have names and data objects within them. without executing, the path of the call in a subroutine will not determine whether the call is correct or not.

1.4 Strategies in Data Flow Testing:-

1.4.1 Terminology:-

Different terminologies have been defined to fulfill the graphs in path testing. The terminologies are,

1) Definition-clear path segment
2) Loop-free path segment
3) Simple path segment
4) Du path

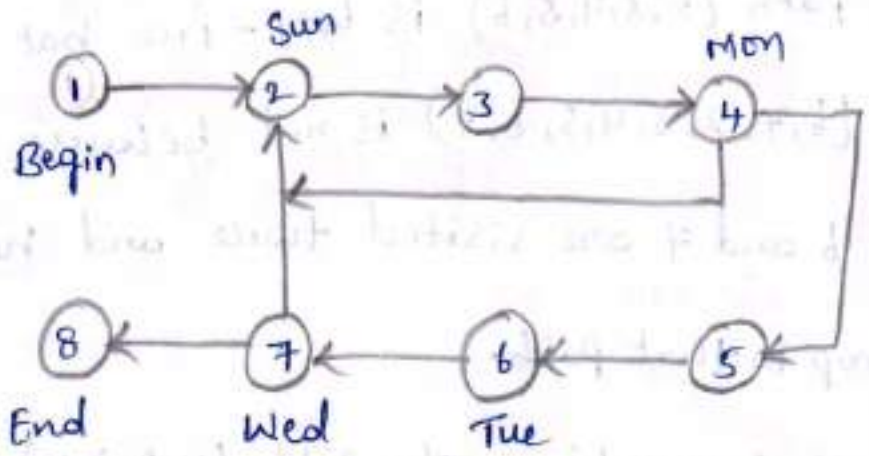


fig: Control flowgraph

1) Definition-clear path segment:- A path segment is a sequence of connected links between nodes. The first link of the path is defined and the subsequent link of that path is killed.

The path (1,2),(3,4),(5,6) are definition-clear. the path (7,2,3,4,5,6) is not definition-clear because the variable is defined on (2,3), (4,5) and again on (6,7). A definition-clear sub-path doesn't include loops.

2) Loop-free path segment:- The path which doesn't have any loop and every node of which is visited atmost once is known as a loop-free path segment.

In above fig:, variable A and B are used on the data flow link (1,3). Any test that starts from the link (1,3) satisfies this criterion (All du paths strategy) only for variables A and B but not for all the variables.

2) All Uses (AU) Strategy:- All-Uses strategy helps in reducing the number of test cases.

Definition:- All Uses (AU) Strategy is a data flow testing strategy that requires 'every definition from atleast one path segment to use every variable that can be reached by that definition'.

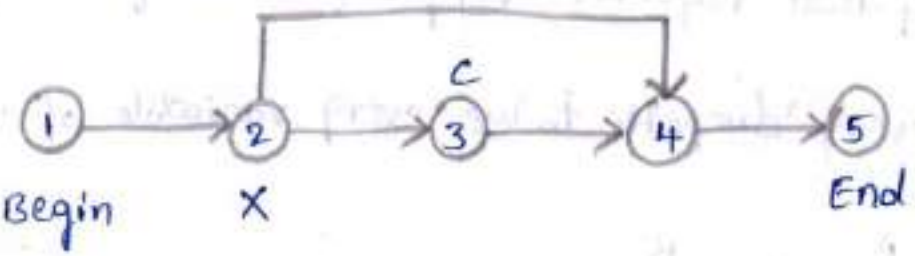


fig: Flowgraph Annotated for Variable x

If a variable x has a predicate use in ADUP strategy, the sub-paths (2,3,4) and (2,4) can be included in some test.

In AU strategy either (2,3,4) or (2,4) can be included, but we don't have to use both to begin a path. The link (2,3,4) can be included in some test cases because that's the only way to reach the c (computational) use in that link.

3) All p uses or Some-C-Uses (APU+C) Strategy:-

If a variable has a predicate use then there's no no need to select a computational use.

Definition:- APU+C is a strategy that requires 'every definition of every variable included atleast one path from that definition to every predicate use'.

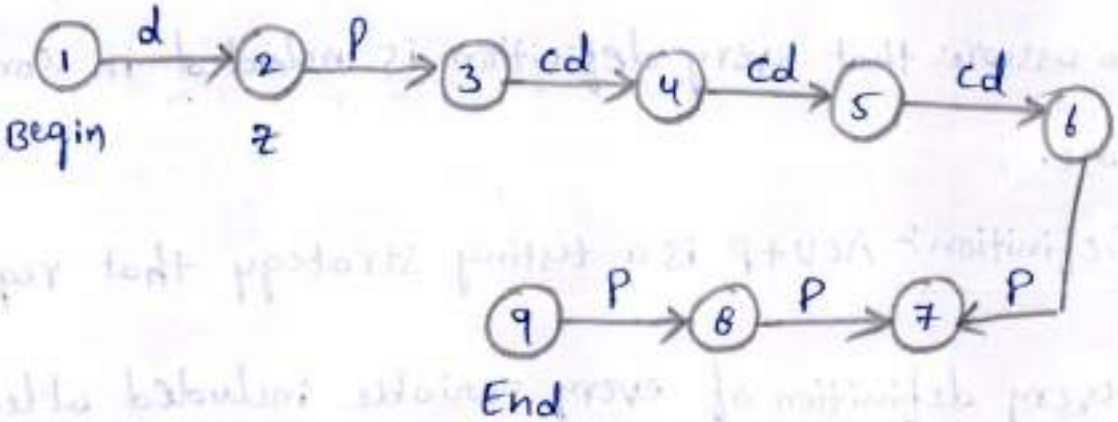


fig: flowgraph Annotated for variable z

In APU+C strategy, every definition of every variable has a path to every p-use of that definition. If there's no p-use of the definition, then a c-use is considered. In the above fig: links (1, 2), (3,4), (4,5), (5,6) must be included in test cases because they contain definitions for variable z. The links (2,3), (6,7), (8,9) must be included in test cases because they contain Predicate uses of z.

4) All c uses or some P uses (ACU+P) Strategy:-

In this testing strategy, first ensure that every definition has a computational use of that definition and if any definition is not covered, Predicate use cases are added to assure that every definition is included in some test case.

Definition:- ACU+P is a testing strategy that requires 'every definition of every variable included at least one path from that definition to every computational use'.

