UNIT - II

Transaction - Flow Testing

→ The control flow graph was introduced as a structural Model. Now use the same conceptual components and Methods over a different kind of flowgraph, the transaction flow graph.

→ Transaction flow graph is a model of the structure of the system's behavior.

Transaction FLOWS

→ A Transaction is a unit of work seen from a system user's point of view. A Transaction consists of a sequence of operations.

→ Some of which are performed by a system, persons, or devices that are out side of the system.

→ A Transaction for an online information retrieval system might consists of the tasks.

1) Accept input (tentative birth)
2) Validate input (birth)
3) Transmit acknowledgement to requester
4) Do input processing
5) search file
6) Request directory from user
7) Accept input.

8) Validate Input
9) Process request
10) Update file
11) Transmit output
12) Record transaction in log and cleanup (death)

→ The users sees this scenario as a single transaction.

Ex: In ATM (Automatic Tellar Machine) several operations takes place such as, With drawing Money, Depositing, Transferring Money, Paying bills etc.

→ The order of operations in any two transactions may differ but they have same 12 operations.

→ For a transaction to be completed and database changes to be made permanent, a transaction has to be completed in entirely.

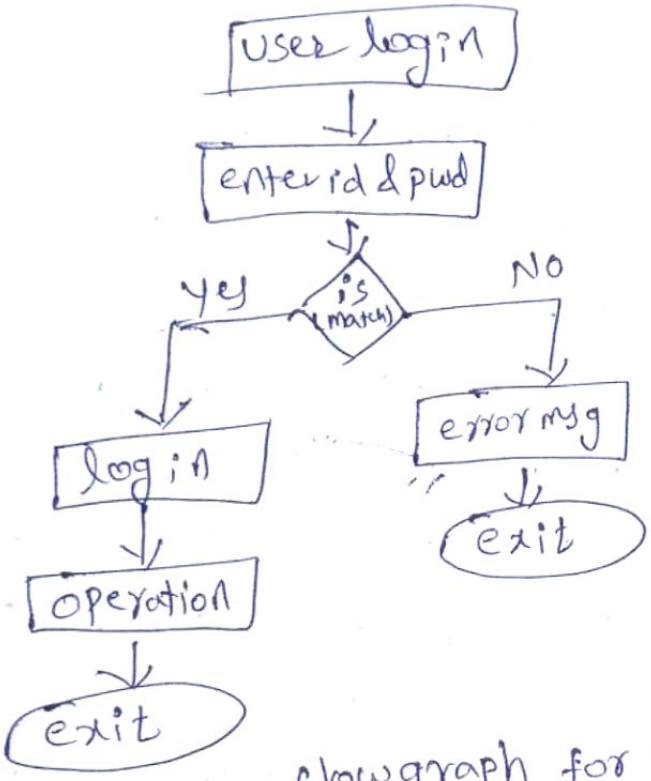
→ If something happens before the transaction is successfully completed, any changes to the database must be kept track so that they can be undone.

Transaction flow Testing:

→ The process of testing & designing test cases for each and every component of the transaction flow graph is known as transaction flow testing.

(2)

eg:



flowgraph for process of user login

Usage:

→ Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.

→ A big system such as an air traffic control or airline reservations system, has not hundreds, but thousands of different transaction flows.

→ The flows are represented by relatively simple flowgraphs, many of which have a single straight through path.

→ Loop are infrequent compared to control flow-graphs.

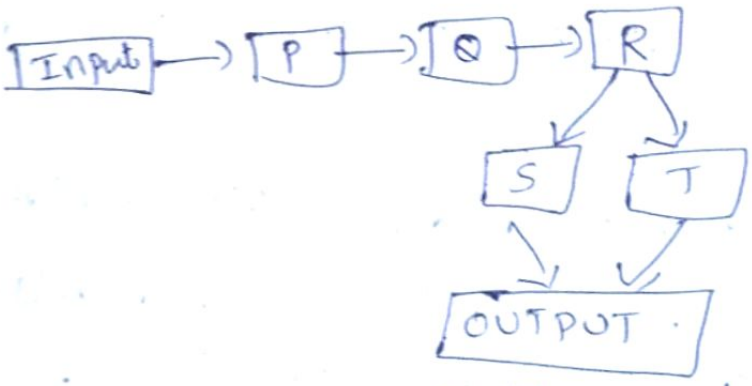
Implementations:

→ There is no direct relation b/n the 'process' and 'decision' of the transcation flow and the respective components of the program.

→ The implementation will be carried out indirectly in control structure of the system.

→ The each transcation is considered as a token. Token is similar to transcation Control Block (TCB).

→ TCB passes from one routine to another routine in its flow. so, in a single sentence a transcation flow can be defined as 'pictorial representation of the flow of tokens.



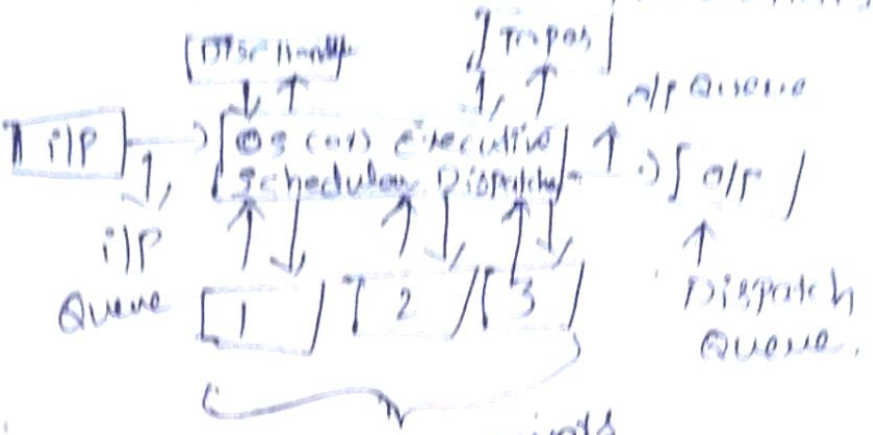
Transcation of Flow graph

→ It passes through the i/p processing and then through process P and then followed by Q, R.

→ The result of process R goes to either process S or process T and finally the o/p is obtained.

→ To process all these tokens or to implement a transcation flow, a generalized system that can process the

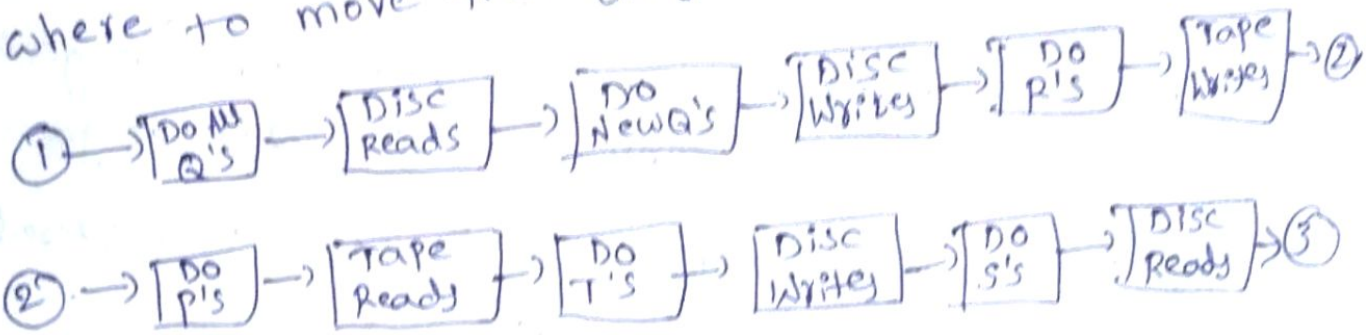
above transaction and also other transactions is required.



Transaction flow graph

→ The control of the entire system is done by OS. All the boxes are processes and all links are process queues.

→ The scheduler maintains a table with details of transactions and with that information it knows where to move the transactions next.



Executive / Dispatcher Flowchart

→ The processing order of transaction flow is P, Q, R, S, T processes are invoked regardless of other order.

→ In multiprocessing systems, there is no direct correspondence b/n the processes and transaction flows.

(5)

→ A transaction will however, receive attention from the processing Modules in the static order.

→ But there could be many other things going on b/n the instances of these transactions.

→ The scheduler invokes module P, which cleans up all the tasks. The task reads are performed and the scheduler again invokes Module P for processing any additional work that has been assigned to it.

→ Transaction flows has a cyclic structure and is common for both in process control and communication systems.

→ The processing Modules of this structure should be invoked in fixed order.

Complications

→ In simple cases transactions have a unique identity from the time they are created to the time they are completed, in many systems a transaction can give birth to others and transactions can also merge.

→ In simple flow graph is inadequate to represent transaction flows that split and merge.

1) Births:

→ The decision symbol can be interpreted in three different possible interpretations.

(i) Decision

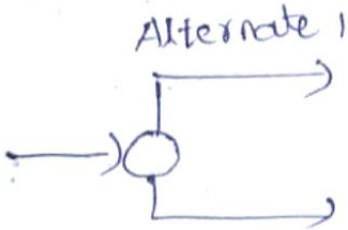
(ii) Biosis

(iii) Mitosis

(i) Decision:

→ Decision node as in a control flow graph. Interpreted in terms of transcation flow graph, this symbol means that the transcation will either take one alternative or the other, but not both.

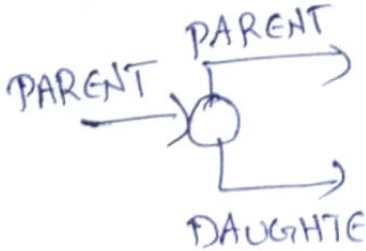
→ This as a decision point of a transaction flow.



Decision

(ii) Biosis:

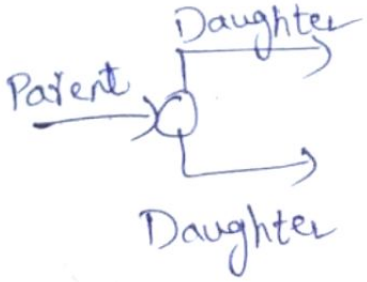
→ The incoming transaction (the parent) gives birth to a new transaction (the daughter), hence both transactions continue on their separate paths, the parent retaining its identity as a transcation. this situation is called as a Biosis.



