UNIT - I.

Part - I : Introduction.

⇒ purpose of testing.
⇒ Dichotomies.
⇒ Model for testing
⇒ consequences of bugs
⇒ taxonomy of bugs
⇒ Flow graphs and path testing : Basics concepts of path testing.
⇒ predicates
⇒ path predicates and achievable paths
⇒ path sensitizing.
⇒ path instrumentation.
⇒ application of path testing.

Part-I: Introduction.

* purpose of Testing:-

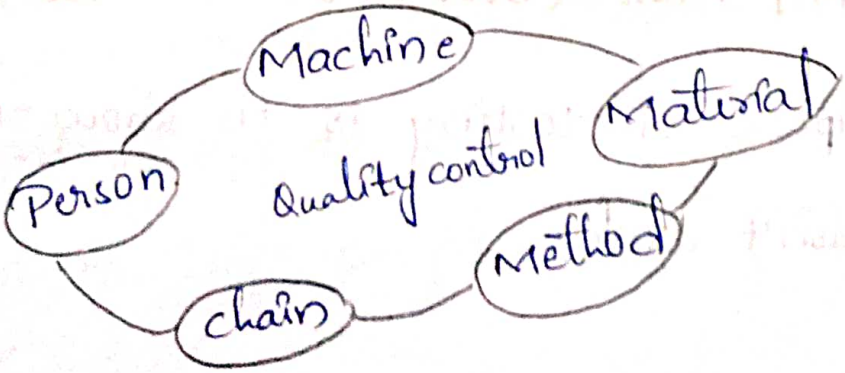
1). Testing consumes atleast half of the time and work required to produce a functional program.

2). History reveals that even well written programs still have 1-3 bugs per hundred statements.

* productivity and Quality in software:-

1). In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.

2). If flaws are discovered at any stage, the product is either discarded (or) cycled back for rework and correction.



* Goals for Testing:-

1). Testing and test design are parts of quality assurance that should also focus on bug prevention.

2). Phases in testers mental life can be divided into the following 5 phases.

Phase 0: until 1956: (Debugging oriented)

1). There is no difference between testing and debugging.

2). phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.

Phase 1: 1957 - 1978: (Demonstration oriented)

1). The purpose of testing here is to show that software works.

2). Highlighted during the late 1970's.

Phase-2: 1979-1982: (Destruction oriented).

1). The purpose of testing is to show that slw doesn't work.

phase-3: 1983-1987: (Evaluation oriented)

1). The purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value.

phase-4: 1988-2000: (prevention oriented)

1). Testability is the factor considered here, once reason is to reduce the labour of testing.

2). Other reason is to check the testable and non-testable code.

* Test Design:-

1). software design is the process of implementing software solutions to one (or) more set of problems, one of the important parts of slw design is the software Requirements Analysis.

* Methodse-

1). Inspection Methods: Methods like walk throughs, desk checking, formal inspections & code reading appear to be as effective as testing, but the bugs caught do not completely overlap.

2). Design style:- while designing the software itself adopting stylistic objectives such as testability, openness and clarity can do much, to prevent bugs..

* Dichotomies :-

① Testing Vs Debugging:-

S.NO	Testing	Debugging.
1.	Testing starts with known conditions, uses predefined procedures & has predictable outcomes	Debugging starts from possibly unknown initial conditions and the end can not be predicted except statistically.
2.	Testing can be done by an outsider	Debugging can be done by an insider

② Function Vs structure:-

SNO.	Function	structure.
1.	It takes the user point of view - bother about functionality and features and not the programs implementation	It looks at the implementation details.
2.	In functional testing, the program (or) system is treated as a black box.	Things such as programming style, control method source languages database design, and coding details dominate

③. Designer Vs Testers:-

1). Test designer is the person who designs the test, whereas the tester is the one who actually tests the code.

2) During functional testing, the designer and tester are probably different persons.

④. Modularity Vs Efficiency:-

1). A Module is a discrete, well-defined, small component of a system.

2). If the system is modular, the tests can also be modular which results in the efficiently, of the system as well as in the efficiency of testing process.

⑤. small Vs Large:-

1). programming in the small, is what we do for ourselves in the privacy of our own offices.