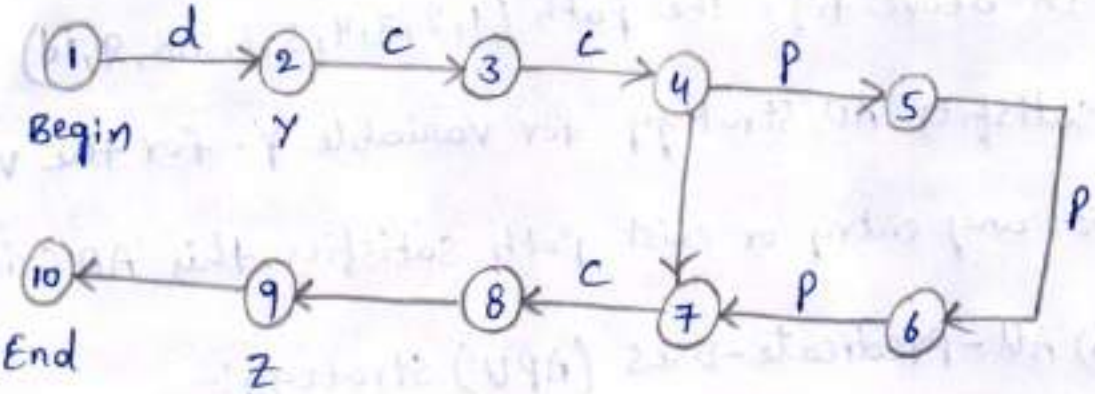


fig: flow graph Annotated for variable Y

In the above fig: ACU+P coverage is achieved for Y by path (1,2,3,4,5,6,7,8,9,10). But several definitions of predicate use are not covered, that is the path (4,5,6,7). In this testing strategy, every definition of every variable has a path to every C-use of that definition. And if there's no C-use of that definition, then a P-use of the definition is considered.

5) All-definitions (AD) Strategy:-

AD strategy is weaker than both ACU+P and APU+C strategies.

Definition:- AD is a testing strategy that requires 'every definition of every variable to cover atleast one use of that variable'. The use can be a computational use C or a Predicate use P.

In above fig: the path (1,2,3,4,5,6,7,8,9,10) satisfies AD strategy for variable Y. for the variable Z, any entry or exist path satisfies this AD criterion.

6) All-Predicate-Uses (APU) Strategy:-

In this testing strategy, 'every definition of every variable has a path to every p-use of that definition'. If there's no p-use in the definition, then it is dropped from contention.

Definition:- APU is a testing strategy that requires 'a p-use of definition if there are no c-use following that definition'.

7) All-computational-Uses (ACU) Strategy:-

Definition: ACU is a testing strategy that require 'a c-use of definition if there are not p-use following that definition'.

1.4.3 Ordering the Strategies:-

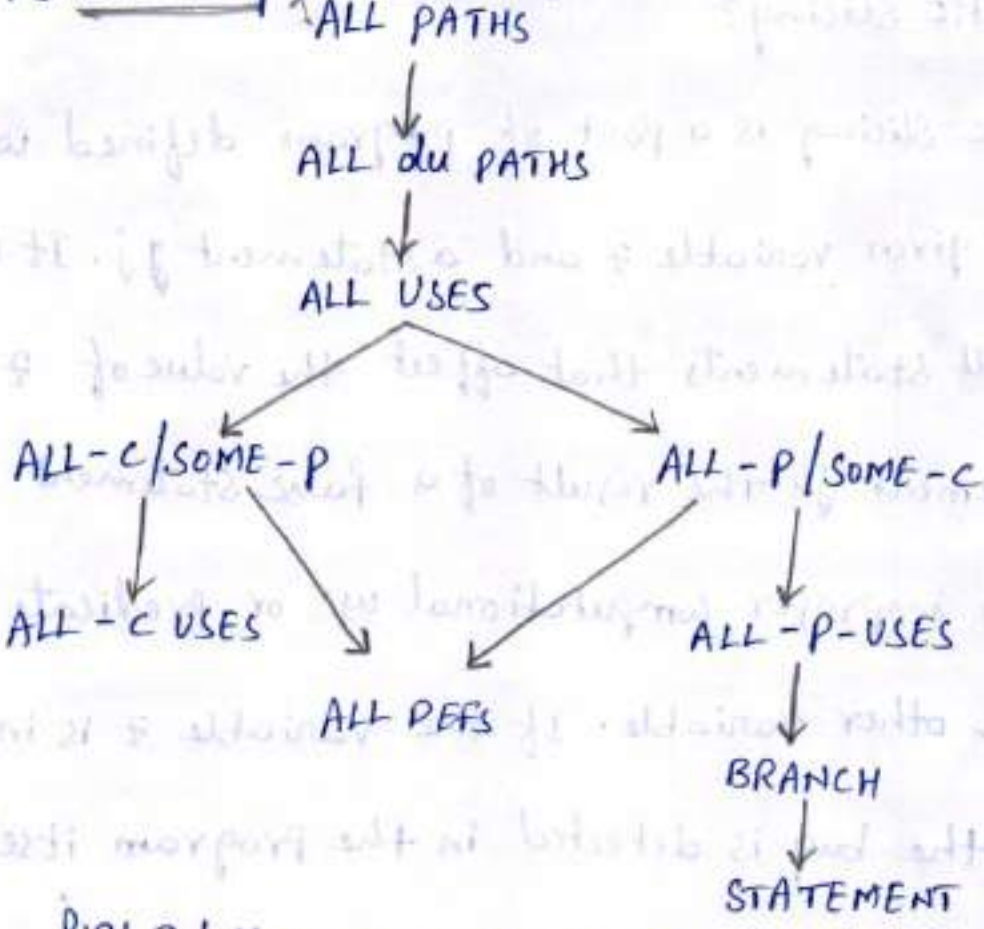


Fig: Relative Strength of Structural Test Strategies

1.4.4 Slicing, Dicing, Data flow and Debugging:-

Slicing is a program oridinally developed for conventional languages. It helps in understanding data flow and debugging techniques. the slicing is done based on variable sharing. There are two types of slicing technique which are as follows,

1) Static Slicing
2) Dynamic slicing

1) Static Slicing:-

Static Slicing is a part of program defined with respect to a given variable z and a statement j. It consists of all statements that affect the value of z at statement j. The result of a false statement effect in an improper computational use or predicate use of some other variable. If the variable z is incorrect, then the bug is detected in the program itself.

2) Dynamic Slicing:-

Dynamic Slicing is a refinement of Static Slicing. Dynamic Slicing compares the data flow relationship with respect to static data flows.

Slicing methods incorporate assumptions about bugs and programs. Due to the presence of bugs, the usage of the real program get reduced.

1.5 Applications of Data flow Testing:-

1) Data flow testing is used to detect the different abnormalities that may arise due to data flow anomalies.

2) Data flow testing shows the relationship between the data objects that represents data.

3) Data flow testing is cost effective.

4) Data flow testing uses practical applications rather that mathematical applications.

5) Data flow testing is used in developing web applications with java technology.

Domain Testing

2.1 Domains and paths :-

2.1.1 Domains :-

Domains can be specified directly and are predefined by their boundaries. A boundary is simply defined as an area of coverage in which decision on a program is predicated. A domain boundary is an area in which most of the domain bugs occur. A single domain may consist of one or more boundaries. Every boundary consists of one or more predicates in order to check the numbers which belong or do not belong to the domain.

2.1.2 Domain closure :-

The aim of the domain closure is to allow the wrong closure bugs to become as frequent domain bugs.

To understand the domain closure, let us take an example which consists of three cases for one-dimensional domain ~~in~~ as shown in the figure:

If the boundary points are related to other domain then the boundary is said to be an open boundary whereas if the boundary points are related to that domain it self then the boundary is said to be a closed boundary.