Biosis

(iii) Mitosis

→ Mitosis is similar to the Biosis. except that the parent transaction is destroyed and two new transactions (daughters) are created. this is called Mitosis.



Mitosis

2) Merges :

→ Transaction-flow Junction points are potentially as troublesome as transaction-flow splits.

→ The ordinary junction, which is similar to the junction in a control flow graph.

→ It is understood that a transaction can arrive either on one link (path 1) or the other (path 2).

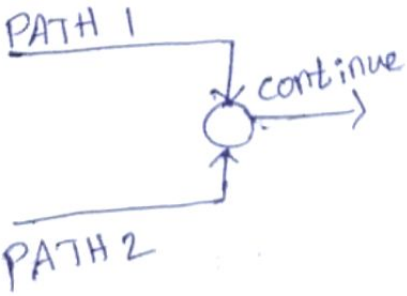
→ Merges are divided in to 3 types.

(i) Junction

(ii) Absorption

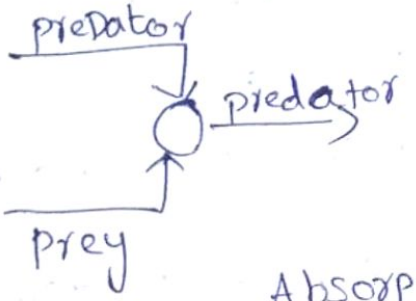
(iii) Conjugation.

i) Junction:



→ The ordinary Junction which is similar to the Junction in a control flow graph.

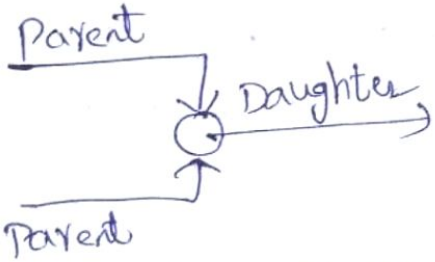
(ii) Absorption:



Absorption

→ A predator transaction absorbs prey. The prey is gone but the predator retains its identity.

(iii) Conjugation:



Two parents transactions merge to form a new daughter. In keeping with the biological flavor of this section. this act conjugation.

→ Models for testing are intended to give us insights into effective test case design.

→ We have no problem with ordinary decisions and functions. prescription for the troublesome cases.

(a) Biosis:

→ Follow the parent flow from begining to end. Treat each daughter birth as a new flow, either to the end or to the point where the daughter is absorbed.

(b) Mitosis : The beginning of the Parent's flow to the mitosis point and one additional flow for each daughter, from the mitosis point to each's respective end.

(c) Absorption : follow the predator as the primary flow. They prey is modeled from its beginning to the point.

(d) Conjugation : Three are more separate flows the opposite of mitosis. from the birth of each parent proceed to the conjugation point and follow the resulting daughter from the conjugation point to her end.

Transaction - Flow Structure

→ Transaction flows look like control flows, it does not follow that what constitutes good structure for code constitutes good structure for transaction flows.

Reasons :

1) It's a model of a process, not just code. Humans may be involved in loops, decisions, and so on.

2) Parts of the flows many incorporate the behaviour of other systems over which we have no control.

3) No small part of the totality of transaction flows exits to model error conditions, failures, malfunctions, and subsequent recovery actions.

4) The number of transactions and the complexity of individual transaction flows grow over time as features are added and enhanced.

5) systems are built out of modules and the transaction flows result from the interaction of those modules.

6) Interrupts, priorities, Multitasking, Multicomputers, synchronization, parallel processing, queue disciplines, polling.

Inspections, Reviews, Walkthroughs

→ Transaction flows are a natural agenda for system reviews or inspections.

→ Transaction flow walk throughs at the preliminary design review and continue them in ever greater detail as the project progresses.

1) In conducting the walk throughs, you should:

(a) Discuss enough transaction types to account for 98%-99% of the transactions the system is expected to process.

(b) Discuss paths through flows in functional rather than technical terms.

(c) Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.

2) Make transaction-flow testing the cornerstone of system functional testing just as path testing is the cornerstone of unit testing.

3) Select additional transaction flows paths for loops, extreme values, and domain boundaries.

4) Select additional paths for weird cases and very long potentially trouble some transactions with high risks and potential consequential damage.

5) Design more test cases to validate all births & deaths and to search for lost daughters.

6) Publish & distribute the selected set of test paths through the transaction flows constitute an adequate System functional test.

7) Have the buyer concur that the selected set of test paths through the transaction flow constitute an adequate system functional test.

8) Tell the designers which paths will be used for testing but not the details of the test cases that force those paths.

Path Selection

-> The process of path selection is completely different from that of control flow graphs.

Step 1: start the process by covering the sets of tests of (C1+C2). This is analogus to that you used in path testing. It is not expected to find many bugs on the path covered.

Step 2: As done in control flow graph, select a covering set of paths based on functionality sensible transaction.

Step 3: After designing the tests for those paths with the help of a designer, do exactly opposite to that what is done for unit tests.

Step 4: create a catlog with all the paths discovered in step 3.

Step 5: Go over them again and again repeat the process. Its detecting the

- Missing interlocks
- Duplicate interlocks
- Interface problems
- Duplicate processing.

-> All these stuff without the transaction flow could have been found during acceptance tests.

13

Advantages:

1) Transaction-flow uses structural path testing for selecting a path.
2) Many bugs don't occur in this path selection.
3) The cost is paid only on some well-known transactions.
4) The path selection is effective.

Disadvantages:

1) The path selected is longest path
2) The code correction is expensive.
3) The wired paths make designers irritate.

Sensitization

-> Most of the normal paths are very easy to sensitize 80% - 95% transaction flow coverage (C1+C2) is usually easy to achieve.

1) Use patches: It is easier to throw an error return from another system by a legal patch. If we don't put a patch into our system the other one's will have to put the patch into their system.
2) Misture: This is an act to test the system of inadequate resources so that most of the resources related exceptions are handled.

3) Break the rules : System data base generator is used to create the objects and guarantee that they are correctly specified.

4) Use Break points : The place where the path is hard to sensitize, put break point at the starting position and the patch the transaction control block to force that path.

Instrumentation

→ Instrumentation plays a bigger role in transaction-flow testing than in unit testing path.

→ counters are not useful because the same Module could appear in many different flows and the system could be simultaneously processing different transactions.

→ All the processing steps for the transaction queues and the entries and exits are needed to be traced.

→ In few systems, the operating system it self it will provide such type of tracing.

Test data base

→ 30% - 40% of the effort of transaction-flow test design is the design and maintenance of the test data-bases.

Implementation comments

1) Transaction Based systems :

(a) **Transaction control Block :**

→ An explicit transaction control block associated with every live transaction.

→ The block contains, among their things, the transaction type, identity and processing state.

(b) **Centralized, common, processing Queues :**

→ Transactions are not passed directly from one process to another but are transferred from process to process by means of centralized explicit processing queues.

(c) **Transaction Dispatcher :** There is a centralized transaction dispatcher. Based on the transactions current state & transaction type, the next process is determined from stored dispatch tables or a similar mechanism.

(d) **Recovery & other logs :** Transaction's life are recorded for several different purposes.

→ The two most important being transaction recovery support and transaction accounting.

(e) **Self-test support :** The transaction control tables have privileged modes that can be used for test and diagnostic purposes.

2) Hidden languages

→ The actual decision may be made in a processing Module, and the central dispatcher is usually in different to the transaction flows.

→ The dispatcher may direct the flows based on control code contained in the transaction's control block flows by or stored else where in the database.

Data-Flow Testing

Data-flow testing Basics:

Motivation and Assumptions:

→ 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 seaweences of events related to the statu of data objects.

Data Flow Machines

→ Data flow machines are programable computers that use Packets switching communications. Data flow machine supports the recursion. It is the faytest Mechanism

→ The prototype in data flow Machines is taken as a processing or Working element.

→ The problems in data flow machines

1) Distribution of computation and storage of data structures.

2) Stop Parallelism when resources tend to get overloaded.

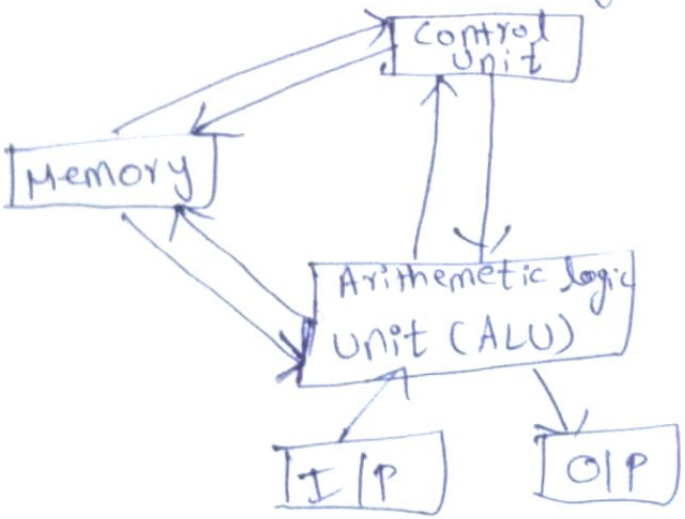
→ some of the data flow Machines are

1) Von neuman Machines
2) Multi Instruction Multi data (MIMD) Machines.

1) Von neuman Machine:

The computers which are being used today are von neumann Machines. The main elements are

(i) ALU (ii) Control Unit (iii) Memory (iv) Input (v) O/p.



Architecture of Von neumann Machine

Each instruction in this architecture is executed one by one.

(i) Fetch the instruction
(ii) Interpret instruction
(iii) Fetch operands
(iv) Execute instruction
(v) store the results in register
(vi) Increment Pc (Program Counter)
(vii) Repeat the process from Step (1)

2) Multi Instruction Multi Data (MIMD) Machines:

→ MIMD machines are massively parallel machines.

→ They fetch several instructions in Parallel. They are several Mechanisms for executing.

→ MIMD Machines can also perform arithmetic or logical operations simultaneously.