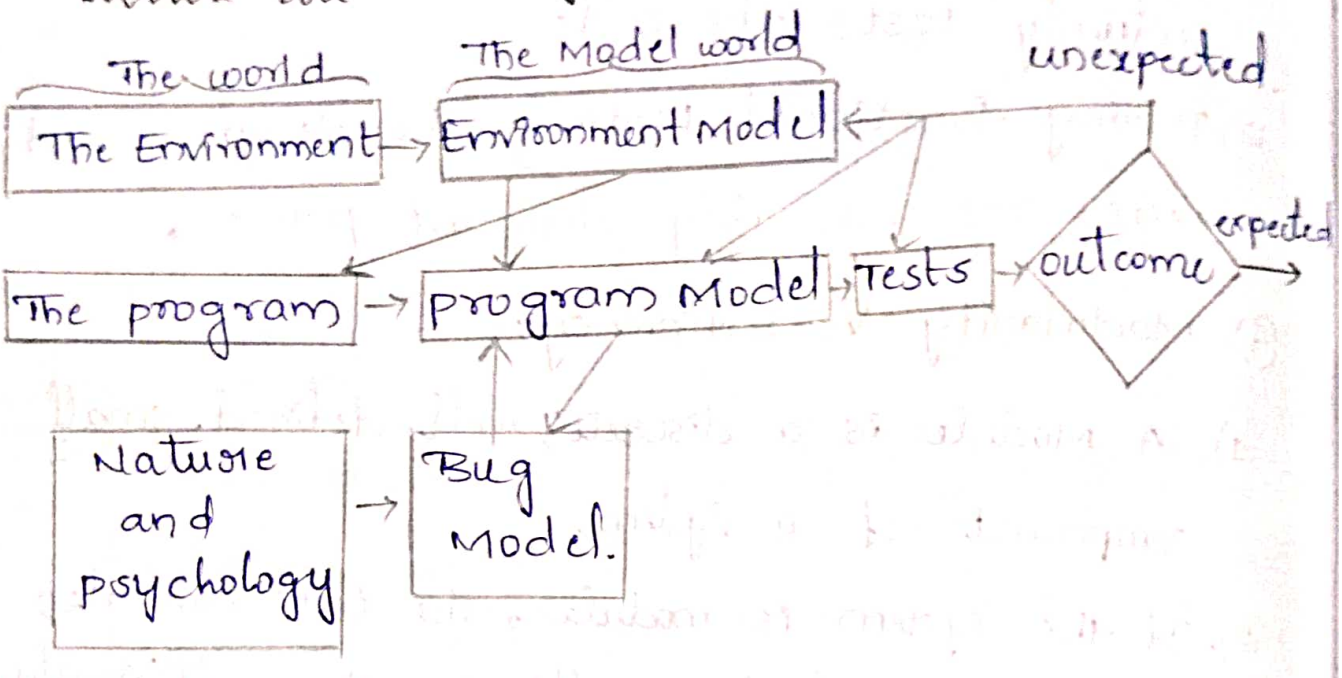
2). programming in the large, means constructing programs that consists of many components written by many different programmers.

⑥. Builder Vs Buyer:-

* Builder:- who designs the system and is accountable to the buyer.

* Buyer: who pays for the system in the hope of profits from providing services.

* Model for Testing:-



* Environments-

1). A programs environment is the hardware and software required to make it run.

2). The environment also includes all programs that interact with and are used to create the program under test such as OS, editors, compiler, etc.

* programs:-

1). Most programs are too complicated to understand in detail

2). The concept of the program is to be simplified in order to test it.

* Bugs:-

1). Bugs are more insidious than ever we expect them to be.

2). An unexpected test result may leads us to change our notion of what a bug is and our model of bugs.

* Tests:-

(1). Unit/component Testing:-

1). A unit is the smallest testable piece of sw that be compiled, assembled, linked, loaded etc..

2). A unit is usually the work of one programmer and consists of several hundred (or) fewer lines of code.

(2). Integration testing:-

1). Integration is the process by which components are aggregated to created larger components

2). Integration testing is testing done to show that even though the components were individually satisfactory.