The cases for one-dimensional domain are as follows,

a) Domain is open on one side
b) Domain is closed on both sides.
c) Domain is open on both sides.

a) Domain is open on one side:-

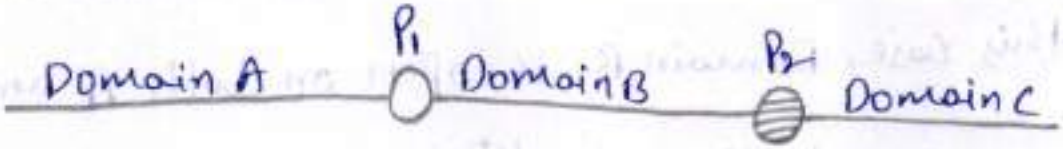


fig: one side open

Above figure there exists three domains namely Domain A, Domain B and Domain C. the two circles (An empty circle which denotes an open boundary P_1 and filled circle which denotes a closed boundary P_2)

Domain B is open on the point P_1 and is closed on the Point P_2.

b) Domain is closed on Both Sides :-

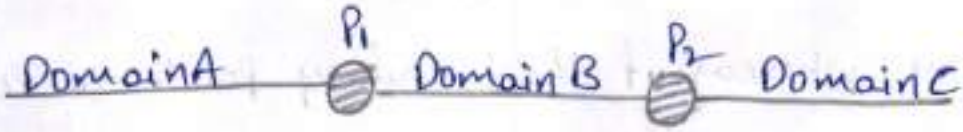


fig: Both sides closed.

Above figure: shows that the Domain B is closed on both sides i.e P_1 and P_2.

c) Domain is open on Both Sides :-

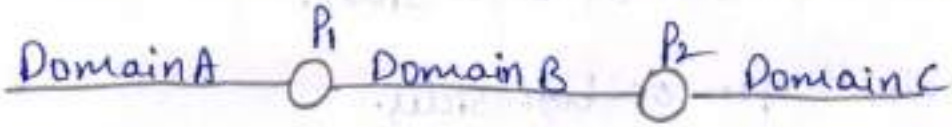


fig: Both sides open

this is the third case for one-dimensional domain. In this case, Domain B is open on both P_1 and P_2.

2.1.3 Domain Dimensionality :-

Depending on the input variables, the domains can be classified as number line domains, planar domains or solid domains i.e for one input variable the value of the domain is on the number line, for two variables the resultant domain is planar and for three variables the domain is solid.

2.1.4 The Bug Assumptions for Domain Testing:-

The bug assumption says that the definition of domain is incorrect whereas Processing of domain is correct.

If we have wrongly implemented the domains i.e the boundaries are taken incorrectly then the control-flow predicates are also inaccurate.

the following are some of the bugs that lead to domain errors.

a) Ambiguous Domains:- In this, the union of the specific domains results either in missing domains or holes. Hence, the union operation is incomplete. this bug will occur when the combination of required domains are not completed.

b) Overloaded Domains:- This bug will occur when the domains consist of too many conditions which result in a null domain.

c) closure Bug:- This bug happens when we have selected the wrong predicate. Such as $x>=0$ is written as $x<=0$.

d) Boundary Errors:- This bug occurs when the boundary is shifted or when the boundary is tilted or missed.

e) faulty logic:- this bug happens when there are incorrect manipulations, calculations or simplifications in a domain.

f) Contradictory Domains:- In this type of domain specification at least two assumed domains overlap. the contradiction between the two overlapped domains can be resolved by assigning 50% Probability of error to each of the overlapped domains.

g) Floating-Point zero checking:-

Any floating point number tends to zero only in the following cases,

i) when it is multiplied by zero

ii) when it is subtracted from itself.

iii) when its previous definition assigns a value '0' to it.

h) Double-zero Representation:-

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

2.1.5 Restrictions:-

Domain testing has restrictions, as do other testing techniques they aren't restrictions in the sense that you can't use domain testing if they're violated but in the sense that if you apply domain testing to such cases, tests are unlikely to be productive because they may not reveal bugs or because the number of tests required to satisfy the the criterion is greater than practicality warrants.

i) Simple Domain Boundaries and Compound Predicates:-

Boundary is simply defined as an area of coverage in which decision on a program is predicated. The two predicates used for defining a boundary are simple domain boundary

and compound predicate. Mostly, simple predicate is used to define each boundary of a program in order to avoid inconsistent boundary closures, where as compound predicates are used in which each part of the predicate specifies a separate boundary.

For example, a>=1 and a<=10 where 'a' is the compound predicate and the numbers 1 and 10 are the two simple domain boundaries.

ii) Linear Vector Space:-

A linear (boundary) predicate is defined by a linear inequality (after interpretation in terms of input variables) using only the simple relational operators >, >=, =, <=, <> and <.

For example, the predicate $x^2+y^2>a^2$ is not linear in rectangular coordinates, but by transforming to polar coordinates we obtain the equivalent linear predicate $r>a$.

Similarly, a polynomial boundary can be transformed into a linear vector space by $y_1 = x$, $y_2 = x^2$ $y_3 = x^3$, ...

2.2 Nice Domains and UGLY Domains:-

i) Nice Domains:-

There exists some properties related to domain boundaries. If we apply these properties then it makes domain testing very simple. If we do not apply these properties then it makes domain testing very difficult.