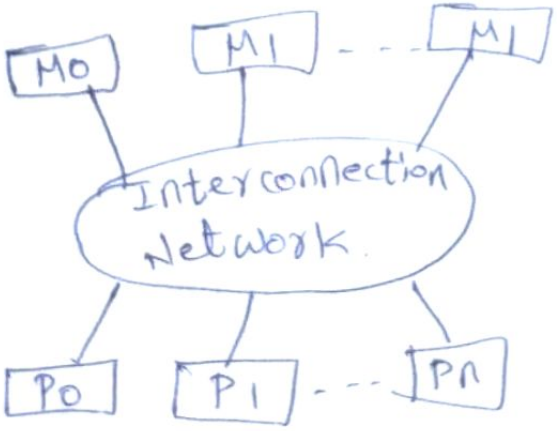
→ MIMD machines can be two categories

(a) shared Memory

(b) Distributed Memory.

(a) shared Memory:

→ The processors are connected to a "globally available" Memory via either a s/w or h/w.

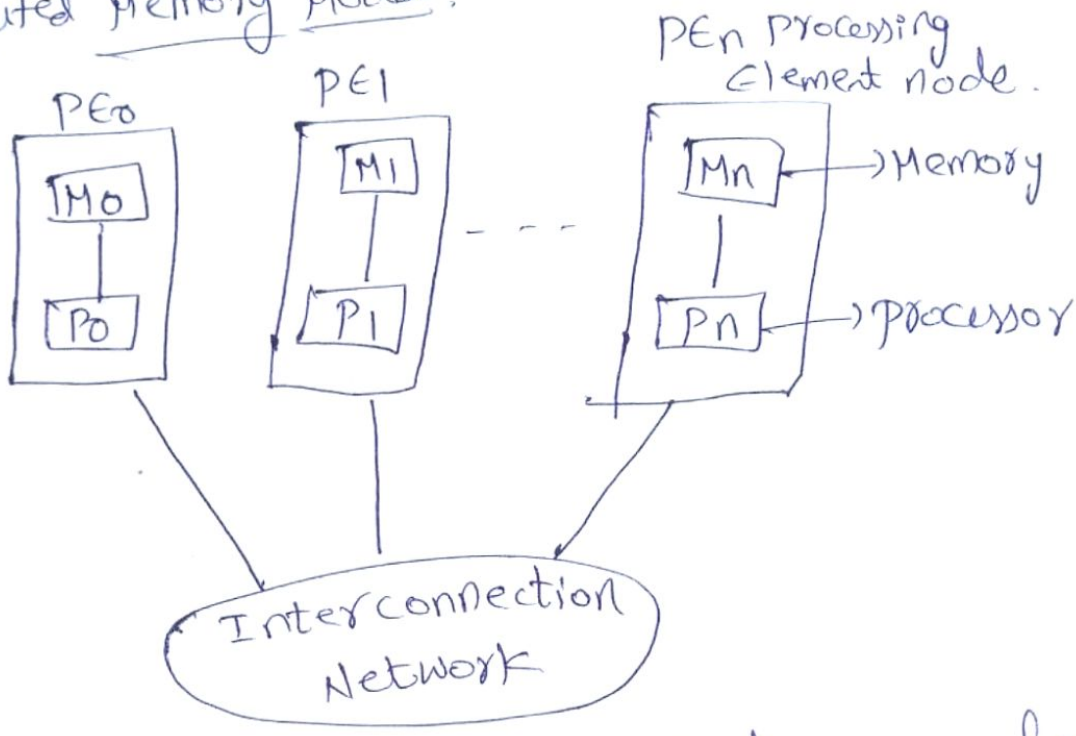


Shared Memory MIMD Architecture

M0, M1, M2 --- Mn → memory units

P0, P1, P2 --- Pn → processors.

(b) Distributed Memory Model:



→ Each processor has its own individual memory location.
→ Each processor has no direct knowledge about other processors Memory.
→ For data shared it must be passed from one processor to another as a Message.

Data Flow Graphs

→ The data flow graph is a graph consisting of nodes and directed links.
→ The data flow is between the data objects in the data flow graph.
→ Data flow graphs are a well known and good Modeling tool for the behavioural specification of a system.

29

Data objects state and usage:

→ Data objects can be created, killed and/or used. They can be used in two ways. in a calculation or as part of a control flow predicate.

d - defined, created, initialized etc
k - killed, undefined, released
u - used for something
c - used in a calculation
p - used in a predicate

1) **Defined**: An object is said to be defined when it is on the left hand side of assignment operator or when it appears in data declarations.

ex: i) z := x+y;
z is a data objects & represents sum of two numbers.

ii) FILE *p;
p = fopen ("professional.txt", "r");
p defined data object used which is referred to as opening of a file.

2) **Killed or undefined**: An object is said to be killed or undefined in few cases such as,
i) when it is released
ii) when it is made unavailable
iii) when it's contents are no longer

Ex: (i) for (i=0; i<10; i++)
{
c=c+i;
}

loop should be terminated after ten iterations and the variable i is not available after its terminations. So "i" is said to be killed.

(ii) a:=10;
a:=a+1;

The assignment statement can simultaneously kill and redefine. "a" value is killed and its redefind with the other value of "a".

3) Usage: Used data objects are the one which are used for computational purposes and also used in predicate.

Ex: (i) if (z>0)
z=z-1
'z' is a control flow element

(ii) z:=z+1
'z' is a computational purposes.

Data Flow Anomalies

→ An anomalies is denoted by a two-character sequence of actions.

→ Ex: ku means that the object is killed and then used. dd means object defined twice with out an intervening usage.

Ex. A := C+D

If A > 0 then X := 1 else X := -1

A := B+C.

The context of some secure systems it might be objectionable because the system doctrine might require.

Ex: A := C+D

If A > 0 Then X := 1 Else X := -1

A := 0
A := 0
A := 0
A := B+C.

→ Two letters combinations of d, k & u.

dd - probably harmless but suspicious.
dk - probably a bug.
du - the normal case.
kd - normal situation.
kk - harmless but probably buggy.
ku - a bug.
ud - usually not a bug because the lang permits reassignment at almost any time.
uk - normal situation
uu - normal situation.

23

Data Flow Anomaly State Graph

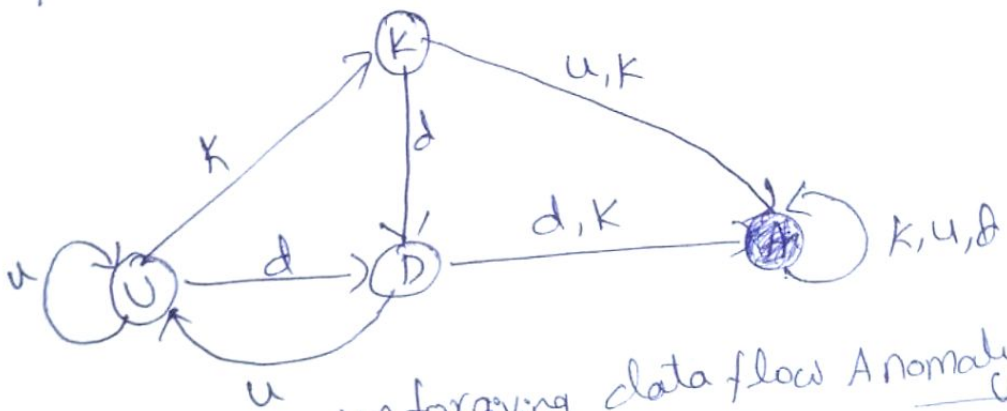
→ Data flow anomaly model prescribes than an object can be in one of four states.

K – undefined, previously killed, does not exit.

D – defined but not yet used.

U – has been used for computation or in predicate

A – Anomalous.



unforgiving data flow Anomaly state graph

→ K, U, D, A denotes the state of the variable.

→ k, d, u is a program action.

→ Variable starts the k state. i.e it has not been defined or does not exit.

→ The object's state becomes anomalous (state A) and once it is anomalous, no action can return the variable to a working state.

→ It is defined (d), it goes to the D, or defined but not yet used state. If it has been defined (D), and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state.

→ If in U, a redefinition (d) brings it to D, it keeps it in U, and k kills it.

Static versus Dynamic Anomaly Detection

Static Anomaly Detection

→ Static analysis is done at compiling time. This analysis is performed on source code with out executing it.

→ The quality is improved by removing bugs in the program.

→ Static analysis is done in design phase so that whole model can be analyzed and incoyistencies can be detected.

Dynamic Anomaly Detection

→ Dynamic analysis is done at runtime.

i. The languages such as Pascal that force variable declarations can detect -u and ku anomaly.

ii. optimizing compilers can detect some of the dead variables but not all.

1) Dead variables: The general problem is unsolvable eventhough it is possible to prove whether a variable is dead or not at a particular time.

2) Arrays: As array pointers are dynamically allocated, static analysis cannot reveal the problem of arrays.

3) Records and pointers: If any of the cases, we create files and their names dynamically.

4) Subroutines: The function names called when the program is being executed. So 'static analysis is not sufficient to analyse'.

5) False Anomalies: If the path is unachievable, even though a clear bug exits, it can't bedetected.

6) Recoverable Anomalies: ~~These anomalies even though a clear bug exits, it can't be detected~~.
These anomalies depend on context, application and semantics.

7) Interrupt: Anomalies become more sophisticated while moving from single processor surroundings to multiple processor enviroment.

Data flow Model

→ Data flow Model is based on the program's control flow graph. data flow consists of links which are denoted by the symbols of d, k, u, c, P or seaences of symbol like du, dd, ddd etc.

d - defined data objects
k - killed data objects
u - used data objects
c - calculation purpose
P - predicate value.

Components of data flow model

1) A unique node exists for each and every statement.
   i) Every node except exit and Entry node have atleast.
      ↳ one outlink
      ↳ one inlink
   ii) Exit node donot have outlinks
   iii) Entry nodes do not have inlinks.

2) The two types of nodes in the model
   i) Entry nodes
   ii) Exit nodes.
   ↳ Entry nodes are the dummy nodes which are placed at the entry.
   ↳ Exit nodes are dummy nodes which are placed at the exit.

3) The statements with only one outlink are known as simple statements and are weighted by proper sequence of data flow ~~controls~~ actions.

4) The statements such as if then else, Do while, case are known as predicate nodes are weighted with p-use on every outlink.