(3). system testing:-

1). A system is a big component.

2). system Testing is aimed at revealing bugs that cannot be attributed to components.

* Consequences of bugs:-

1) Mild:- The symptoms of the bug offend us gently, a misspelled output (or) a misaligned printout.

2). Moderate:- outputs are misleading (or) redundant the bug impacts the systems performance

3). Annoying:- The systems behaviour, because of the bug is dehumanizing.

4). Disturbing:- It refuses to handle legitimate transactions.

5). Serious: It loses track of its transaction.

6). Very serious: The bug causes the system to do the wrong transactions.

7). Extreme:- The problems are not limited to a few users (or) to few transaction types

8). intolerable:- long term unrecoverable corruption of the database occurs and the corruption is not easily discovered.

9). Catastrophic:- The decision to shut down is taken out of our hands because the system fails.

10). Infectious:- what can be worise than a failed system? one that corrupts other systems even though it does not fail in itself; that

erodes the social physical environment, that melts reactors and starts a war.

* Taxonomy of Bugs:-

1). There is no universally correct way to categorize bugs.

2). The taxonomy is not rigid.

3). A given bug can be put into one (or) another category depending on its history and the programmers state of mind.

4). The major categories are:-

① Requirements Features & functionality bugs

↓

② structural bugs

↓

③ Data bugs

↓

④ coding bugs

↓

⑤. Interface, integration and system bugs.

① Requirements, Features and Functionality bugs:-

1) Requirements and specifications Bugs:- Requirements and specifications developed from them can be incomplete ambiguous, (or) self - contradictory.

2). Feature Bugs:- specification problems usually create corresponding feature problems

3). Feature Interaction Bugs:- providing correct, clear, implementable and testable feature specifications is not enough.

4). specification and Feature Bug Remedies:- Most feature bugs are rooted in human to human communication problems.

2. structural bugs:-

1) control and sequence bugs:- control and sequence bugs include paths left out, unreach -able code, improper nesting loops, duplicated processing, and pachinko code.

2). Logic bugs:-

1) Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and

combinations.

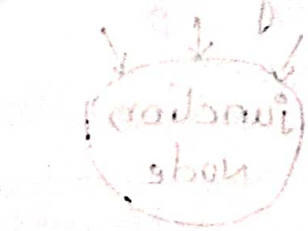
3. Data bugs:- Data bugs include all bugs that arises from the specification of data objects, their formats, the number of such objects, and their initial values.

4. Coding bugs:- coding errors of all kinds can create any of the other kind of bugs.

(5). Interface, integration and system bugs:-

1) External interfaces:- The external interfaces are the means used to communicate with the world.

2). Internal interfaces:- The internal interfaces are in principle not different from external interfaces but they are more controlled.



* Flow graphs and path testing:-

* Basics concepts of path testing:

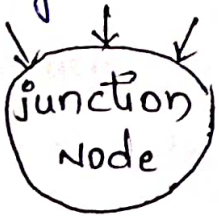
1). path testing is a method that is used to ~~testing~~ design the test cases.

2). To design test causes using this technique, four steps are followed.

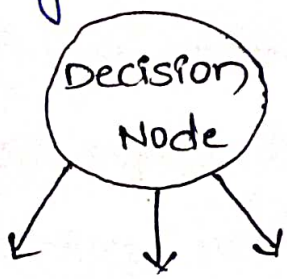
(1). control Flow Graph: A control flow graph is a directed graph which represents the control structure of a program (or) module.

2) A control graph can also have three Nodes. They are:-

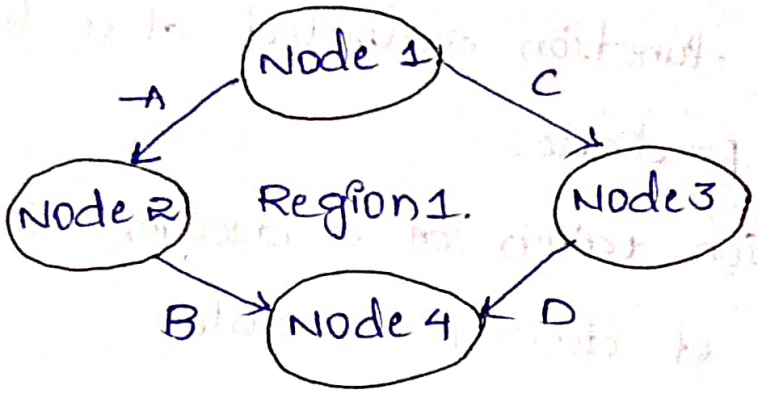
i). Junction Node: A node with more than one arrow entering it.



ii) Decision Node: A node with more than one arrow leaving it.



