Software Testing Methodologies

IV/I

Key Paper. (R22 CS743PE)

Part-A

1. a) **Debugging :-**

Debugging is the process of fixing the errors that were identified during the process of testing.

---

b) **Reasons for Testing :-**

The two major reasons / goals for testing are as follows ..

1. **Bug prevention :**

It is considered as the primary goals for testing. when a bug is detected, appropriate method should be used to remove it.

2. **Bug Discovery :-**

It is considered as the secondary goal for testing. It is performed when the primary goal fails to prevent the bugs.

c. Data flow Testing :-

It deals with path selection in control flow graph in order to examine the occurance of events related to data-flow anomalies.

Ex:

```
if (x>y)
  a = x+1
else
  a = y-1
```

---

d) Ugly Domain :-

Ugly domains are the domains which can be simplified by programmers & testers. These simplifications can be either good or bad depending upon the specificatn. If the domain's complexity is important & the programmers simplification is safe, then it will not cause any bugs.

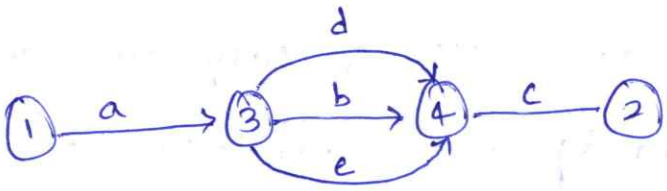
---

e) Path Sum with Example :

Path Sum is the sum of all parallel paths available between 2 nodes in which path passing through intermediate nodes are also considered as Parallel.

Path sum is denoted by "+" sign.

Ex:



Here d, b, e are links between the nodes 3 & 4. Hence, they are referred as parallel paths and represented by d + b + e.

---

f) Logic Based Testing :-

Logic based testing is a type of testing which is usually performed on structure & specification of a program. Therefore, it is structural testing when performed on structure & functional testing when performed on specification.

* First criteria is that one should clearly mention the functional requirements of software ase well as its design & test cases.
* Second criteria is to test & analyse the correctness, completeness & consistency of logic.

---

g) Good State graph :-

In good state graph, only one output action is specified for every transition because a single output gives best result rather than different graphs.

* The bugs are less & easy to identify.
* It has exactly one transition specified for every state & input combination. So the transition bugs may not occur.

---

h) State testing :-

State testing is defined as a functional testing technique used to test the ~~following bugs~~ functional bugs in the entire system. The principles of State testing are very similar to the principles of path testing.

---

i) Graph Matrices :-

The matrix in which every node of a graph is represented by one row & one column is called as graph matrix. In a graph matrix, each row-column interaction represents the relationship between the respective row nodes & column nodes.

---

j) Partitioning algorithm :-

It is an algorithm which is used to transform the graphs with loops in to partly ordered or loop-free graphs.

Part-B

2(a) **Testing:-**

Testing is a process of estimating the Productivity & quality of a software. Testing not only Performs program execution but also detect bugs that halts the execution of a program.

Testing is performed for the following main purposes.

(a) Quality assurance.

(b) Verification & Validation.

(c) Reliability estimation.

Purpose of Testing:

Testing consumes at least half of the time & work required to produce a functional program.

- Myth : Good programmers write code without bugs.
- History says that even well written programs still have 1-3 bugs per hundered statements.

Productivity & quality in software:

* In production of consumer goods & other products, every manufacturing stage is subjected to quality control

& testing from component to final stage.

* If flaws are discovered at any stage, the product is either discarded or cycled back for rework & correction.
* Productivity is measured by the sum of the costs of the material, the rework, & the discarded components, & the cost of quality assurance & testing.
* Testing & quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or a high as 80% in products such as space-ships, nuclear reactor, & aircrafts, where failures threaten life. Where as ~~soft~~ the manufacturing cost of software is trivial.
* For software, quality & productivity are indistinguishable because the cost of a software copy is trivial.
* Testing & Test Design are parts of quality assurance should also focus on bug prevention. A prevented bug is better than a detected & corrected bug.

2(b) Elements of flow graph :-

The elements of flow graph are :

1. process blocks.
2. Decision & case statements.
3. Junctions.

1. Process blocks :-

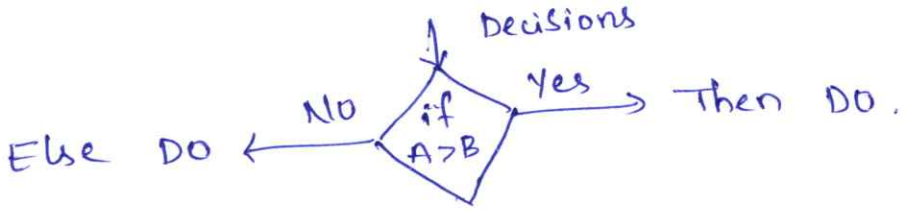
Process block is a block which consists of sequence of statements, which once initiated results in the execution of all the statements in that block.

Process block.

2. Decision & case statements :-

(a) Decision :

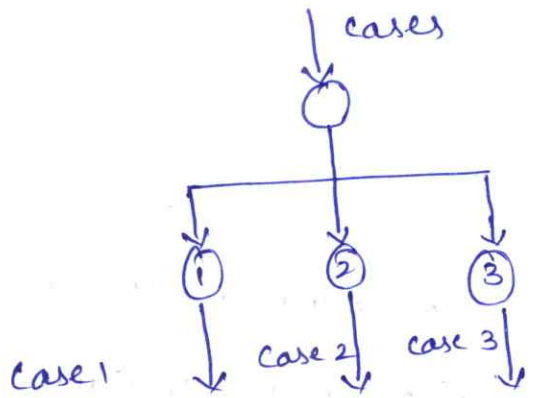
Decision is a point in a program at which the content flow can split. The flow gets diverted in one of the main options available.



Two-way Decisions.