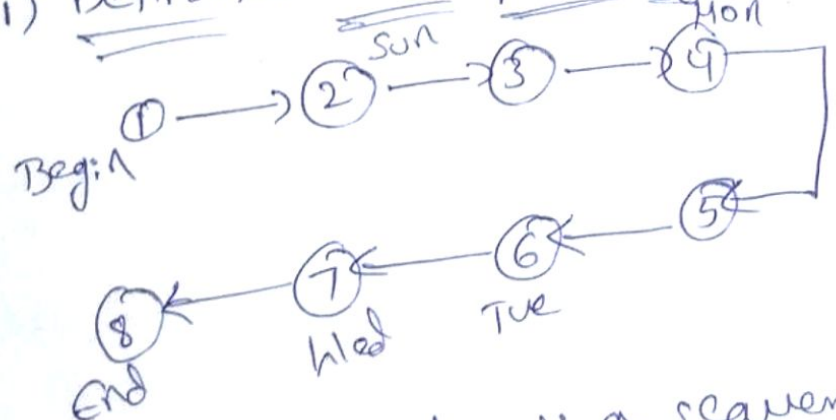
5) A pair of nodes can be placed in place of seaurences of simple statements.

Dataflow Testing strategies

Different terminologies have been defined to fulfill the graphs in Path testing.

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

1) Defination - clear path segment



→ A Path segment is a seaurence of connected links bln nodes.

-) The first link of the path is defined and the subseauent 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 non-defination clear.

-) The variable is defined on (2,3), (4,5) & again on (6,7).

-) There are many sub-paths b/n the nodes that are not definition-clear because some should have definations on node & som don't have definations on the nodes.

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

3) Simple Path segment. The path (4,2,3,4) is a simple path segment because the node 4 is visited twice in the path.

4) Du Path: The path segment from node a to c such that, if the last link has a computational uses of X, then this path is considered as a simple and defination-clear path.

Strategies

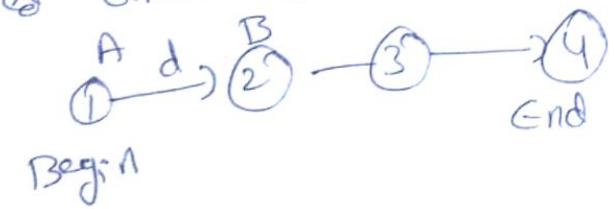
-> Data-flow testing strategies are based on the program control flowgraph.

-> They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set.

1) All-du Paths

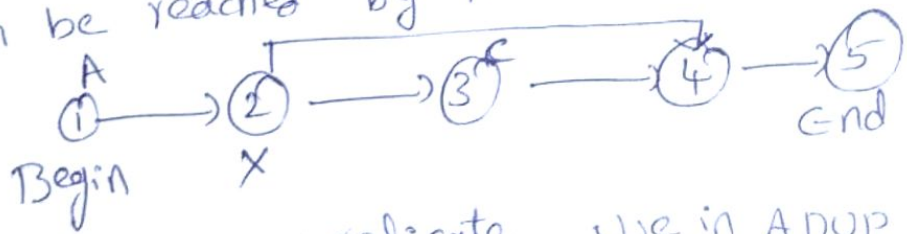
-> All-du paths (ADUP) strategy is the strongest data-flow testing strategy.

-> It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.



2) All Uses (AU) Strategy

-> All uses (AU) strategy is data flow testing strategy that requires every definition from atleast one path segment to use every variable that can be reached by that definition.

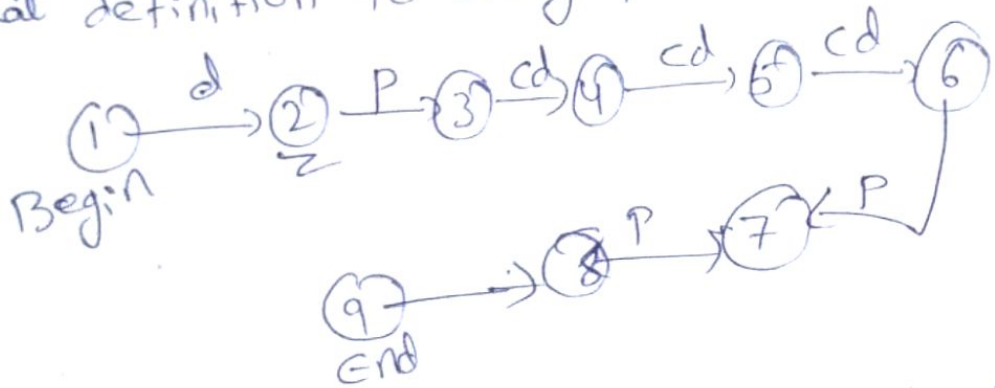


-> Variable x has a predicate use in ADUP strategy, the sub-paths (2,3,4) & (2,4) can be included in some test.

→ 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.

3) All P uses or some - C- uses (APU+C) Strategy :

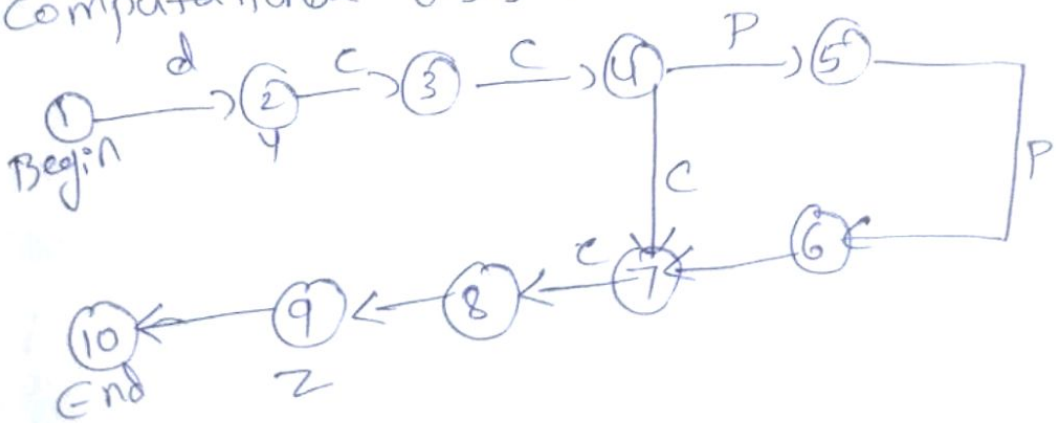
→ APU+C is a strategy that requires every definition of every variable included atleast one path from that definition to every predicate use.



→ APU+C strategy every definition of every variable has a path to every p-use of that definition.

4) All C uses or some P uses (ACU+P) strategy

→ ACU+P is a testing strategy that requires every definition of every variable included atleast one path from that definition to every computational uses.



→ ACU+P coverage is achieved for 4 by path (1,2,3,4,5,6,7,8,9,10) But several definations of predicate use are not covered, i.e the Path (4,5,6,7).

5) All-definations (AD) strategy:

→ AD is a testing strategy that requires every defination of every variable to cover at least one use that variable.

6) All predicate uses (APU) strategy:

APU is a testing strategy that requires a p-use of defination if there are no c-use following that defination.

7) All-Computational uses (ACU) strategy:

→ ACU is a testing strategy that reaulire a c-use of defination if there are not p-use following that defination.

Slicing, Dicing, Data-flow and Debugging

slices and Dices:

→ A static program slice is a part of a program defined with respect to given variable X and a stement i.

→ The set of all statements that could affect the value of X at statement i.

→ The influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.

→ If x is incorred 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.

Dynamic slicing:

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

→ Slicing Methods have been supported by tools and tired experimentally on a small scale.

→ current Models of slicing and dicing incorporate assumptions about bugs and programs that weaken their applicability to real programs and bugs.

Applications:

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

→ Dataflow testing shows the relationship b/n the data objects that represents data.

→ Data flow testing is cost effective.

→ Data flow testing uses practical applications rather that mathematical applications.

→ Dataflow testing used in developing Web applications with Java technology.

DOMAIN TESTING

Domains and Paths:

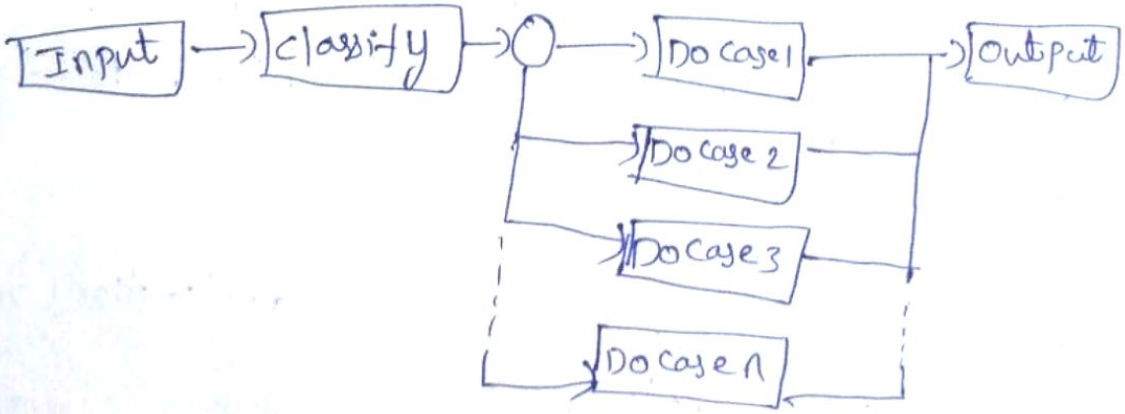
→ In Mathematics, domain is a set of possible values of an independent variable or the variables of a function.

→ Domain Testing can be based on specifications and or equivalent implementation information.

→ If domain testing is based on specifications, it is a functional test technique. if based on implementations, it is a structural techniques.

→ for example: Testing when you check extreme values of an i/p variable. Domain testing as a theory, however has been primarily structural.

→ All i/p's to a program can be considered as if they are numbers. for example a character string can be treated as a number by concatenating bits and looking at them as if they ~~are~~ were a binary integer. This is view in domain testing, which is why this strategy has a mathematical flavor.



Schematic Representation of Domain Testing

→ A schematic representation of this notion. processing begins with a classifier section that partitions the i/p vector into cases.

→ An invalid input is just a special processing case called "reject" say. The input then passes to a hypothetical subroutine or path that does the processing.

Domain 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 doesn't work well with arbitary discrete sets of data objects because there are no simple, general strategies.