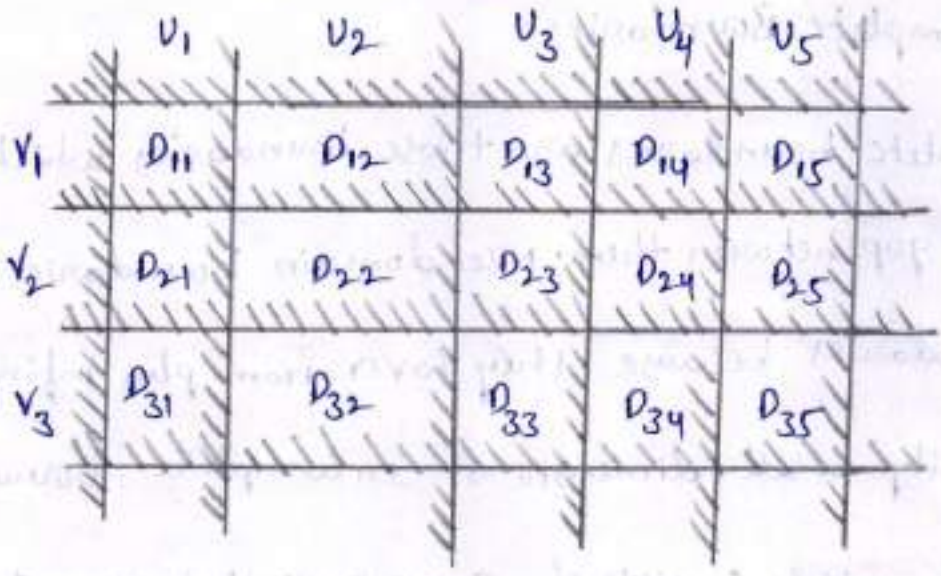


fig1: Nice Two-dimensional Domains

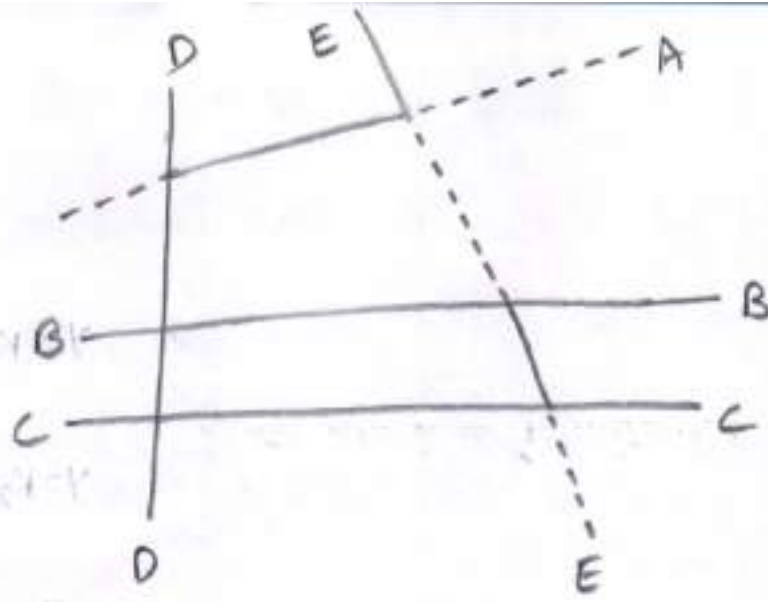
The following are some of the properties that are associated with domain boundaries.

a) Linear and Non-linear Boundaries:-

Nice domain boundaries can be described by using linear inequalities or equations. The effect on testing comes from the reality that if it considers only two points then it specifies a straight line. If it consider three points, then it represents a plane. Linear boundaries are more frequently used than the non-linear boundaries.

b) Complete Boundaries:-

Complete boundaries are those boundaries which do not have any gap between them. Nice domain boundaries have complete boundaries because they cover from plus infinity to minus infinity in all dimensions. Incomplete boundaries are those which consists of some gaps between its two boundaries and which are not covered in all dimensions.



Boundaries A and E have gaps.

Fig! Incomplete Domain Boundaries

c) Orthogonal Boundaries:-

The U and V boundary sets in fig1 are orthogonal, that is every individuality in V is perpendicular to every inequality in U. If two boundary sets are orthogonal, then they can be tested independently. In fig1: we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6+4 = 10 (O(n)).

d) Consistent closure:- Consistent closures are the most simple and fundamental closures that gives a systematic results after performing the required analysis.

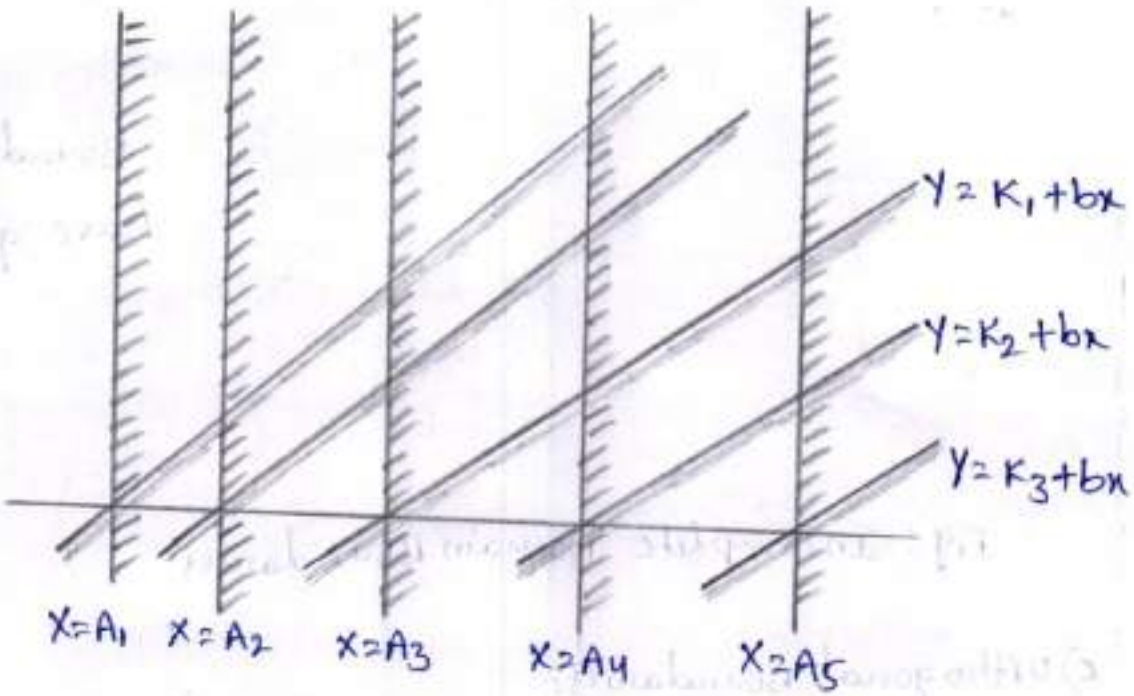


fig: Linear, Nonorthogonal Domain Boundaries

Above figure the shading lines on the boundary shows that the domain consists of the shading lines.

Non-obthogonal domain boundaries, which means that every inequality in domain x is not perpendicular to every inequality in domain y.

e) Systematic Boundaries:- Systematic boundaries refer to boundary inequalities associated with simple mathematical functions such as a constant.

Example: Consider the following relations,

$f_1(Y) >= C_1$ (or) $f_1(Y) >= G(1,K)$

$f_2(Y) >= C_2$ $f_2(Y) >= G(2,K)$

$f_3(Y) >= C_3$ $f_3(Y) >= G(3,K)$

. . . .

$f_i(Y) >= C_i$ $f_i(Y) >= G(i,K)$, where

f_i = An arbitrary linear function

Y = An input vector

$C_{i,K}$ = Constants.

$G(i,K)$ = A decent function over i and K that produces some constant.

ii) Ugly Domains :-

Ugly domains are the domains which can be simplified by Programmers and testers. These simplifications can be either good or bad depending upon the specifications. But when the domain's complexity is important and the Programmers simplification is not safe, it will result in bugs.

The following are some of the simplifications of ugly domains.

i) Non-Linear Boundaries:-

these boundaries may or may not be present in simple Programming because these boundaries do not provide any information regarding simplification of boundaries even if they are important.

ii) Ambiguity and Contradictions:-

Programs can't be ambiguous or contradictory where as specifications ~~tab~~ can be.