(b) case statements :-

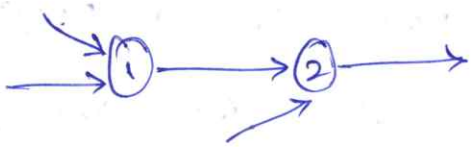
Any decision can split the control flow into different way branches. this multiway branches can be termed as case statements.



case statements.

3) Junctions :-

A junction is just a contradict to decision. All the control flows can merge at a point in a Program which can be termed as junction.



Junctions

3) Bug :-

A bug is a fault or failure of software program. It produces an unintended or incorrect result. Generally bugs are caused due to mistakes done by programmers while coding. However bugs can also result due to incorrect coding in compilers.

Consequences of Bugs :-

The consequences of bugs can range from mild to catastrophic. The programmers write programs for humans rather than in terms of machine. The various bug consequences are as follows.

1. Mild :

This consequence results in an incorrect, misspelled or ~~use~~ wrongly aligned output.

2. Moderate :

This consequence effects the performance of the system & hence it results in a duplicate output.

3. Annoying :

Because of the presence of bugs in the system, the performance of the system degrades. For ex :

(i) The names are shortened or changed.

(ii) Bills for even null amount are sent.

4. Disturbing:

Because of this consequence, even the correct transactions are not executed. For ex, an 'automata Telles Machine' refuses to process the 'withdrawal' transaction.

5. Serious:

The information about the transactions gets lost such as.

(i) Tracking of transactions.

(ii) Accountability of a transaction.

(iii) Transaction occurrence.

6. Very Serious:

This consequence results in reversing the operations. The transaction such as, deposit transaction is converted to withdrawal transaction. Hence, the system is forced to Perform incorrect transactions.

7. Extreme:

This consequence occurs frequently & is not limited to small number of users or transactions.

8. Intolerable:

When large amount of data is corrupted it because of very difficult to peeform the recovery process. in, this

situations, shutting down the system is considered as the best option.

9. Catastrophic:

Because of this consequence, the system fails and the right to shut down the system is taken from the users.

10. Infectious:

A system that does not fail but corrupts the other systems results in the following consequences.

(i) Destroys the physical conditions.

(ii) Liquefies the nuclear reactors.

(iii) Kills the system.

(iv) Prevents the system from normal functioning.

4) Nice & ugly domains with examples:

Nice Domains:

There exists some properties related to domain boundaries. If these properties are applied then it makes domain testing very simple or nice, otherwise it makes domain testing very difficult.

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

(i) Linear and Non-Linear Boundaries :-

Nice domain boundaries can be described by using linear inequalities of equations. The effect on testing comes from that fact that it considers only 2 points then it specifies a straight line. If it considers 3 points, then it represents a plane Linear boundaries are more frequently used than the non-linear boundaries.

(ii) Complete Boundaries :-

Complete boundaries are those boundaries which do not have any gap between them. Nice domains boundaries have complete boundaries because they cover from plus infinity to minus infinity in all dimensions.

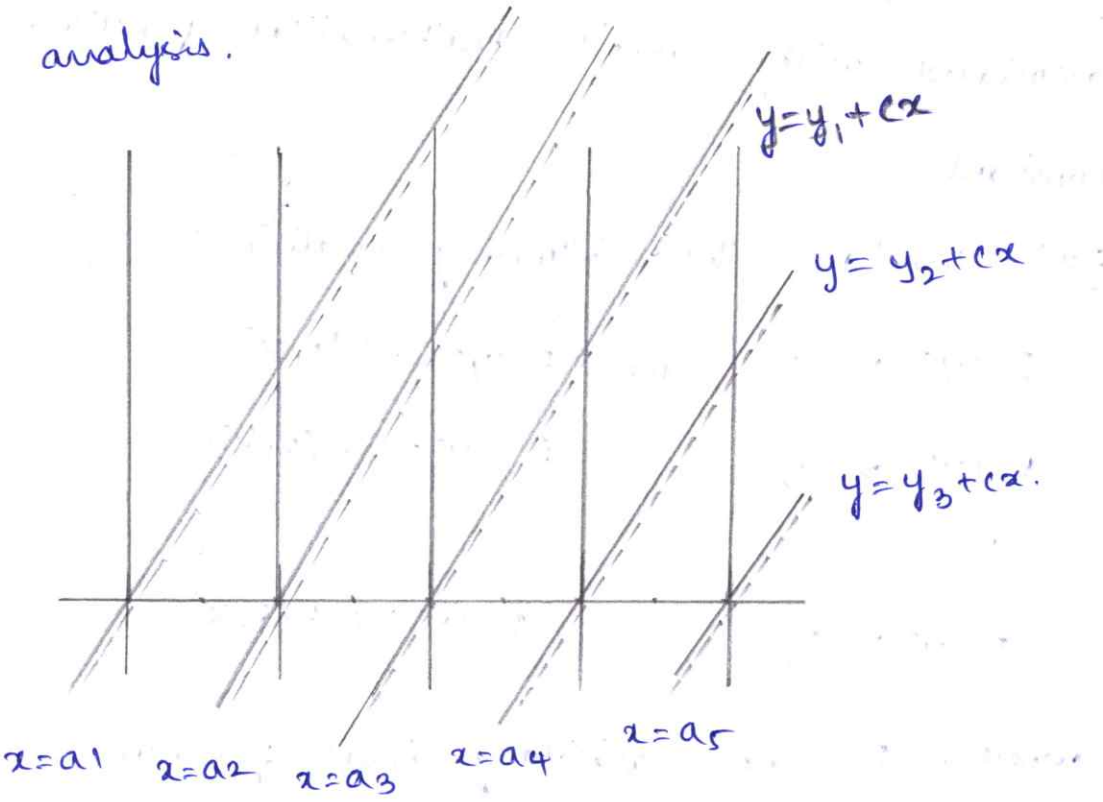
(iii) Orthogonal Boundaries :- The table shows an example of Orthogonal boundaries.