Domains, paths and predicates

→ Domain testing, predicates are assumed to be interpreted in terms of i/p vector variables.

→ If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine. its based on the implementation control flow graph.

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

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

Domain closure

-> Domain closure is based mainly on the boundary points.

1) open
2) closed

1) Open Boundary: It is said to be a open boundary if the values does not belong to the same domain but belong to some other domains.

2) closed Boundary: It is said to be closed boundary if the values of the boundary points belong to the same domain.

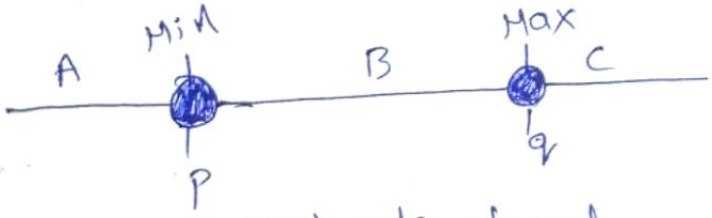
-> The domain boundaries or domain closure are,

(i) Both the boundary points are closed.

(ii) Domain is open on one side

(iii) Domain is open on both sides.

(i) Domain is closed on Both sides:



Both sides closed

-> Three domains A, B, C. The boundary points of domain B are represented as P and q. The filled circles indicate that the two boundary points are closed.

-> The two end points (P & q) are included in the boundary itself.

ii) Domain is open on one side

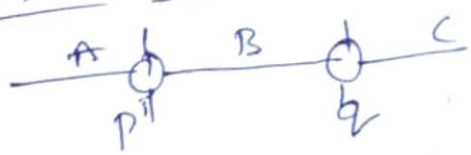
open at P

→ The domain is open at P. It can be considered as p does not include in domain B. and it is included in domain A. It can be represented as (p, q).

open at q

→ The domain is open at q. It can be considered as p includes in boundary or domain B where as q does not include in B. It includes in domain C. It represented as (p, q).

iii) Domain is open on Both sides



→ The domains are open at both end points p & q. It means both p & q values does not belong to the domain B. The value 'p' belongs to domain A & value q belongs to domain 'c'. Its represented as (p, q).

Domain Dimensionality:

→ Domain dimensionality means the number of planes in which the domain exists. It depends on the number of input variables.

→ Each boundary divides the i/p space with different dimensionalities lesser than the space dimensionality.

Ex: 1) Volume space can be divided by points, lines & planes.

2) n space can be divided by hyperplanes.

Bug Assumptions

→ The bug assumptions for domain testing means that the definition of the domain is wrong but the processing is correct.

→ The domains are implemented wrongly, the boundaries are taken wrongly and then control flow predicates are inaccurate.

1) Double zero representation:

→ Many of the languages there are two distinct zeros
i) Positive zero ii) negative zero.
Boundaries errors are common for negative zeros.

2) Floating point zero check: Floating point number can be equal to zero.
i) The number of multiplied by zero.
ii) The number is subtracted from itself.
iii) The number is assigned zero ~~in its previous~~

3) Contradictory Domains:
i) specified Domains
ii) Implemented Domains.

→ specified domain can be ambiguous or contradictory but an implemented domain cannot.

→ The contradictory domain means two different domains overlapping. In such situation the values belong either to one domain or other. So there is a chance of 50-50 error.

4) Ambiguous Domains: If there are some holes or missing values, then they are called ambiguous domains and there is a chance of error.

5) Overspecified Domains: There would be chance of error if domain path is unachievable or if the domain is overloaded with many conditions and the result would be null.

6) Boundary Errors:
(i) Missing boundary (ii) Extra boundary (iii) Tilted / shifted boundary.

7) Closure Reversal: This is a common bug. It occurs when the user chooses < = incase of >=.

8) Faulty logic: Due to fault in the logic of the code or program, all types of domain bugs can occur.

Restrictions

→ Domain testing has restrictions.

1) Coincidental correctness:

→ Coincidental correctness is assumed not to occur. Domain testing is not good at finding bugs for which the outcome is correct for the wrong reasons.

2) Representative out come:

-> Domain testing is an example of partition testing.

-> Partition testing strategies divide the programs input space into domains such that all i/p's with in a domain are equivalent in the sence that any i/p represents all i/p's in that domain.

-> Most test techniques, functional or structural, fall under partition testing and therefore make this representative outcome assumption.

3) Simple Domain Boundaries and Compound predicates

-> Simple predicates are used to define the boundaries rather than compound predicates. In compound predicate, if each part indicates a different boundary it is not a problem.

ex: $x > 5$ AND $x < 20$ it is a compound predicate which has two domain boundaries

Consider $y = 0$ AND $x >= 2$ AND $x <= 10$ this expression gives one boundary equation.

-> The compound predicates are removed in order to get simple convex and conneted domains.

-> To construct the simple and compound predicates, the relational operators such as >, >=, =, <=, <= are used.

4) Functional Homogeneity of Bugs:

-> The bug is it will not change the functional form of the boundary predicate.

→ For linear predicate the bug is such that the resulting predicate will still be linear.

Ex: $ax >= b$, the bug will be in the value of a or b. but it will not change the predicate $a^1 >= b$.

5) Linear vector space:

→ Linear boundary predicate is defined by a linear ~~inequ~~ inequality. Using the simple relational operators $>$, $>=$, $=$, $<=$, $<>$, and $<$.

→ A more general assumptions is that boundaries can be embedded in a linear vector space.

6) Loop-free software

→ Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression its a possible domain boundary change.

→ If a loop is an over all control loop on transactions, there is no problem.

Nice Domains AND Ugly Domains

→ Two types of Domains.
(i) specified Domains (ii) Implemented Domain.

(i) Specified Domain:

→ Specified Domains are incomplete or inconsistent. Incomplete means there are i/p vector for which path is not specified and inconsistences means over the same segment, there may be two or more contradictory specifications.

(41)

(ii) Implemented Domains:

No domains are incomplete or inconsistent; if there exists a incomplete or inconsistent, they are converted into consistent.

Nice Domains:

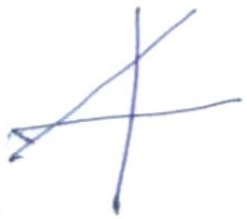
-> There are some properties that should be satisfied by the domain boundaries so that they called Nice domains.

-> The process of testing could be easy in case of Nice domains.

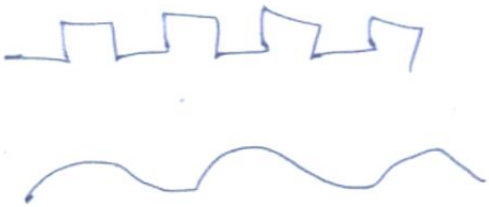
-> Nice Domains properties are 7 types.

1) Linear and Nonlinear boundaries:

-> Nice domains are defined by a property called "Linearity". Their boundaries represented in the form of linear equation or in equations.



Nice Domain



Not nice domain.

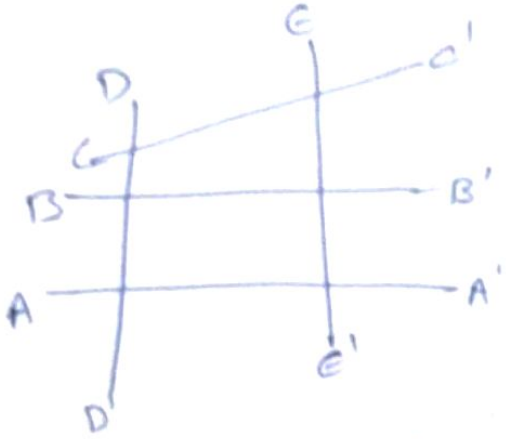
-> 99% of the Non linear boundaries can be converted to linear by applying transformations.

2) Complete boundaries:

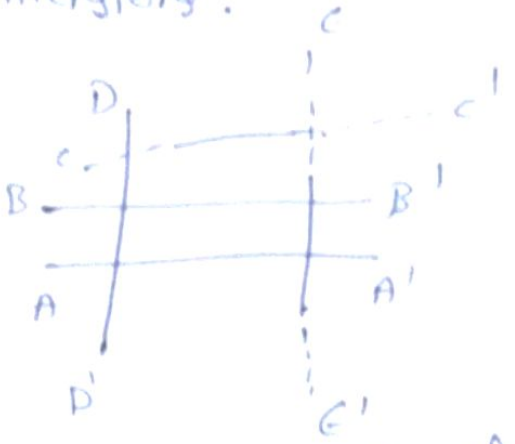
-> Complete Boundaries are the one which does not have any gaps bln them.

42

→ Nice domains have complete boundaries they cover a span of $+\infty$ to $-\infty$ in all dimensions.



Complete Boundaries



Incomplete Boundaries

3) Systematic Boundaries: the boundary inequalities are related by a simple function & a constant than they are known as systematic boundaries.

	P_1	P_2	P_3	P_4
Q_1	R_{11}	R_{12}	R_{13}	R_{14}
Q_2	R_{21}	R_{22}	R_{23}	R_{24}
Q_3	R_{31}	R_{32}	R_{33}	R_{34}

The domain boundaries of P & Q differ by only constant. Such domain are said to have systematic boundaries.

$f_1(x) \geq k_1$ $\quad$ $f_1(x) \geq g(1,c)$

$f_2(x) \geq k_2$ $\quad$ $f_2(x) \geq g(2,c)$

$f_i(x) \geq k_i$ $\quad$ $f_i(x) \geq g(i,c)$

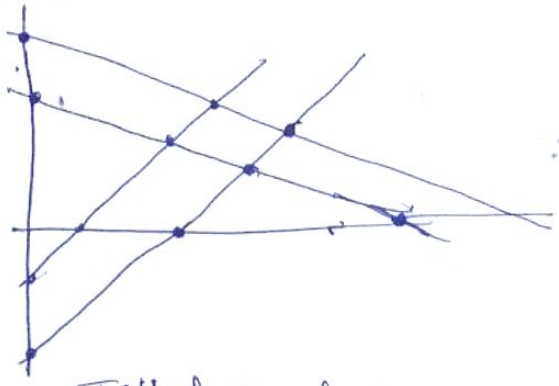
4) Orthogonal Boundaries:

	P_1	P_2
Q_1	R_{11}	R_{12}
Q_2	R_{21}	R_{22}

→ orthogonal if every inequality in P is perpendicular to every inequality in Q.