(iii) Region: Area bounded by edges and nodes.



2. cyclomatic Complexity:- The cyclomatic complexity is said to be a measure of the logical complexity of a program.

3. Independent paths: An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined.

4. Design Test cases from independent paths:- Finally, after obtaining the independent paths, test cases can be designed where each test case represents one (or) more independent paths. "

* predicates:-

1). The logical function evaluated at a decision is called predicate.

2). The direction taken at a decision depends on the value of decision variable.

* Example: A>0, x+y >=90.....

* path predicates:-

1). A predicate associated with a path is called path predicate.

* Example:- "x is greater than zero" is true

AND

"x+y >=90" is false.

AND

"w is either negative (or) equal to 10" is true.

* Multiway Branches:-

The path taken through a multiway branch such as computed GOTO, case statement, (or) jump tables cannot be directly expressed in TRUE/FALSE terms.

* Inputs:- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.

* Achievable paths:-

1). In software testing methodologies, "achievable paths" refer to the different paths (or) scenarios that can be taken during the testing process.

2) It helps identify the possible combinations of inputs and actions to ensure through testing coverage.

* Examples:-

1) successful login with correct username and password.

2) Failed login with incorrect username (or) password.

3) Login with a valid username but an expired password.

4) login with a valid username but an account that has been locked.,

* path sensitizing :-

1). Most of the normal paths are very easy to sensitize - 80% - 95% transaction flow coverage (C_1+C_2) is usually easy to achieve.

2). The remaining small percentage is often very difficult.

3). sensitization is the act of defining the transaction.

4). If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows (or) a design bug.

* path instruementation :-

1). Instrumentation plays a bigger role in transaction flow testing than in unit path testing.

2) The information of the path taken for a given transaction must be kept with that transaction & can be recorded by a central transaction dispatcher (or) by the individual processing modules.

3). In some systems, such traces are provided by the operating systems (or) a running log."

* Applications of path Testing:-

Integration, coverage and paths is called components

↙

New code

↓

Maintenance

↓

Rehosting:

(1). Integration, coverage and paths is called components:-

1) path testing methods are mainly used in unit testing; especially for new software.

2) The new component is first tested as an independent unit with all called components and coreqursite components replaced by stubs.

(2). New code:-

1). New code should always be subjected to

enough path testing to achieve C_2 stubs are used where it is clear that the bug potential for the stub is significantly lower than that of the called components.

2). Typically we will try to use the shortest entry/exit path that will do the task.

3. Maintenance:-

1). There is a great difference between maintanance testing and new code testing.

2). Maintenance testing is a completely different situation, it involves modifications which are accommodated in the system.

4. Rehosting:-

1) We get a very powerful, effective, rehosting process when C_1+C_2 coverage is used in conjunction with automatic (or) semiautomatic structural test generators.

2) software is rehosted because it is no longer cost effective to support the environment in which it runs the objective of rehosting is to change the operating environment.

UNIT-II.

⇒ Transaction Flow Testing:

⇒ Transaction Flows.

⇒ Transaction flow testing techniques.

⇒ Dataflow testing:

⇒ Basics of dataflow testing.

⇒ strategies in dataflow testing.

⇒ Application of dataflow testing.

⇒ Domain Testing:

⇒ Domain and paths

⇒ Nice & ugly domains

⇒ Domain testing.

⇒ Domains and interfaces testing

⇒ Domain and interface testing

⇒ Domain and testability.

* Transaction Flow Testing:

⇒ 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 outside of the system.