B \ A	A1	A2	A3	A4	A5	A6
B1	Dom1	Dom2	Dom3	Dom4	Dom5	Dom6
B2	Dom11	Dom12	Dom13	Dom14	Dom15	Dom16
B3	Dom21	Dom22	Dom23	Dom24	Dom25	Dom26
B4	Dom31	Dom32	Dom33	Dom34	Dom35	Dom36

In the table, A & B are 2 boundary sets that are orthogonal to each other which means that the inequalities of A & B are perpendicular to each other.

(iv) Consistent closure:

Consistent closures are the most simple & fundamental closures that give systematic results after performing the required analysis.



Boundary closures are consistent.

Convex:

A figure is said to be convex when, for any 2 boundaries with 2 points placed on them are combined by using a single line then all the points on that line are within the range of the same figure.

Simply connected:

Nice domains are simply connected because they are available at one place as a whole but not dispersed in other domains.

Systematic Boundaries:

Systematic Boundaries refer to boundary inequalities associated with simple Mathematical functions such as a constant.

Ex: Consider the following relations:

$F_1(Y) >= c_1$ or $F_1(Y) >= G(1,K)$

$F_2(Y) >= c_2$ $F_2(Y) >= G(2,K)$

⋮

$F_i(Y) >= c_i$ $F_i(Y) >= G(i,K)$

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.

UGLY DOMAINS:-

The domains that are born ugly ~~(i.e., with~~ or are uglified by specifications of the user are referred to as ugly domains.

The following are some of the simplifications of ugly domains.

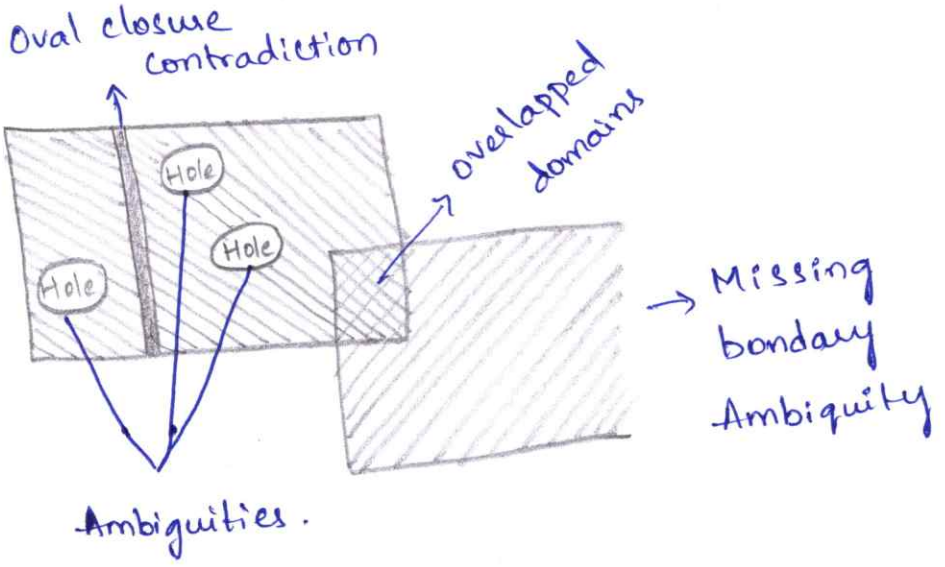
(i) Nonlinear Boundaries:

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

Ambiguity & contradictions:

Programs cannot be ambiguous or contradictory whereas specifications can be the same.

Domain ambiguities lead to holes or cracks in the i/p space as shown. A hole in one-variable input space can be easily viewed whereas identifying ambiguity in 2 variables i/p space is difficult to identify.



Ambiguities.

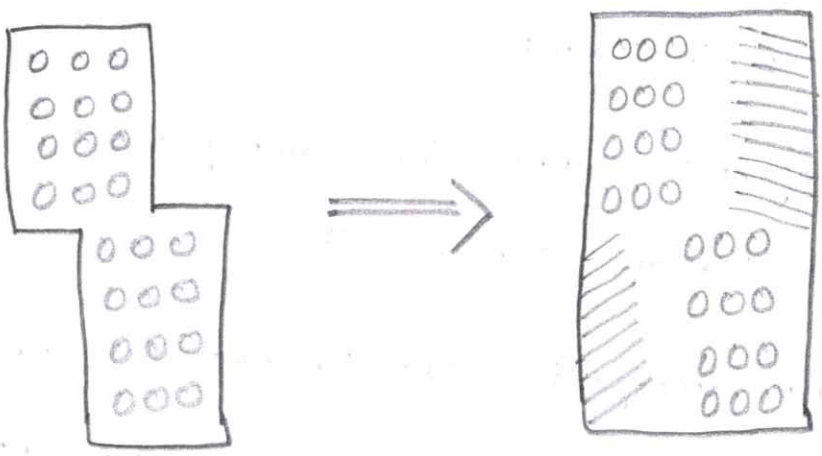
Domain Ambiguities & contradictions.

(ii) Simplification of Topology :-

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

(a) Milding Out Concavities :-

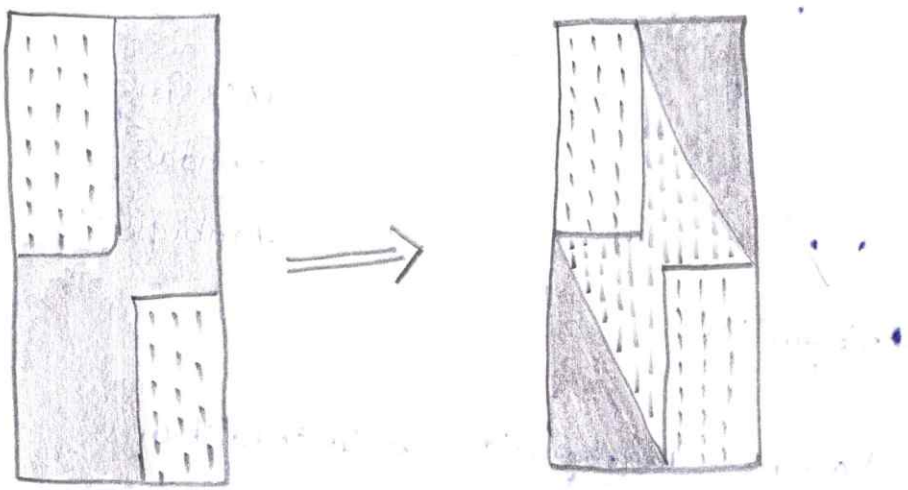
Bugs are introduced by the programmers which inturn lead to the misdesigning of test cases by the testers.