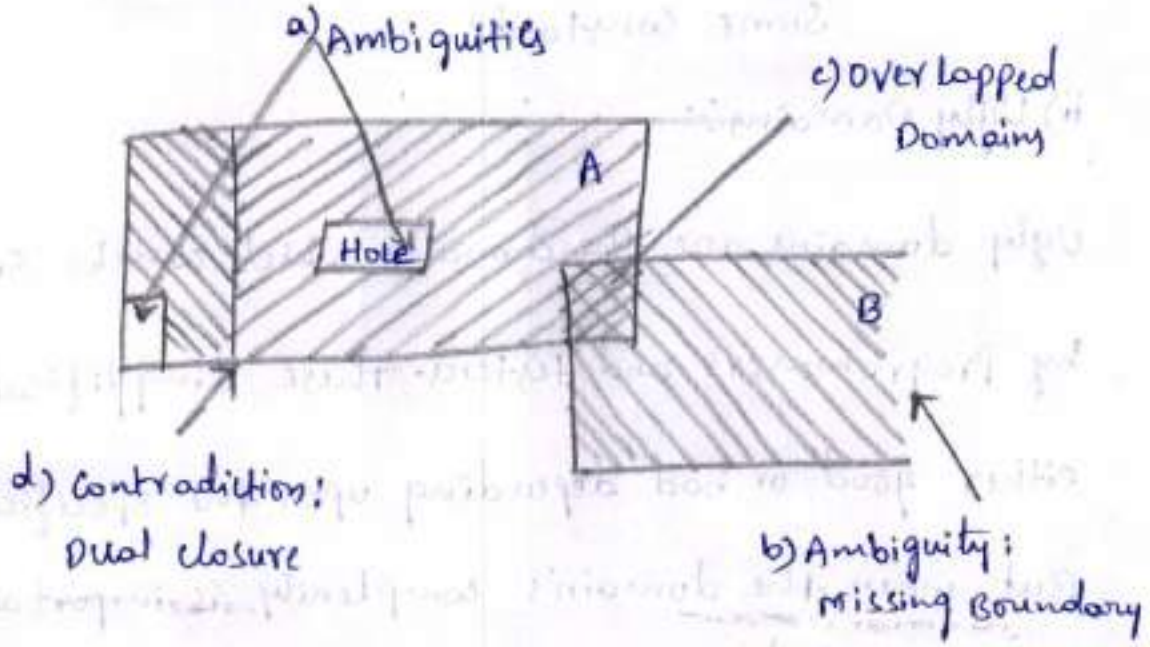


fig: Domain Ambiguities and contradictions.

Domain ambiguities lead to holes or cracks in the input space as shown in above figure. A hole in the one-variable input space can be easily viewed whereas ambiguity in two variables input space is difficult to identify. Furthermore, ambiguities in three or more variable input space are almost impossible to identify without applying appropriate tools.

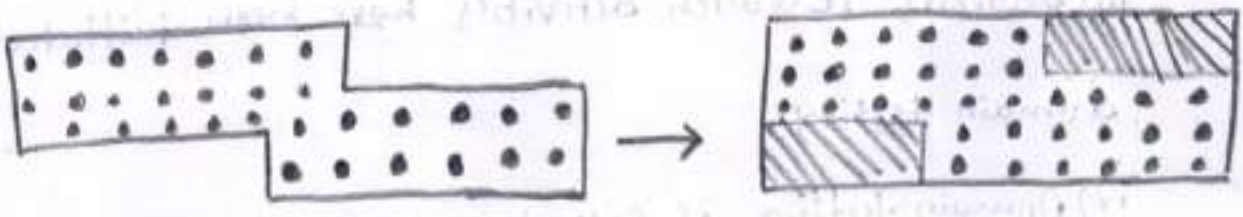
Two types of contradictions exists, they are as follows,

a) Overlapped domain specifications.

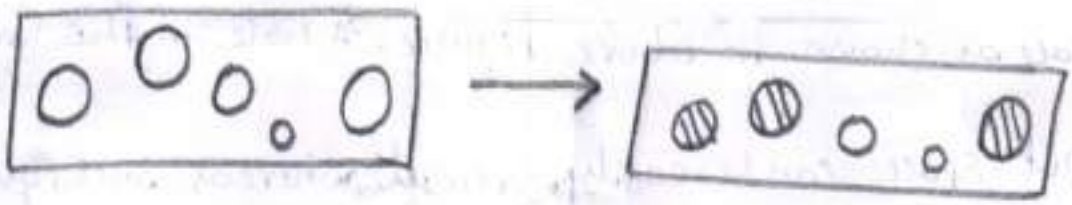
b) overlapped closure specifications.

iii) Simplification of Topology:-

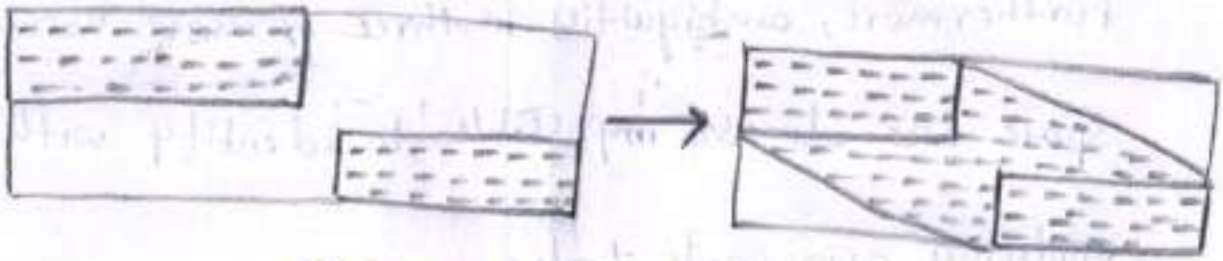
Once the programmers find out the bugs from domain topology then the testers try to specify the domains topology by using the following three cases.



a) Making it convex



b) Filling in the holes



c) Joining the Pieces.

figure: Simplifying the Topology.

2.3 Domain Testing:-

The following are the applications of domain testing:

i) Domain testing is used to detect domain errors or to provide assurance of the correctness of Path domain.

ii) Domain testing is also used to detect linear errors in a non linear predicate.

iii) Various research activities have been initiated in domain testing.

iv) Domain testing is easy to use for one dimension, but it is difficult for two and more than two dimensions.

2.3.1 Interior Point, Boundary Point and Extreme Point:-

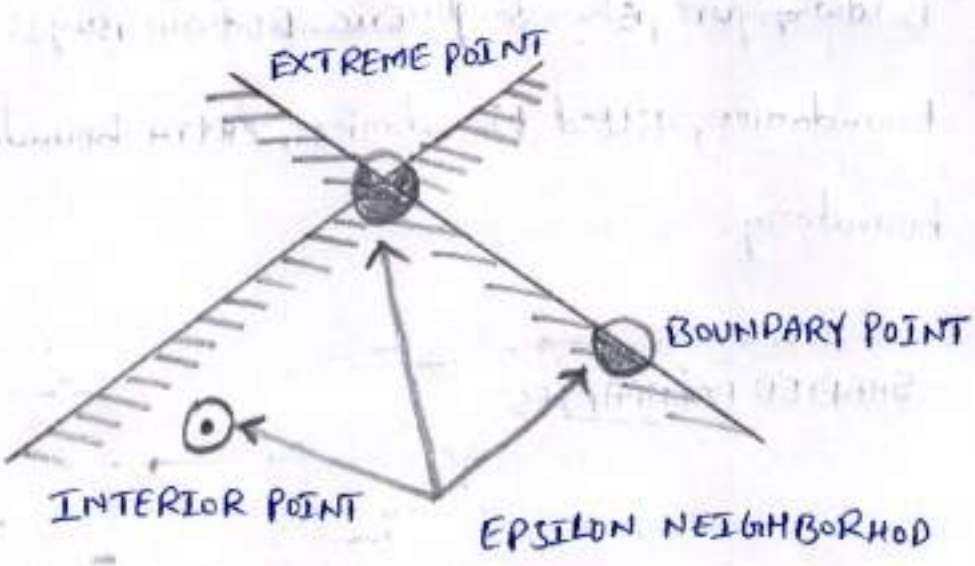


Fig: Interior, Boundary and Extreme Points

a) Interior Point:- An interior point is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.

b) Boundary Point:- A boundary point is one such that within an epsilon neighborhood there are points both in the domain and not in the domain.

c) Extreme Point:- An extreme point is a point that does not lie between any two other arbitrary but distrinct points of a domain.