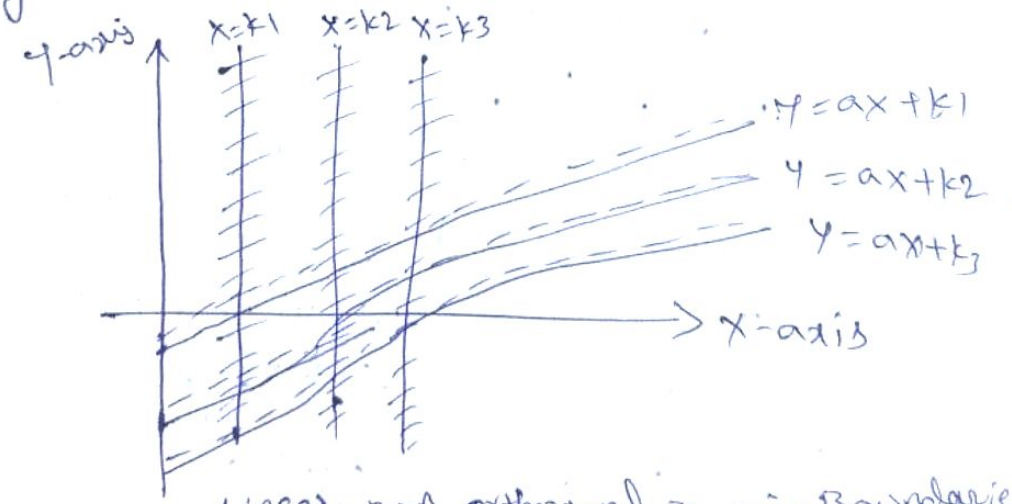
→ The two boundaries are orthogonal, the advantages is that they can be tested independently.

→ The no. of tests is proportional to the no. of boundaries. i.e O(n). The boundaries tilted the no. of cases would be $O(n^2)$.



Tilted Boundaries

Its concluded that they are two types of orthogonality conditions
(i) boundaries are orthogonal to one another and orthogonal to coordinate axis.
(ii) The boundaries are orthogonal to one other but not orthogonal to coordinate axis.



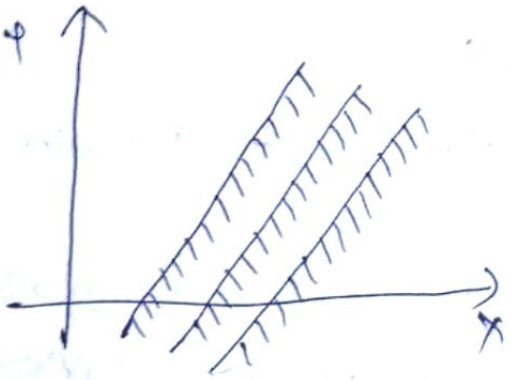
Linear non orthogonal Domain Boundaries

→ $X=K_i$ are perpendicular to x-axis. The other set of lines are systematic but not orthogonal to either coordinate axis or other set of boundaries.

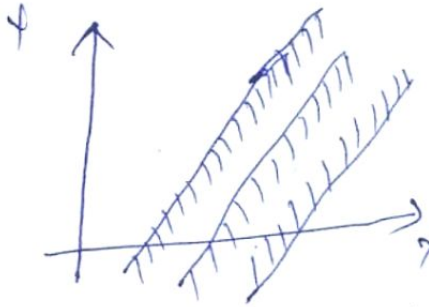
5) closure consistency:

→ closure consistency is the property of the nice domain that all the boundaries of the domain should indicate same direction.

→ IF used the same relational operator for all boundaries.

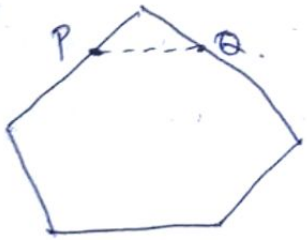


closure consistency

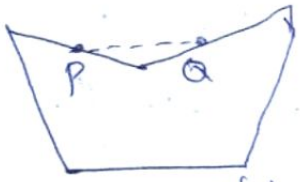


closure inconsistent

6) convex:



convex



Not convex

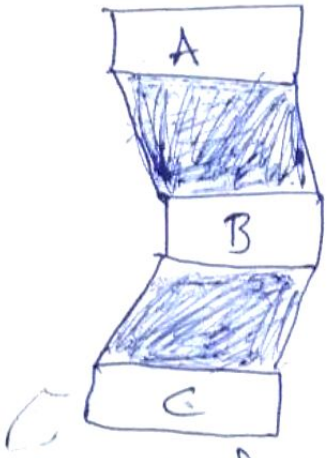
→ Convex when for any boundaries, with two points placed on them are combined by using a single line then all the points on that line are within the range.

→ In convex $\overline{PQ}$ lies inside the graph and in next case $\overline{PQ}$ is outside the ~~gene~~geometric.

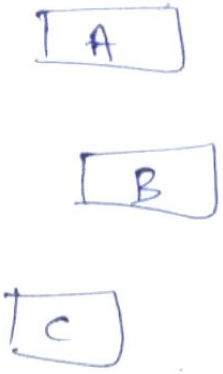
7) Simply connected: Nice domains are simply connected. It means that all the domains are considered as a single piece where as direct domains are connected.

45

→ They are considered as many number of Pieces.



Connected Domains



Not connected Domains

→ All domains are joined and considered as a single piece where as in second case, they are not connected.

Ugly Domains

→ Some domains are born ugly and some are uglified by bad specifications. Programmers in search of nice solutions will 'simplify' essential complexity out of existence.

1) Nonlinear Boundaries:

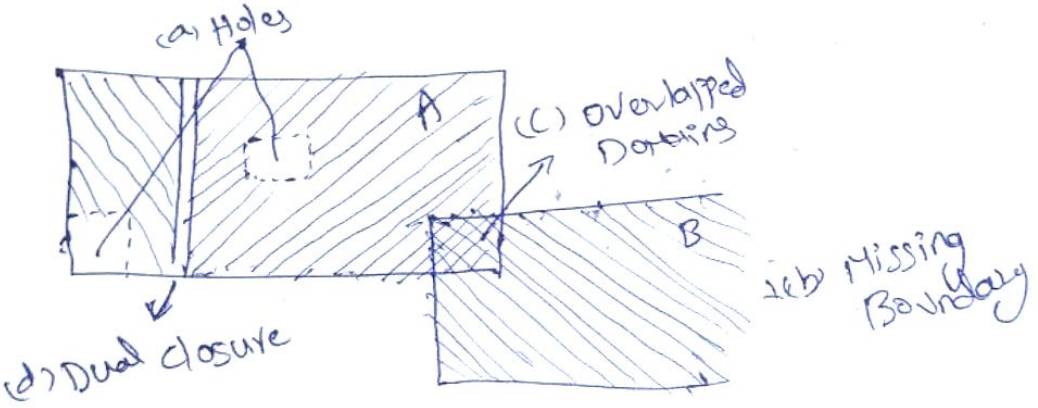
→ Nonlinear boundaries are so rare in ordinary Programming that there's no information on how Programmers might 'correct' such boundaries if they're essential.

2) Ambiguities and Contradictions:

→ Programs cannot be ambiguous but specifications can be there are several types of ambiguities

and contradictions of the domains such as,

(i) Holes
(ii) Missing Boundary
(iii) Overlapped domains
(iv) Dual closure.



(i) Holes; Holes domain ambiguities. Holes may occur anywhere in a domain or in cracks b/n the domains.
↳ A hole in one variable domain is easy to find.
↳ In a two variable domain, a hole is difficult to spot.

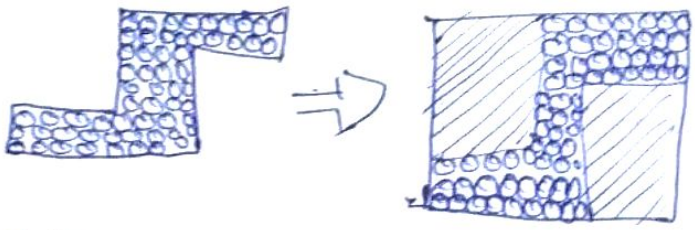
(ii) Missing Boundary: There should be no missing boundaries in the domains to call them as nice domains.

(iii) Overlapped Domains: This is a kind of contradiction.
↳ overlapped Domain specification
↳ overlapped closure specification

(iv) Dual closure: Dual closure assignment is a special case of overlapping.

2) Simplification of Topology:

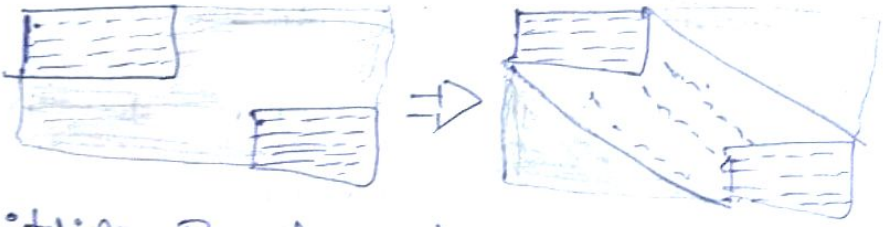
(i) Making convex: The concavities are smoothed into convex by smoothing techniques.



(ii) Filling the Holes: Holes present in the ugly domains are filled to make it into nice domains.

→ Holes are connected some missing values.

(iii) Join the pieces:



4) Rectifying Boundary closures:

→ The responsibility of the programmer and tester to apply a rule which helps to cover all cases in order to obtain consistent results.

Domain Testing

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

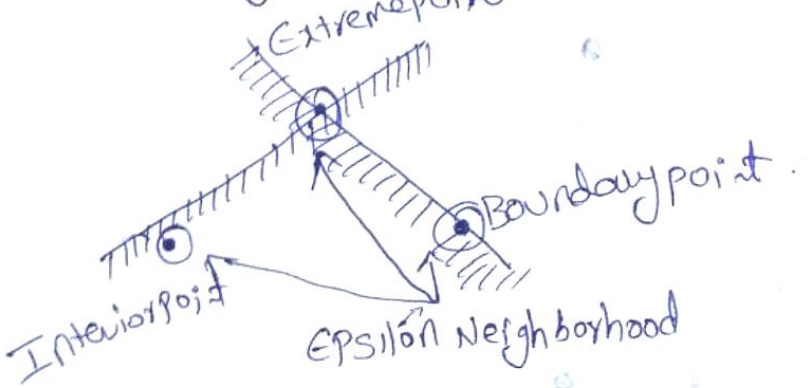
→ An interior point is a point in the domain such that all points with in an arbitarily small distance are domains.