Milding out concavities.

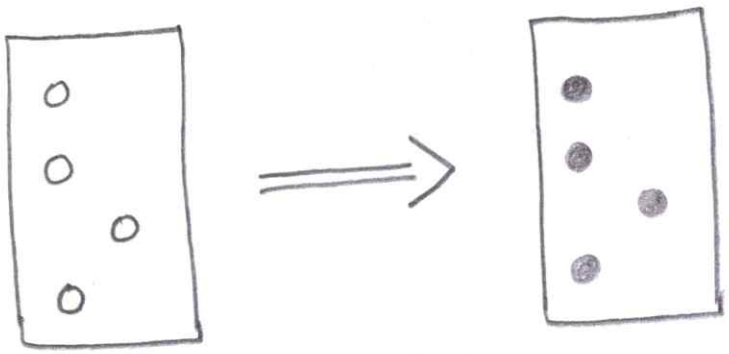
(b) Connecting Disconnected pieces :-

The testers will connect the disconnected pieces into a single one.



c) Filling the holes:

The testers will fill up the empty holes as shown to simplify the domain topology.



Filling of holes.

iii) Fixing Boundary closure:

It is the responsibility of programmer & tester to apply a rule which helps to cover all cases in order to obtain consistent results.

For ex:, if there are 2 domains & the programmer wants to remove the spaces b/w them, then he can apply a rule to make all boundaries to follow the same direction.

5(a) Restrictions for domain testing:-

The following are restrictions of domain testing.

(i) General restrictions.

(ii) Coincidental correctness restrictions.

(iii) Representative outputs restrictions.

(i) General Restrictions:

Basically every testing technique has certain restrictions no testing technique is completely accurate. Similarly, the main testing also has certain restrictions. So, one should be aw of those restrictions to avoid wrong decisions. There are main cases in which if domain testing is used then it will result correct errors or bugs.

(ii) Coincidental Correctness Restrictions:-

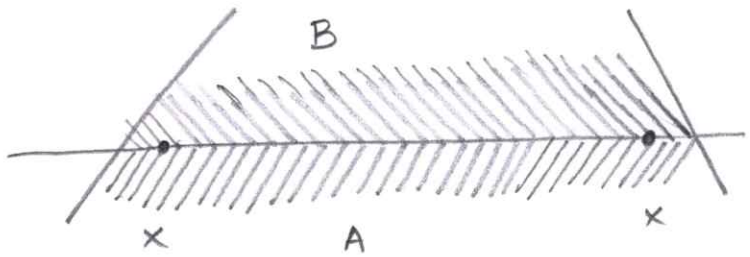
If the output or result of a program is incorrect then lying domain testing to the program is just waste of time. In though if one is not concerned about the output which results from domain testing, analysis needs to be done. This is because, if incorrect boundary is measured then it is often very difficult to find out the true output. One important point to be held here is that, domain testing does not support boolean becomes(true/false).

iii) Representative Output Restrictions:

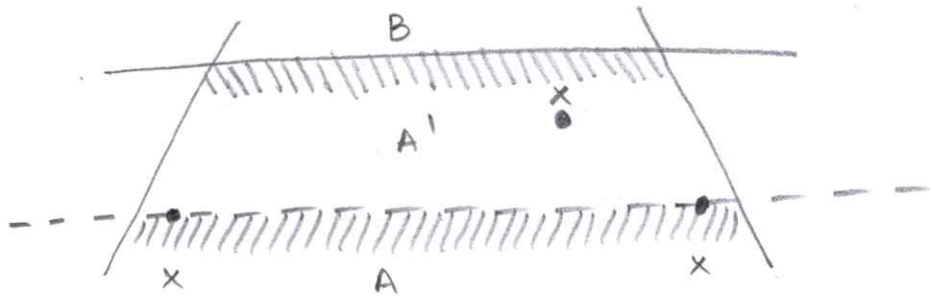
Domain testing follows the concept of partition testing in such an input space is partitioned into some specific domains such a way that all the inputs within that domain are equal to each other. If the test proves the selected input as correct all the inputs within the specific

domain are assumed to correct. Almost all types of testing follow the concept of interaction testing & therefore it can be said that these functions result in representative output assumption.

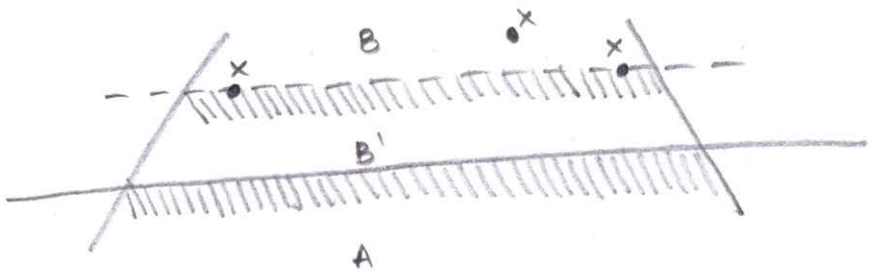
5(b) Two Dimensional Domains and path testing :-