~~3.3.2 Testing One-Dimensional Domains:~~

Below figure shows generic Domain Bugs: Shifted boundaries, tilted boundaries, extra boundary, missing boundary.

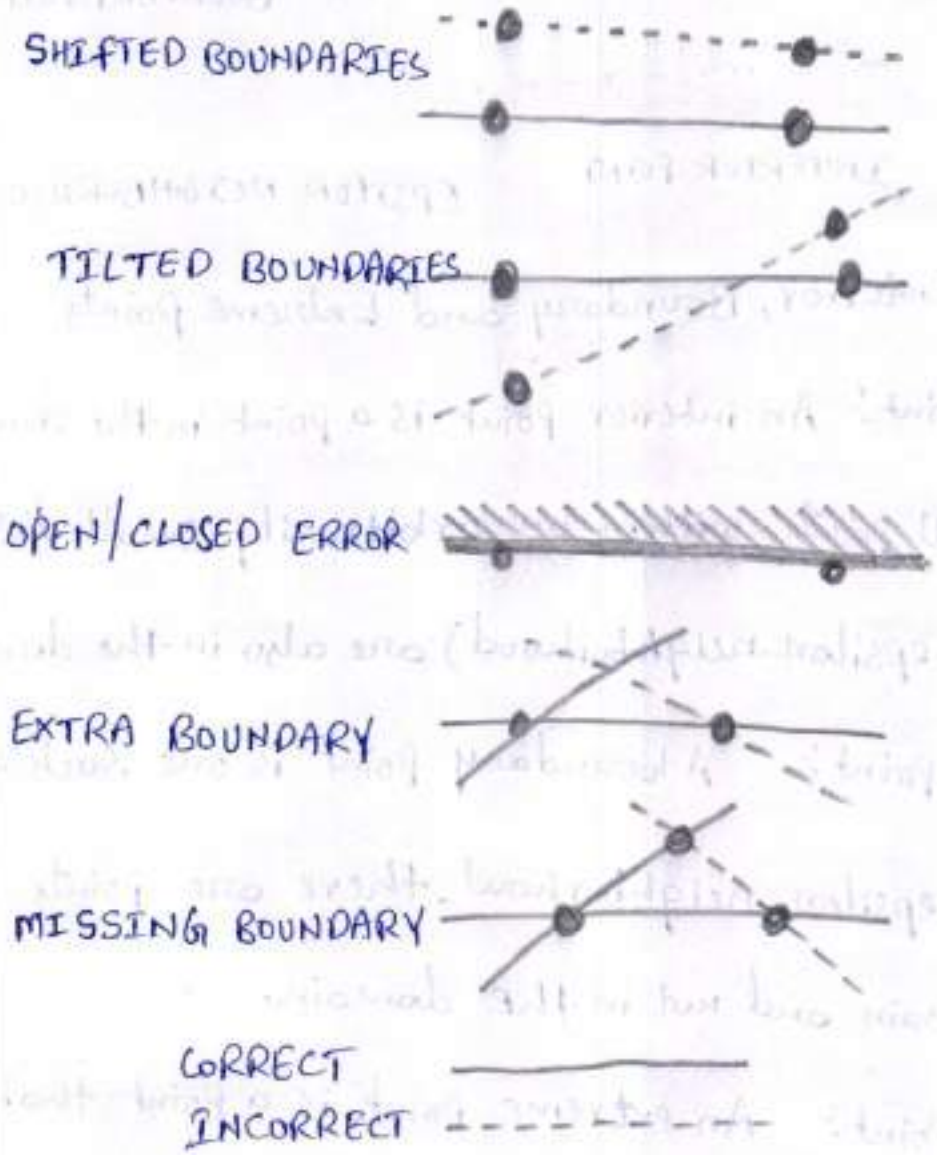
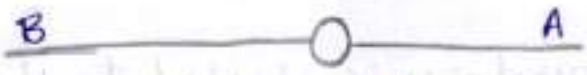


Fig: Generic Domain Bugs

2.3.2 Testing One-Dimensional Domains:-

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



a) An open domain (A)

i) One-Dimensional open Domain Boundary :-

a) closure Bug:- This bug happens when we assign a wrong domain. Let us first consider that our boundary in open for x. If we assign a wrong domain, then this error is known as closure bug and it is shown in the below figure:

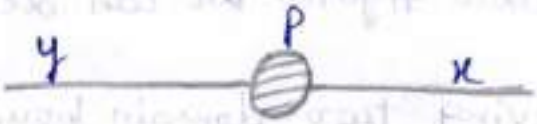


fig : closure Bug

∵ where P is the point/boundary.

Above figure, a test point P on the boundary identifies this bug allowing the processing at domain x rather than at domain y.

b) Boundary Shifting:- This bug happens when a boundary is shifted to either left side or right side as shown in the following figure.

A test point 'P' identifies the shift (either to left or right) in the boundary.

fig: Boundary is shifted to the left side

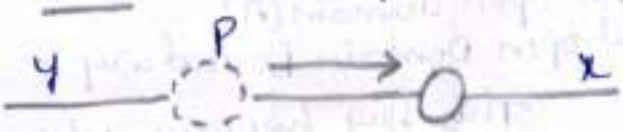


fig: Boundary is shifted to Right side.

c) Extra Boundary :- This bug occurs when an extra Point (boundary) is added, for instance 'z' is added as shown in the below figure. We can detect this extra boundary by observing two domain boundaries.

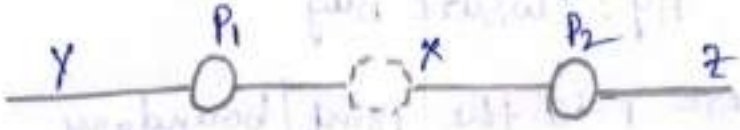


fig: Extra Boundary.

In the above figure an extra boundary i.e z is added and also we can see that x is divided into two parts. So extra boundary can be detected by the two off Points i.e x and y.

d) Missing Boundary:-

This bug occurs when any one of the domain's boundaries is missing is shown in the below figure.