Domain Bugs :.

(i) Interior point : It is a point in the domain such that all points with an arbitrarily small distance (called an epsilon neighborhood) are also in the Domain.

(ii) Boundary point : Boundary point is one such that with an epsilon neighborhood there are points both in the domain and not in the domain.

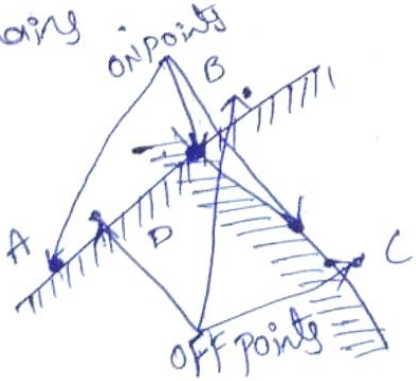
(iii) extreme point : It is a point that does not lie b/nary two other arbitary but distinct points of a domain.



Interior, Boundary, Exteme points

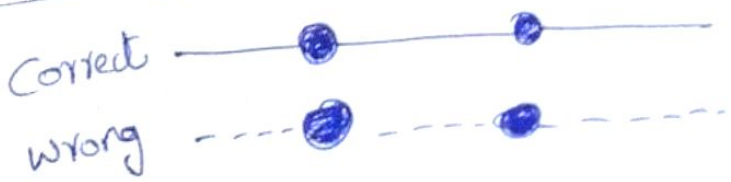
(iv) on point : It is a point on the boundary. If the domain boundary is closed.

(v) off point : It is a point near the boundary but in the adjacent domains

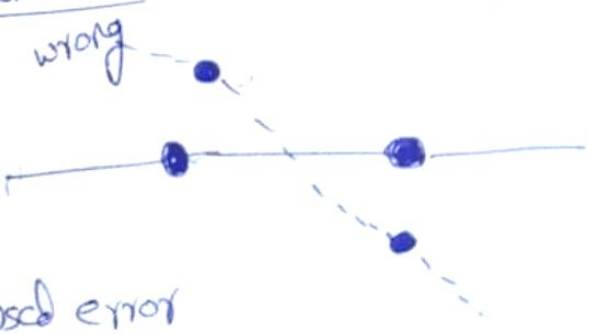


Testing one-Dimensional Domains

(i) shifted boundaries

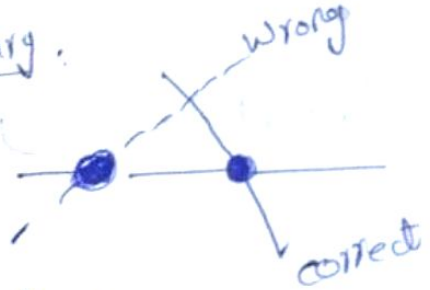


(ii) Tilted Boundaries

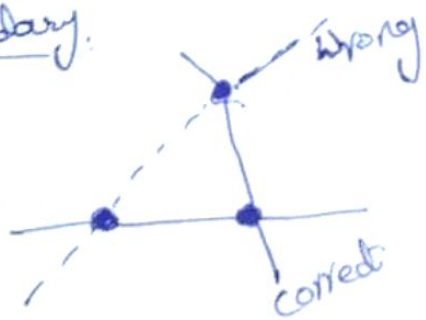


(iii) open/closed error

(iv) Extra boundary:



(v) Missing Boundary.



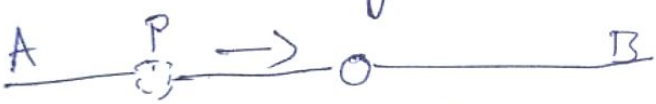
one dimensional domains, the occurance of bugs may differ for the (i) open boundaries
(ii) closed boundaries.

1) One dimensional open domain boundaries :

→ The open domain is represented by unfilled circle [o], The various bugs in open boundaries are,

i) shifted boundary :

Boundary shifted to left



Boundary shifted to right

→ The shifting of open boundary can be to the left or right. shifting of boundary causes a bug and the test point is used to identify whether the point has shifted to the right or left.

→ In boundary shifted to right the on point cannot tell about the shift because the test point (p) will be processed in domain.

ii) Tilted Boundary: Tilted boundary cannot occur ~~the~~ in one dimensional domains as this is caused due to change in the angle of two predicates.

iii) open or closed Error : The bug is caused by assigning wrong domain.

open domain for B

closure Error

→ The domain is open for B, if we make it closed for B, the closure bug occurs.

(iv) Extra boundary :- Extra boundary bug is occured when an extra point is added to the existing one.

It can detected by observing the two boundary values.

(v) Missing Boundary :

This can be detected by "open off Inside" point. If the boundary is open then the ~~point~~ off point is in D.

One Dimensional closed Domain Boundary :

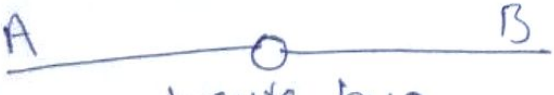
→ The closed domain boundaries are very much similar to that of open boundaries.

→ The only thing to be considered is unfilled circle is replaced with filled circle.

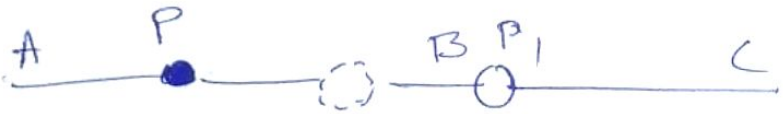
Boundary shifted left

Boundary shifted Right

closed Domain for B.



closure bug.



Extra boundary.

Missing Boundary.

Testing two Dimensional Domains :

The domain bugs in two dimensional boundaries are caused due to

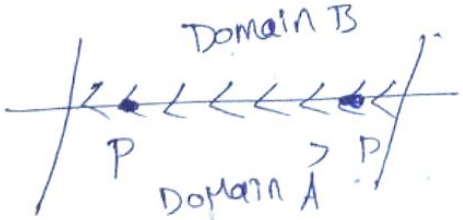
1) closure error 2) shifted boundary 3) Tilted boundary
4) Extra boundary 5) Missing boundary.

→ The bugs should be verified for the opend closed boundaries.

1) closed Boundaries :

Considering A & B are adjacent domains. The boundary is closed with respect to A. It implies that the boundary should be open with respect to B.

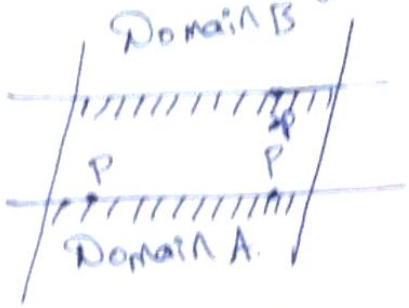
i) closure Error :



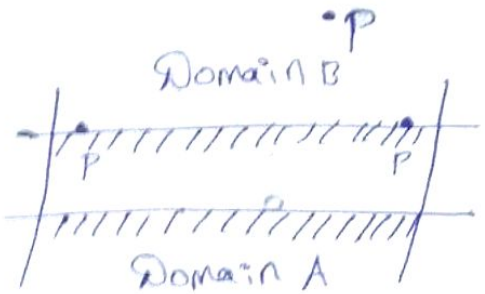
It may happen due to declaration of the operators i.e $P >= K$ instead of $P > K$.

Test strategy : The two on points could detect the bug. This is due to Wrong usage of domains or relational operators.

(i) Shifted Boundary:



Boundary shifted up



Boundary shifted Domain

Shifted up : The boundary is shifted up, because of which the domain B is divided into A'. They may happen due to wrong declaration of the constants in the equation.

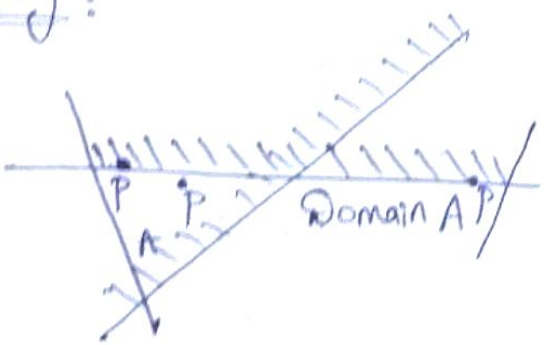
Ex: $x+y >= 10$ is declared as $x+y >= 5$.

Test strategy : The off point (p) is used to detect the bug which is "closed off outside".

Shifted Down : If the boundary is shifted down i.the domain A is subdivided into B.

Test strategy : This shift down bugs can be caught using the two on points.

(iii) Tilted Boundary :

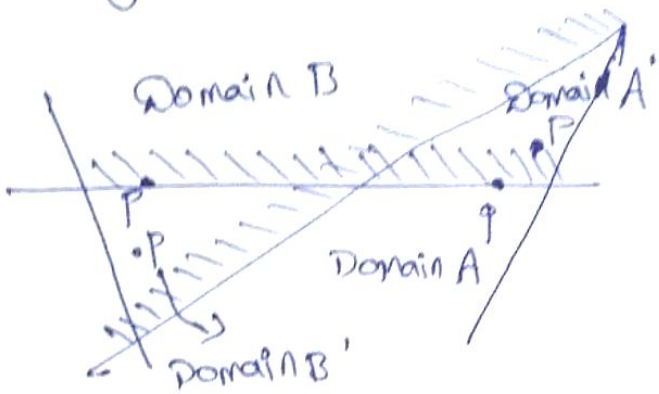


It is occured when the angle b/n the lines is changed i.e, coefficients in the in equality are changed.

Ex: If we take the in equality $10x+7y \geq 70$ instead of $7x+10y \geq 70$, a tilted boundary errors occurs. (54)

Test strategy: This bug can be caught by knowing few points which will change the domains while tilting.