⇒. Example:- A transaction for an online information retrieval system might consist of these steps:-

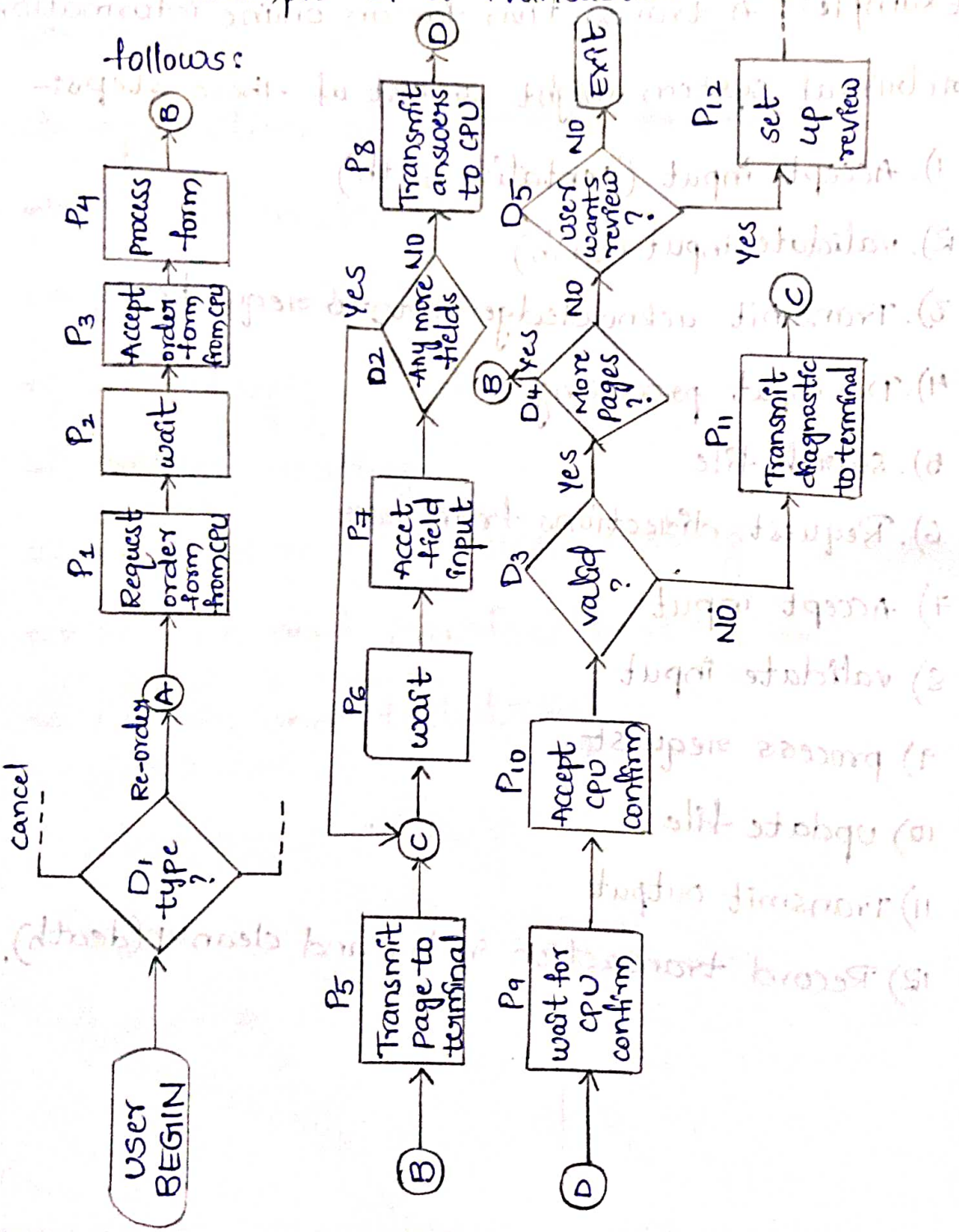
1). Accept input (tentative birth)
2). validate input (birth)
3). Transmit acknowledgement to requester
4). Do input processing.
5). search file
6). Request directions from user
7) Accept input
8) validate input
9) process request
10) update file
11) Transmit output
12) Record transaction in log and cleanup(death).

* Transaction Flow

=> Transaction flows are introduced as a representation of a system's processing.