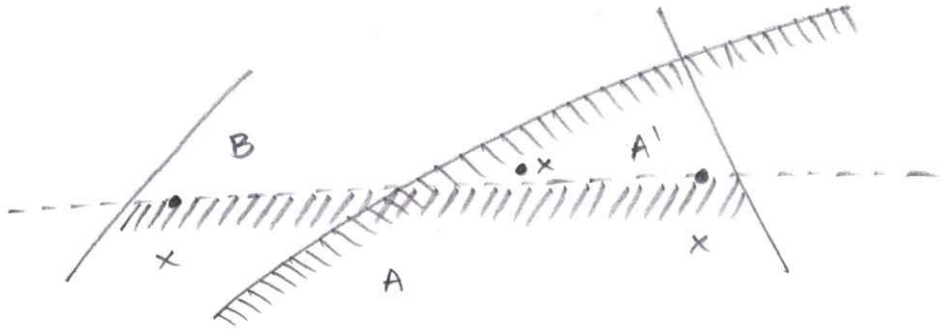
(a) closure Bug



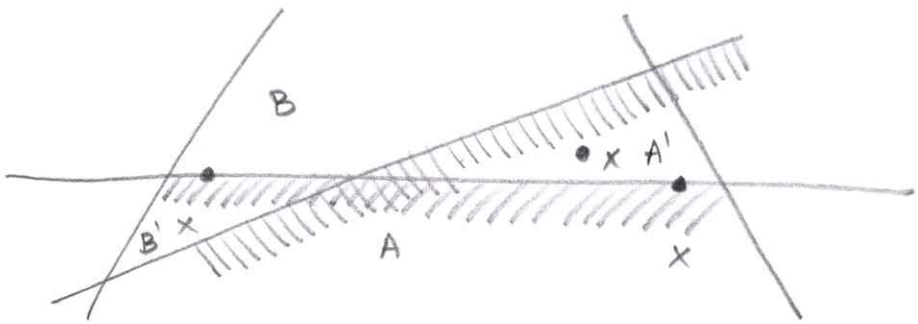
(b) shifted up



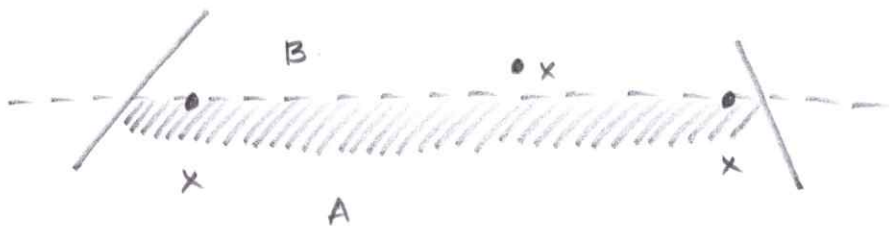
(c) shifted down



(d) Tilted Boundary.



(e) Extra Boundary



(f) Missing Boundary

Domain boundary bugs for two-dimensional domains. A & B are adjacent domains & the boundary is closed with respect to A, which means that it is open with respect to B.

1. Closure Bug: Fig. shows a faulty closure, such as might be caused by using a wrong operator. The two on Points detect this bug because those values will get B rather than A processing.

2. Shifted Boundary: In fig. the bug is a shift up, which converts part of domain B into A processing, denoted by A. This result is caused by an incorrect constant in a predicate, such as $x+y >= 17$ when $x+y >= 7$ was intended.

3. Tilted Boundary: A tilted boundary occurs when coefficients in the boundary inequality are wrong. For Ex: $3x+7y > 17$ when $7x+3y > 17$ was intended. Fig shows has a tilted boundary, which creates erroneous domain segments A' & B'. In this example the bug is caught by the left on point.

4. Extra boundary: An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains & will therefore cause many test failures for the same bug.
Note that extra boundary must result in different functions for A' versus A & B' versus B. If it doesn't i.e., $f(A) = f(A')$ and/or $f(B) = f(B')$.

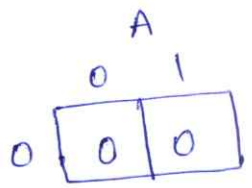
5. Missing Boundary: A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains &, as the extra boundary can, will cause many test failures although there is only one bug.

6) Karnaugh Veitch Matrix :-

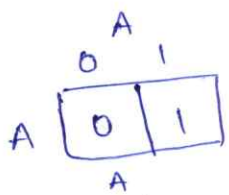
The Karnaugh-Veitch chart known by every combination of "Karnaugh" &/or "Veitch" with any one of "map", "chart", or "diagram" reduces boolean algebraic manipulations to graphical trivia.

Simple Forms

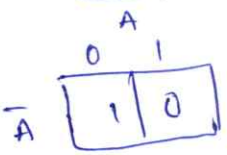
All the boolean functions of a single variable & their equivalent representation as a KV chart. The charts show all possible truth values that the variable A can have.



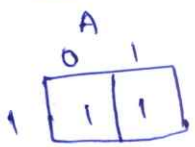
The function is never true.



The function is true when A is true



The fn is true when A is false.



The fn is always true.

KV charts for functions of a single variable.

A "1" means the variable's value is "1" or True.

A "0" means that the variable's value is "0" or false.

B \ A	0	1
0	z(0,0) 0	z(1,0) 2
1	z(0,1) 1	z(1,1) 3

Two variable karnaugh veitch matrix.

C \ AB	00	01	11	10
0	z(000) 0	z(010) 2	z(110) 6	z(100) 4
1	z(001) 1	z(011) 3	z(111) 7	z(101) 5

Three variable KV chart.

00 \ 01	00	01	11	10
00	z(0000) 0	z(0100) 4	z(1100) 12	z(1000) 8
01	z(0001) 1	z(0101) 5	z(1101) 13	z(1001) 9
11	z(0011) 3	z(0111) 7	z(1111) 15	z(1011) 11
10	z(0010) 2	z(0110) 6	z(1110) 14	z(1010) 10

Four Variable kv chart.