(iv) Extra Boundary:

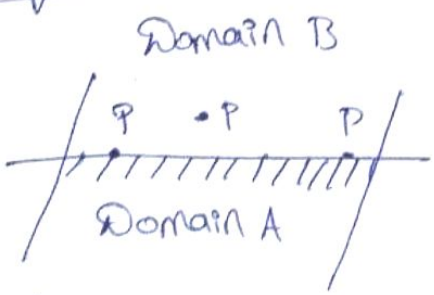


→ Extra boundary is occurred when the predicate value is increased. The extra boundary will divide the available space into many domains than before.

→ Domains earlier where A & B. with an extra bounday the domains now are A, A', B and B'.

Test strategy: This bug is detected by two on points and the redundant boundary should be removed or neglected.

(v) Missing Boundary:



Missing boundary can occur when any predicate among the set of predicates is left unnoticed.

Test strategy: The on points are used to detect the Missing boundaries.

2) Open boundaries: All the above bugs can be tested for open boundaries by making the up side down.

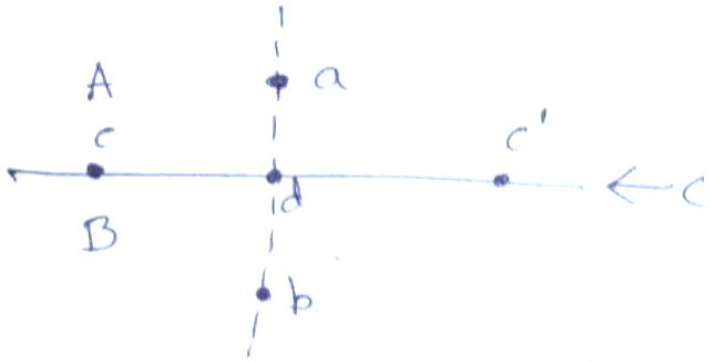
55

Equality and Inequality predicates

-> Equality predicates such as $x+y=17$ define lower dimensional domains.

-> Ex: There are two i/p variables, a two dimensional space, an equality predicate defines a line a one dimensional domain.

-> An equality predicate in three dimensions defines a planner domain.



-> A & B are planar while c, the equality boundary predicate b/n A & B is a line.

Random Testing

-> Random Testing is a kind of functional testing where tests are conducted on a randomly generated value.

-> It is not that efficient as direct Testing. But random Testing is done when the direct test cases takes much long time to write and run or in a case where the complexity problem.

-> Even random testing is performed for random values, care should be taken such that it covers the entire specifications.

Testing n-dimensional Domains

→ Generally it takes (n+1) x test points per domain to test the n-dimensional domains.

n → no. of dimensions

x → no. of boundary segment.

1) If there is possibility of extreme point sharing

no. of test points = 2x

2) If the domains are nice i.e they are orthogonal to coordinate axis, consistent closure etc, can be tested independently.

Procedures:

The procedure can be conceptualy implemented for any variables but practically, it can be applicable to 2 to 3 variables and not more than that

1) All the i/p variables must be identified.
2) The variables which appear in defining the predicates should be identified.
3) All the domain predicates must be interpreted in terms of i/p variables.

Domains And Interface Testing

→ Domain Testing is very difficult to be applied for more than two variables.

→ Domain testing is helpful in testing one variable at a time and this is really very helpful for Integration Testing.

1) Domain and Range:

-) Domain is the set of values taken as input by the given function.

-) By taking those values as input, the function gives the set of results as o/p which is known as "Range".

-) The testing technique considers the i/p values & checks for the o/p values that they are obtained by specified i/p's or not.

2) closure compatibility:

-) The process to check for the closure affinity b/n the domain & the range.

->The set of no's b/n the smallest and largest value is known as the "Domain span".

-) Thick number line for closed (.....)

-) Dashed number line for open (- - - -)

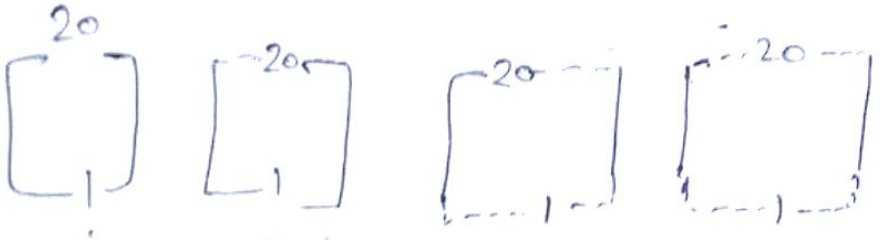
→ The four possible ways to represent the closure compatibility.

(i) Thick lines at top & bottom

(ii) Dashed line at top & thick line at bottom.

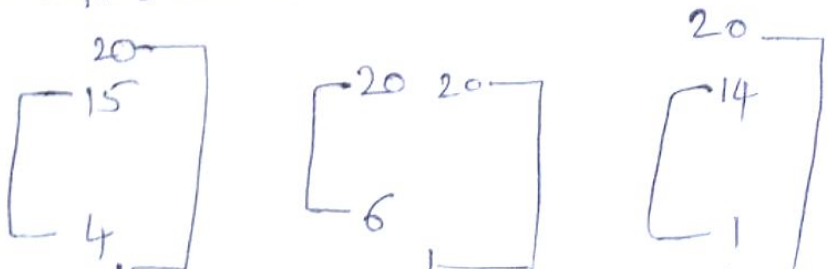
(iii) Thick line at top & dashed line at bottom.

(iv) Dashed lines at top & bottom.



(i) (ii) (iii) (iv)

3) Span compatibility : Span is nothing but a range of values. The three possible span incompatibilities of called and caller are represented.

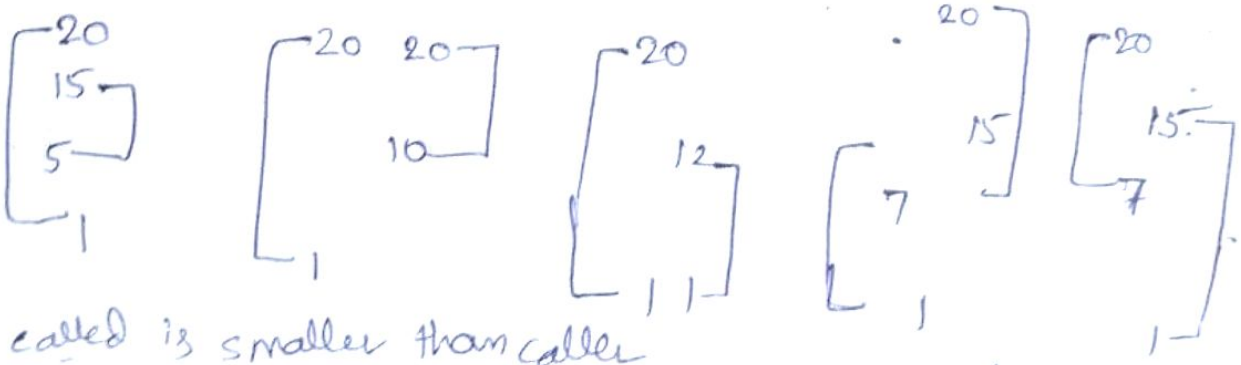


(i) (ii)

(i) It denotes the caller's range

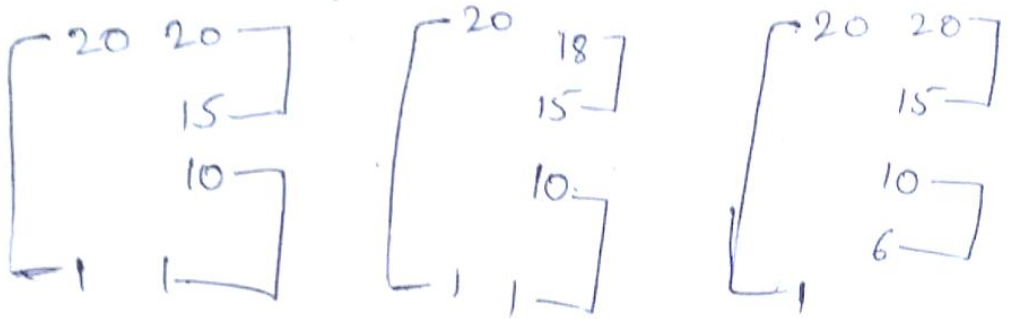
(ii) denotes the called's domain.

Q1. Consider a routine of square root. The square root of a -ve no. of gives a imaginary number which does not belong to the set of integers.



called is smaller than caller

Mismatch of Domain and Range



Holes called Domain

→ The three conditions of errors cases are.

(i) The called domain is smaller than caller's range.

(ii) The span of domain & range not matching.

(iii) The called domain has some missing values.

4) Interface Range & Domain capability:

→ The application of domain testing is also very important for interface testing because it tests the range & domain compatibilities among the caller and called routines.

→ The responsibility of the caller to provide the valid i/p's to the called routine.

Domains And Testability

→ Domain Testing need things to be very simple.

1) Linearizing Transformation

2) Coordinate Tranyformation.

1) Linearizing Transformation

→ The unlikely event that we're faced essential nonlinearity we can often convert nonlinear boundaries to equivalent linear boundaries. its applying the linearizing transformations.

(i) Polynomials: A boundary is specified by a polynomial or multinominal in several variables.

→ For a polynomial each term (for ex, $x, x^2, x^3 \ldots$) can be replaced by a new variable $y_1 = x, y_2 = x^2, y_3 = x^3 \ldots$ for multinomials you add more new variables for terms such as $xy, x^2y, xy^2 \ldots$.

→ The advantages is that you transform the problem from one we can't solve to one for which there are many available methods.

(ii) Logarithmic Transforms: products xyz can be linearized by substituting $u = \log(x)$, $v = \log(y)$, $w = \log(z)$.

(iii) More General Transforms:

Forms include $x/(ax+b)$ & ax^b. linearize (approximately) by using the Taylor series expansion of non linear functions.

2) Coordinate Transformations:

→ Parallel boundary sets are set of linearly related boundaries. They differ only by the value of a constant.

→ An $O(n^2)$ procedure for n boundary equations.

→ It has a coefficient a_i in inequality i, Divide each inequality by its x coefficient so that the coefficients of the transformed set of inequalities is unity for variable x.

→ If two inequalities are ||el, then all the coefficients will be the same & they will differ only by a constant.