=> The Transaction flow graph is to create a behavioural model of the program that leads to functional testing.

=> An example of a Transaction Flow is as follows:



→ Usages:-

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

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

3). Loops are infrequent compared to control flowgraphs.

* Transaction flow Testing Techniques:-

1) Get the transactions flows:

⇒ complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.

⇒ Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.

2). Inspections, Reviews and walkthroughs:

1). Transaction flows are natural agenda for system reviews or inspectional.

2). In conducting the walkthrough, you should: discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.

3). path selection:-

1). select a set of covering paths (C_1+C_2) using the analogous criteria you used for structural path testing.

2). select a covering set of paths ~~are~~ based on functionally sensible transactions as you would for control flow graphs.

4). path Sensitization:-

1). Most of the normal paths are very easy to sensitize - 80%-95% transaction flow coverage (C_1+C_2) is usually easy to achieve.

2). The remaining small percentage is often very difficult.

5). path instrumentation:-

1). Instrumentation plays a bigger role in transaction flow testing than in unit path testing.

2). The information of the path taken for a given transaction must be kept with that transaction & can be recorded by a central transaction dispatcher or by the individual processing modules.

6). Design & Maintain Test Database.

1). Design and maintenance of the test databases constitute about 30% to 40% of the effort in transaction flow test design.

2). people are often unaware that a test database needs to designed.

7). Test Execution:

1). commit to automation of test execution, if you want to do transaction flow testing for a system of any size.

2). If the number of test cases run into several hundred, performing transaction flow testing to achieve (C_1+C_2) needs execution automation without which you cannot get it right.

Part-II: Dataflow testing.

* Basics of Dataflow testing:-

* Dataflow testing:- Dataflow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.

* Data flow Machines:- There are mainly two types of data flow Machines. They are:-

1). Von Neumann Machine Architectures:-

The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:

1). Fetch instruction from memory.

2). Fetch operands.

2). Multi-instruction, Multi-data machines (MIMD) Architectures:-

1). These machines can fetch several instructions and objects in parallel.

* Bug Assumption:- The bug Assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, (or) silly things are being done to data objects.

* Data flow graphs:- The data flow graph is a graph consisting of nodes and directed links.

* Data object state and usages:- Data objects can be created, killed and used.

1). Defined (d):- An object is defined explicity when it appears in a data declaration.

2). Killed (K):- An object is killed on undefined when it is released (or) otherwise made unavailable.

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

* Data flow Anomaliess:- There are mainly nine possible Anomalies are:-

1). dd:- probably harmless but suspicious.

2). dk:- probably a bug.

3). du:- the normal case.

4). kd:- normal situation.

5). kk:- harmless but probably buggy.

6). ku:- a bug.

7). ud:- usually not a bug.

8). uk:- normal situation.

9). uu:- normal situation.

* Data flow Anomaly state graph:-

Data flow anomaly state graph model prescribes that an object can be in one of four distinct states.

1). K:- undefined, previously killed, doesnot exist.

2). D:- defined but not yet used for anything.

3). U:- has been used for computation (or) in predicate

4). A:- anomalous.