The steps to determine minimal logic function are as follows,

1. choose a kv matrix according to the number of input

Variable present.

2. Arrange the matrix by placing 1 in the corresponding input vector value.

3. Determine the largest set of adjacent 2 cells. They are usually formed in the following manner,

(i) An adjacent group usually consists of 2, 4 or 8 cells group which can be formed from powers of 2 ie.., 2, 4, 8, 16, 32.

(ii) In a 3 variable KV of matrix, adjacent group can be formed with the right edge & left edge.

(iii) Adjacent group can be made either vertically or horizontally, but not diagonally.

(iv) In a four variable KV matrix, adjacent group can be formed with the right edge as well as left edge or with the top & bottom edge.

(v) write the product term for all the groups formed. It is done by taking each group one at a time.

(vi) The product is transcribed if the group contains only 1's.

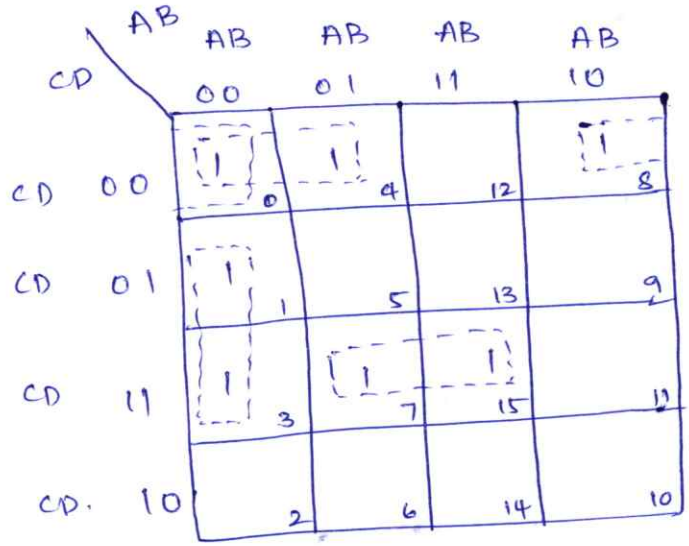
Example: Consider the minimization of given expression using four variable K-map.

$$F(A, B, C, D) = \Sigma m(0, 1, 3, 4, 7, 8, 15)$$

It is in the form of 'sum of products', we write

ones in the cells corresponding to maxterm for 0 outputs.

In sum of products, all the ones that are adjacent form a group.



Therefore the minimized function is,

$\overline{B}CD + \overline{A}C\overline{D} + \overline{A}\,\overline{B}\overline{D} + BCD$.

---

7) Logic Based Testing:-

→ The functional requirements of many programs can be specified by decision tables, which provide a useful basis for program & test design.

→ Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using kv-charts.

→ "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.

- Boolean algebra is to logic as arithmetic. ~~is to programmers vocabularies but one of their least used techniques.~~ without it, the tester or programmer is cut off from many test and design techniques & tools that incorporate those techniques.
- Logic has been, for several decades, the primary tool of hardware logic designers.
- Many test Methods developed for hardware logic can be adapted to s/w logic testing. Because hardware testing methods and its associated theory is a fertile ground for s/w testing methods.
- The trouble with specifications is that they're hard to express.
- Boolean algebra is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.

7(b) Regular Expressions and Flow-anomoly detection:-

A Regular expression represents a set of strings in a compact form. It is used for expressing finite and infinite test sequences. It plays an important role in flow anomoly detection.

Flow-anomoly detection problem is used to search for particular sequence of operations. This is done by

travelling through all paths that are achievable in the flowgraph.

Consider the operations GET/RETURN that are denoted by variable q and r respectively.

Flow-anomaly detection is used to know if the sequence of qq (or) rr has occurred or not. i.e., if GET is followed by GET operation or if RETURN is followed by RETURN operation.

Flow-anomaly detection consider sequence but not the total impact of procedure. It is used to detect the bug sequence in following situations.

1. The operations that are performed on a file are open(o), close(c), read(r) and write(w). If operations read/write are performed on a file that is already closed, then sequence crlcw are said to be anomalous.

2. The operations performed by tape-transport device are read(r), write(w), rewind(d), forward(f), skip(k) & stop(p). These are certain rules that are to be followed while using a tape-transport device which involves rewind and forward operations.

3. With the help of generic flow-anomaly detection, it is possible to detect the dataflow bugs sequence such as dd, kk, dk, ku.

Huang Theorem:

The regular expression obtained should be simplified carefully as the null operations cannot be combined with other operations. The resultant expression represents all possible sequence of operators in the flow graphs.

Huang's theorem is used to simplify the regular expression and to examine the specific operation sequence. Consider P, Q, R as non-empty sets of character sequences that consist of atleast one-character. Let 'M' be a 2-character string. If 'M' is a substring of sequence PQ^iR, it means that the string 'M' is present in the path PQ^2R.

Consider the following example,

$P = aa$

$Q = bcc$

$R = ca$

$M = bb$

According to Huang's theorem, bb will appear in sequence $aa(bcc)^n ca$, if it appears in sequence $aa(bcc)^2 ca$. It is not necessary to use the theorem to check if bb appears in the given string.

8) State Graphs and Transition Testing :-

State Graph :-

A state graph ~~consists of a set of~~ is simply defined as a graphical representation of the system behaviour. It is used as a functional testing tool for state testing to identify state bugs, transition bugs etc. It consists of following terms.

(i) States :

A state graph consists of a set of states that represent certain characteristics of the system to describe its behaviour. States are denoted by nodes where each node can be identified by either number of characters.

(ii) Inputs :

A state graph takes inputs that can be values or variables provided to states. The changes to the states are based on inputs. They can be denoted by numbers or characters.

iii) Transitions :-

The links joining the states are called transitions. When an i/p is applied, the state changes which means that the system has made a transition. The i/p value that causes this transition is written on the link & it is called link weight.