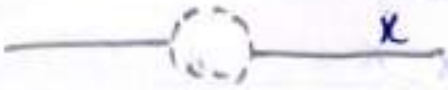


figure: Boundary missing

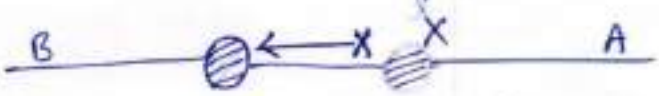
In the above figure we can see that the boundary 'y' is missing. This bug can be detected by using open off inside point. As we know that if the boundary is open then the off point is in 'x' i.e whatever is intended to happen in boundary x, now occurs in boundary 'y'.

ii) One-dimensional closed Domain Boundary:-

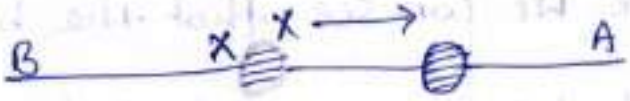
The bugs which we have seen in the open domain boundary remains same as in closed domain boundary except that, the point or boundary of a closed domain can be represented as filled circles, and it requires a strategy known as closed off outside.

a) A closed Domain (A)

b) closure Bug

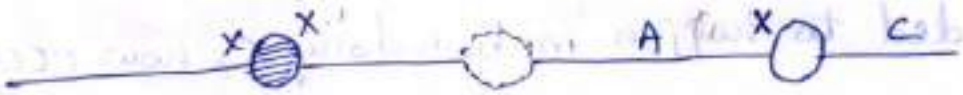


c) Boundary shifted left



d) Boundary shifted Right

e) Missing Boundary



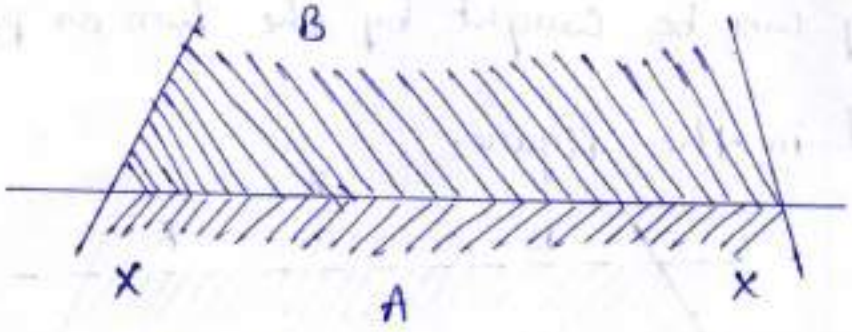
f) Extra Boundary

figure: one-dimensional Domain Bugs, closed Boundaries.

2.3.3 Testing Two dimensional Domains:-

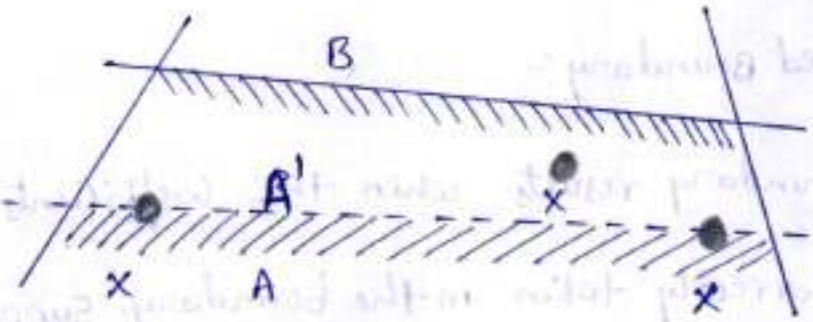
i) closure Bug:- The below figure depicts a wrong closure, because it uses some wrong or incorrect operators. for instance, if the original operator was $a>b$, but a wrong operator $>=$ ($a>=b$), is used instead, then it

results in closure bug the two off points are as shown in the below figure, can catch this bug.



a) closure Bug

ii) Shifted Boundary :-



b) Shifted up

In the above figure we can see that the boundary is shifted up, which transfers some portion of the domain B into A, and this shifted up boundary is denoted by A'. All this happens because of wrong constant in a predicate. for instance, if the boundary a+b<=4 is used instead of a+b <=40 then it results in shifted up boundary.

ii) Shifted Down Boundary:-

this case is reverse of shifted up boundary. But the bug can be caught by the two on points as depicted in the figure.

c) Shifted Down Boundary.

iii) Tilted Boundary:-

This boundary results when the coefficients of inequality are incorrectly taken in the boundary. Suppose if the actual boundary is $2a+4b>13$ and if $4a+2b>13$ is used instead, then it results in tilted boundary which is shown in the below figure:

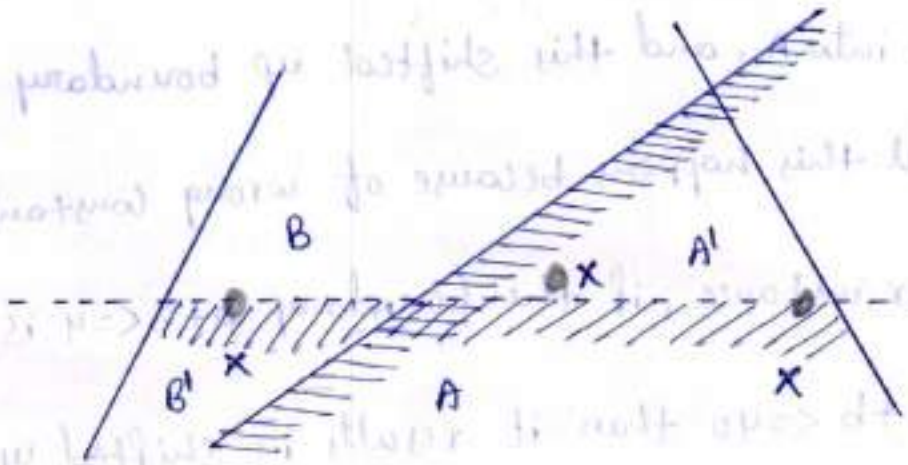


fig: Tilted Boundary

iv) Extra Boundary:-

An extra predicate is added to build an extra boundary. This extra boundary passes through several domains and leads to many test failures for the same bug.

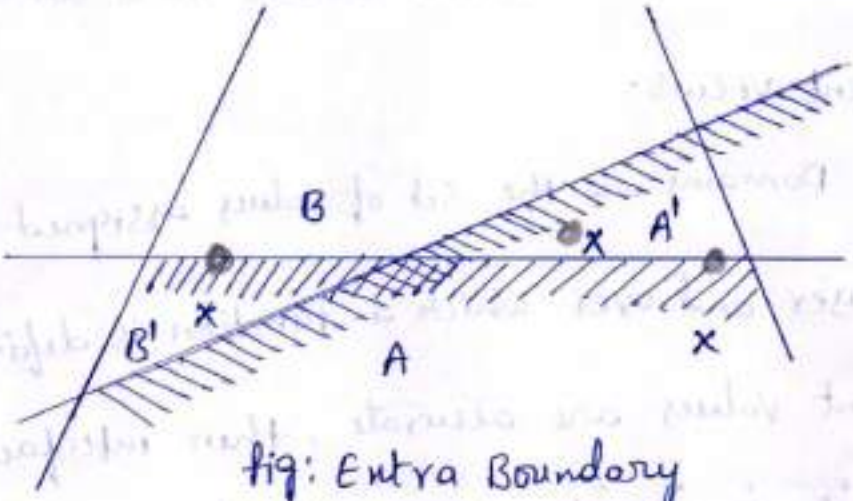


fig: Extra Boundary

v) Missing Boundary:-

This is also one of the possible cases for closed boundaries in which a boundary is generated by leaving out the predicate. This boundary combines various boundary domains and may effect large number of test failures even though there is only one bug.