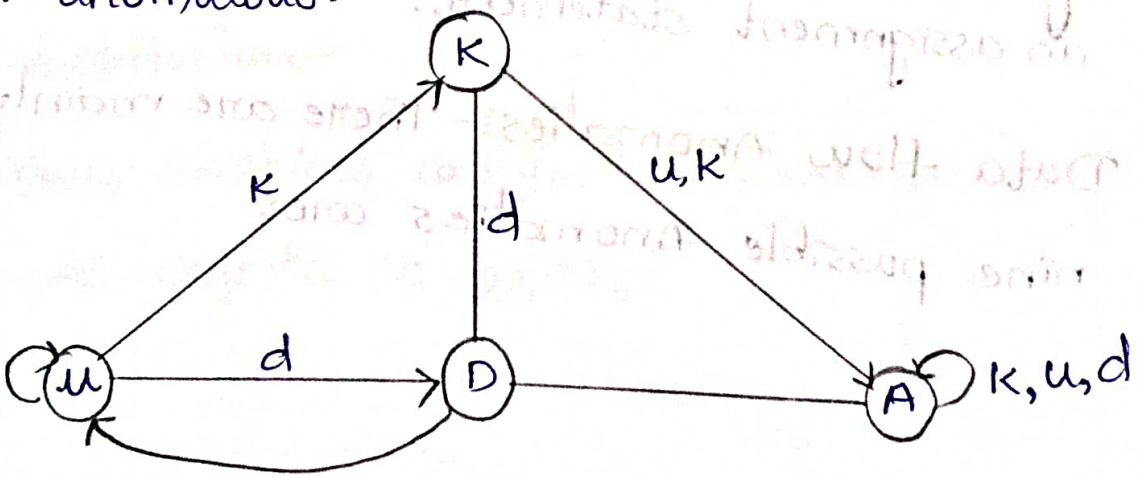
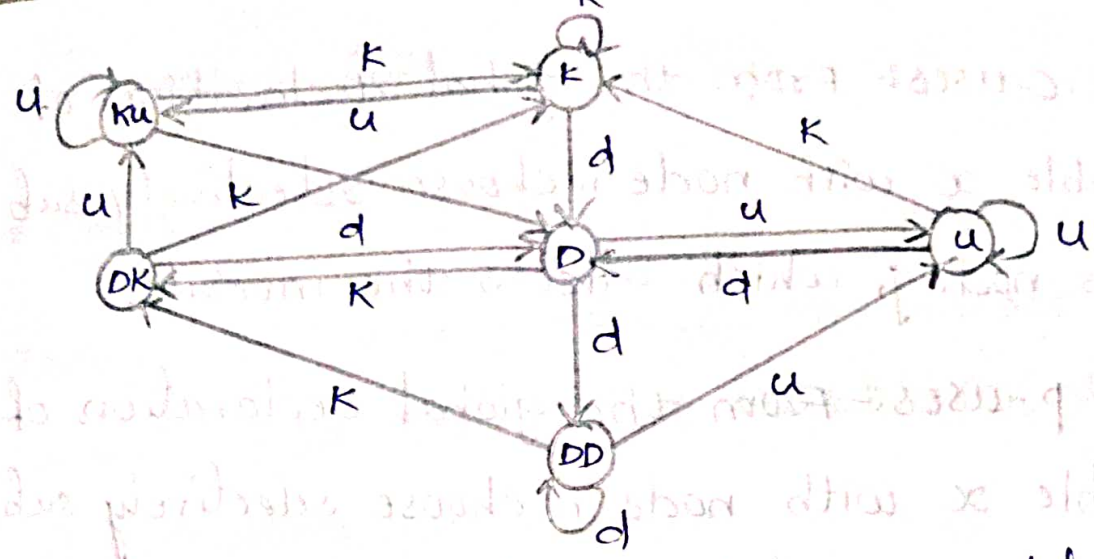


Fig: unforgiving Data Flow Anomaly state graph



* Data flow Model:- The data flow model is based on the program's control flow graph- Don't confuse that with the program's data flow graph.

* strategies in dataflow testing:- The following are the some strategies in dataflow testing.

1). All- defs:- All the sub-paths that include c-use and p-use, suppose there is a variable x and node i to select all edges.

2). All c-uses:- There is global node i to all the sub-paths to node j, select all the sub-paths which are c-use.

3). All p-uses:- There is global node i to all the sub-paths to node j, select all the sub-paths which directly use variable x.

4). Some C-uses:- From the global declaration of variable x with node i, choose selectively sub-paths node j, which affects the module.

5). Some p-uses:- From the global declaration of variable x with node i. choose selectively sub-pathsnode j directly affecting the value of a variable.

* Applications of dataflow testing:-

1). Noticing the change and updation of a variable make the module more bug-free than merely doing branch testing.

2). Debuggers are an excellent tool to track data changes in a variable.

3). Data flow testing tools are integrable into the compilers helping in unit testing.

Part-III: Domain testing.

* Domains and paths:-

Domain:- In mathematics, domain is a set of possible values of an independent variable (or) the variables of a function.

The Model:- The following figure is a schematic representation of domain testing.