There exists one outgoing link from every state for every input.

iv) Outputs:

Outputs are based on the input operations performed on the states. When an input is applied to a state, it is processed in order to produce an output. Each input & output of the state graph are seperated by slash '/' symbol. An output can be related to any line from the states and denoted by either numbers or characters.

Consider an example:

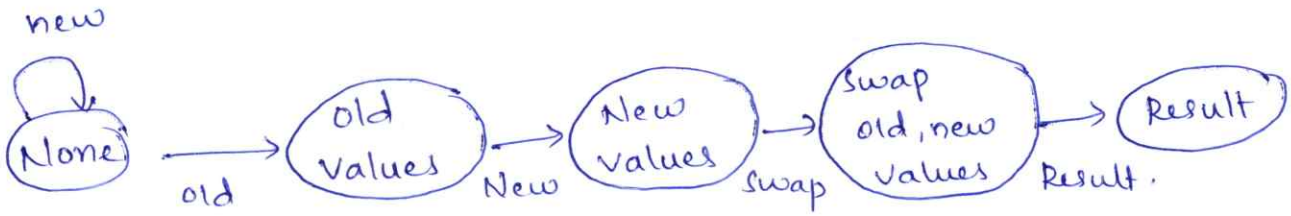
(i) in first state, neither old values nor new values are initialized.

(ii) old values are initialized.

(iii) new values are initialized.

iv) both old values and new values are interchanged using swap function.

v) the resultant swapped values are displayed.



Program State Graph.

Transition Testing:-

A technique in which the states of a system are recognized and test cases are made to test the transition of one state to another is called transition testing. It is based on Finite State Machine (FSM).

Finite State Machine:

FSM is an important mechanism in software engineering as a functional testing tool and programming testing tool. FSM can be represented by a state graph which consists of a finite number of states as well as a finite number of transitions between those states and actions. It is an essential tool for state testing used to identify or model the behaviour of software.

A Finite State Machine is simply a set of states, among which one of the states is the starting state and the other is the end state. Each state has a set of transitions to other states based on the next input applied. A transition has a condition and an action.

9) Principles of State Testing :-

State Testing:

State testing is defined as a functional testing technique used to test the functional bugs in the entire system. The principles for state testing are very similar to the principles of path testing.

For path-testing, it is not possible to test every possible path in a flow graph. Similarly, for state testing, it is not possible to test every possible Paths in a state graph. In a state graph, a path is a sequence of transitions caused by a sequence of inputs.

One of the imp principles of a state testing is that it allows the testers to give the priority to the interesting nodes in order to initiate the state. In state testing, the primary interest is given to the states and transitions rather than outputs.

To Begin the state testing, perform the following:

1. Define a set of input sequences such that when a system starts from the initial state, after going through all states it gets back to the initial state.

2. Define expected next state, expected transition and also expected output code for each step in each input sequence.

3. At last, a set of tests define three sets of sequences, transition sequences and output sequences to complete the test.

Advantages :-

The advantages of states testing are as follows,

(i) State testing is useful when the error corrections are less expensive.

(ii) It is also useful when the testers want to detect a specified input or when the order of input sequence is important.

(iii) It is specifically designed for catching the deep bugs.

(iv) It provides rewards during the design of the tests.

Disadvantages :

The disadvantages of state testing are as follows,

(i) State testing does not provide thorough testing because when a test is completed, there might be some bugs remained in the system.

(ii) Testers require large number of i/p sequences to catch transition errors, missing states, extra states & the link.

10 a) Node Reduction Algorithm :-

Node reduction algorithms are the set of tools / steps to be followed in order to remove a node from a pictorial graph or graph matrix. These algorithms are mainly used for removing all the intermediate nodes between an entry and an exit thereby creating a direct link between the 2 nodes. Once all the intermediate nodes are removed, debugging can also be done easily because of a direct link present between the initial and the final node with a single path expression.

Matrix reduction method is similar to the node-by-node reduction procedure of the pictorial graphs, but it is well structured and ordered. Moreover, node reduction procedure in terms of matrix operations eliminates the need of drawing the graphs multiple times.

Algorithm for Node Reduction :-

Step-1 : choose the node to be removed and replace it with the links that passes through that node.

Step-2 : If this removal of the node leads to any parallel links, combine all those parallel links thereby adding their respective path expressions to form a single link.

Step-3: check all the loops. The nodes having self loops, alter their outerlinks relative to values of their loops.

Step-4: This results in a reduction of graph matrix size by 1.

Step-5: Repeat the above steps until all the intermeadiate nodes are removed.

10.b) Building tools of Graph Matrices:

The building tools of graph matrices are as follows,

1. Representation of Matrices in Software:

A graph can be used to provide visual representation of the software structures, which when used in theorem proving, their matrix representations are needed to be applied.

Linked lists can be used to store and process the graph matrix stored in a computer, because mostly row - column entries in the graph matrices are Empty.

(a) Degree of Node and Density of Graphs:

Each intermediate node may have any number of inner links and outer links. The no. of inner links and outerlinks of a node represents the indegree and

outdegree of that node respectively. The sum of indegree and outdegree of a particular node gives the degree of that node. Its value can be any one of the following,

(i) Average degree of a node for a graph is between 3 and 4.

(ii) Degree of a simple branch and simple junction is 3.

(iii) Degree of a loop in a graph is 4.

Note: Any graph with average degrees of 5 or 6 is said to be a very busy flow graph.

(b) Merits and Demerits of Graph Matrix Representations.

Demerits:

A graph is an abstract representation of a software structure. Tracing a path in a graph is not an easy task since there are many risks associated with it.

Matrix representations make use of 2-dimensional array which is optimal only for small graphs with simple link weights. However, with larger graphs, Matrix representation may lead to the following inconveniences, if implemented.

(i) Large Storage Space:

Matrix representation requires a large storage space of order n^2. Hence, a linked list representation is used which requires less storage space for storing graphs.