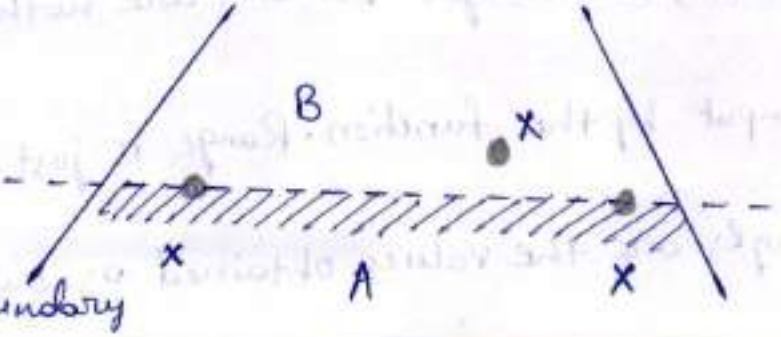


fig: Missing Boundary

2.4 Domains and Interface Testing:-

Interface testing can be defined as a method of testing in which a call to the subroutine is made to detect the presence of any errors. It is also used to determine the correct values in a call by identifying different views.

Domains are the set of values assigned to the variables by a user and over which a function is defined. If the resultant values are accurate, then interface testing must check the domains.

Some of the applications of domain testing for interface testing are as follows,

1) Domains and range
2) Closure compatibility
3) Span compatibility
4) Interface range and domain compatibility.

1) Domains and Range:- Domains are nothing but the values taken as input by the function. Range is just the opposite of domains. Ranges are the values obtained as output from the

function. In most testing techniques, more focus is on the input values.

2) closure Compatibility :-

It is used to determine the closure affinity between the domain and range. Suppose that the domain of the called and caller subroutines are equal. Let their domains between the numbers 0 to 17. The closure compatibility can be determined by taking two types of number lines. They are as follows,

a) Thick number line (closed line)

b) Thin number line (open line)

If the number line is thick, then the line is said to be closed, otherwise it is said to be open.

These two lines can be represented in four possible ways.

i) Thick lines on top and bottom.

ii) Thin line on top and thick line at bottom.

iii) Thick line on top and thin line at the bottom.

iv) Thin lines on top and bottom.

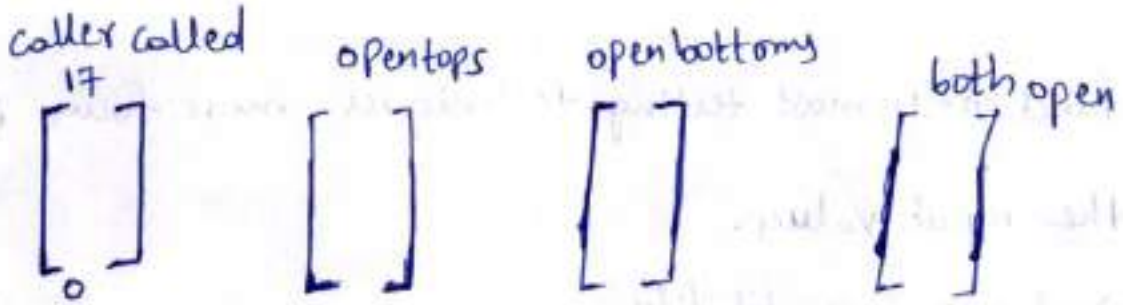


Figure: Range/Domain closure compatibility

3) Span compatibility :-

The span compatibility can be represented in twelve different ways. The diagrammatic representation is given below figure:

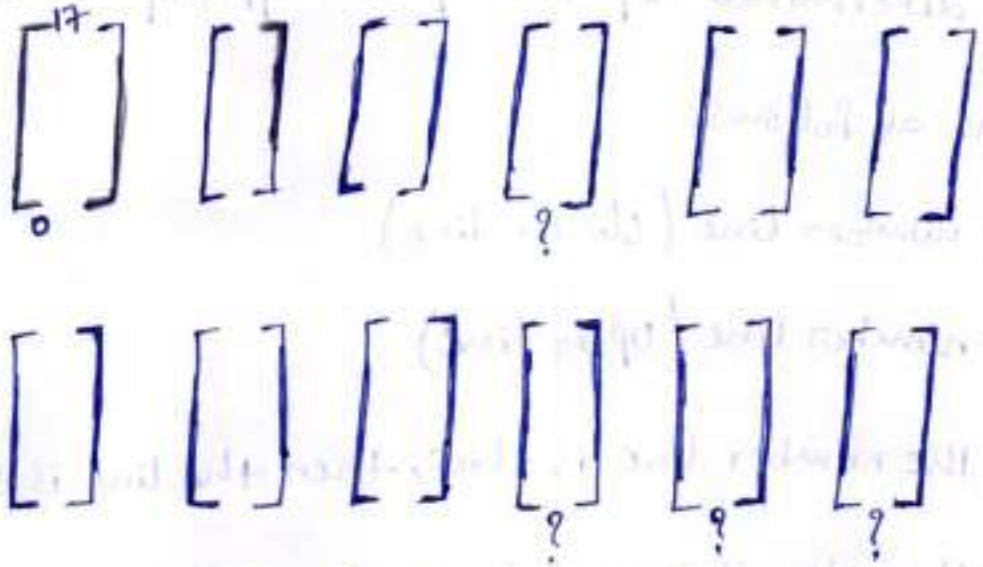


Fig: Equal-span Range/Domain Compatibility Bugs

In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug.

The span incompatibility of caller and called routines are represented in three possible ways as shown in the below figure: The range of caller is a subset of called routines domain which is not necessarily a bug.

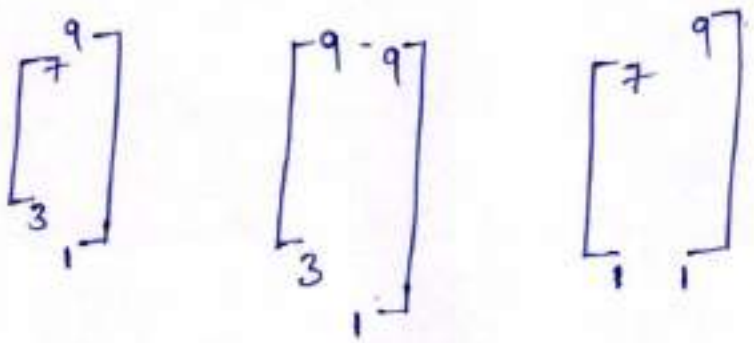


fig: Harmless Range/Domain Span compatibility Bug.
(caller span is smaller than called)

4) Interface Range and Domain Compatibility:-

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

on every input variable, independent of every other input variable to ensure that the domain of the caller routine is compatible with that of the called routines domain.