(ii) weight Matrix:

An additional weight matrix is needed for complicated link weights.

(iii) Time Duration:

Since many entries of the graph matrices are null / vacant, the time taken to process and remove such entries is a complete wastage.

(iv) String Array:

A Two-dimensional string array consisting of large no. of null entries is needed, if the link weights are in the form of regular expressions.

Merits:

Apart from various inconveniencies, matrix representation also has several merits or uses

i) Matrix representation forms the basis for constructing test tools and can be obtained by using analysis routine.

ii) Matrix representation yields the best results, when the number of nodes are within the range of 20 & 30.

iii) Linked list representation is used to represent graph matrices.

---

11) Graph Matrices and their applications :-

Graph matrices are introduced as another representation for graphs; some useful tools resulting therefrom are examined. Matrix operations, relations, node-reduction algorithm revisited, equivalence class Partitions.

Problem with pictorial Graphs:

* Graphs were introduced as an abstraction of sw structure.
* Whenever a graph is used as a model, sooner or later we trace paths through it - to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable or not.
* path is not easy, & it's subject to error. You can miss a link here and there or cover some links twice.

* One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they are more reliable.

Tool Building:

* If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on these methods.
* It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to tool building.

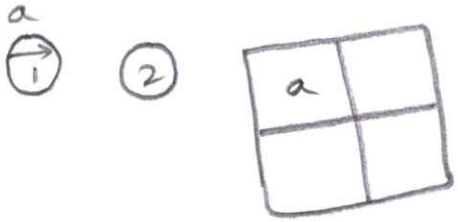
The Basic Algorithms:

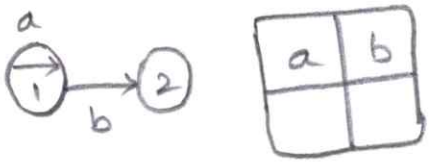
* The basic tool kit consists of:
* Matrix multiplication, which is used to get the path expression from every node to every other node.
* A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes.
* A collapsing process which gets the path expression from any node to any other node.

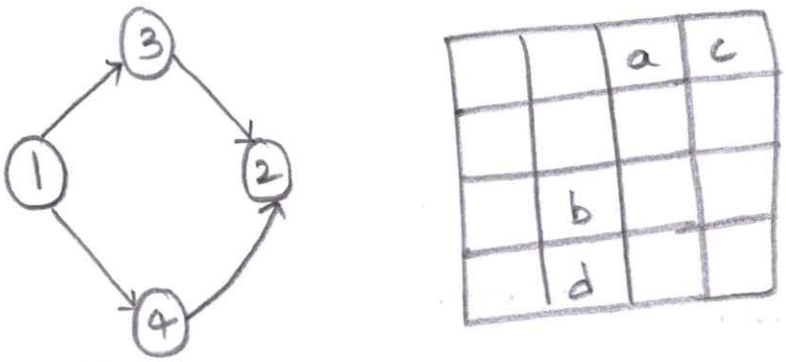
The Matrix of a Graph:

* A graph matrix is a square array with one row and one column for every node in the graph.
* Each row-column combination corresponds to a relation between the node corresponding to the row & the node corresponding to the column.
* The relation for Eg, could be as simple as the link name, if there is a link between the nodes.
* Some of the things to be observed:
  - → The size of the matrix equals the no. of nodes.
  - → There is a place to put every possible direct connection or link between any and any other node.
  - → The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.
  - → A connection from node i to j does not imply a connection from node j to node i.
  - → If there are several links between 2 nodes, then the entry is a sum; the '+' sign denotes parallel links as usual.

Some Graphs & their Matrices.







A simple weight.

* A simplest weight we can use is to note that there is or isn't a connection. Let '1' mean that there is a connection and "0" mean that there isn't.

The arithmetic rules are:

$1+1=1$ $\quad 1*1=1$

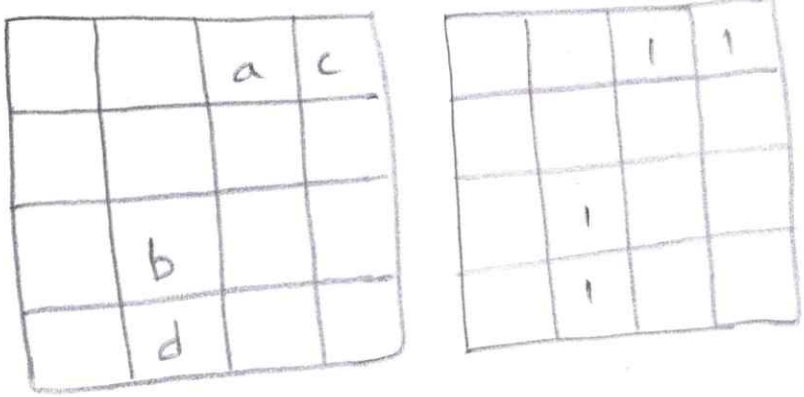
$1+0=1$ $\quad 1*0=0$

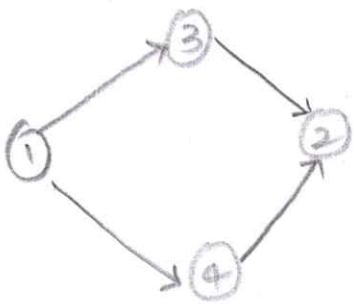
$0+0=0$ $\quad 0*0=0$

→ A matrix defined like this is called connection matrix.

Connection matrix:

* The connection matrix is obtained by replacing each entry with 1 if there is a link & 0 if there isn't.





Connection Matrix - continued.

- Each row of a matrix denotes the out links of the node corresponding to that row.
- Each column denotes the in links corresponding to that node.
- A branch is a node with more than one nonzero entry in its row.
- A junction is node with more than one nonzero entry in its column.
- A self loop is an entry along the diagonal.