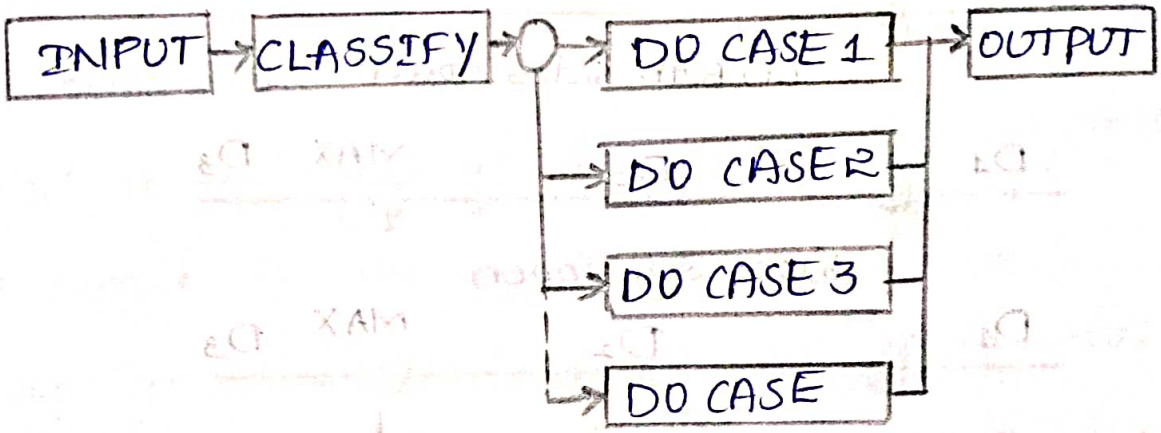


Fig: Domain testing.

* A Domain is a set:- An input domain is a set.

* Domains, paths and predicates:-

1). For every domain, there is atleast one path through the routine.

2). In domain testing, predicates are assumed to be interpreted in terms of input vector variables.

* A Domain closure:-

1). A domain boundary is closed with respect to a domain if the points on the boundary belong to the domain.

2). If the boundary points belong to some other domain, the boundary is said to be open.

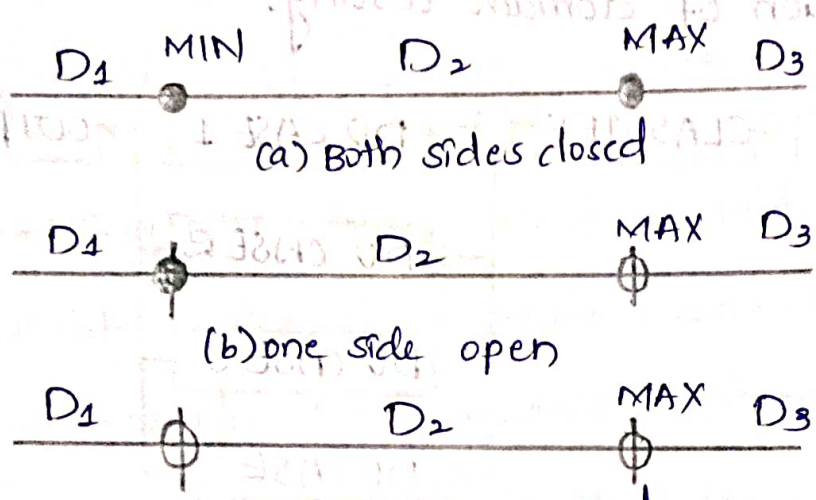


Fig: open and closed Domains

* Domain dimensionality:- Every input variable adds one dimension to the domain.

* Bug Assumptions:- The bug assumption for the domain testing is that processing is okay but the domain definition is corong.

* Domain Errors:-

1). Double zero Representation:- In computers (or) Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.

2). Boundary Errors:- Errors caused in and around the boundary of a domain.

* Restrictions to domain testing:-

1). Co-incidental correctness:- Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons.

2). Loop free software:- Loops are problematic for domain testing, the trouble with loops is that each iteration can result in a different predicate expression, which means a possible domain boundary change.

* Nice and ugly domains:-

* Nice Domains:- some important properties of nice domains are: Linear, complete, systematic, orthogonal, consistently closed, convex and simply connected.

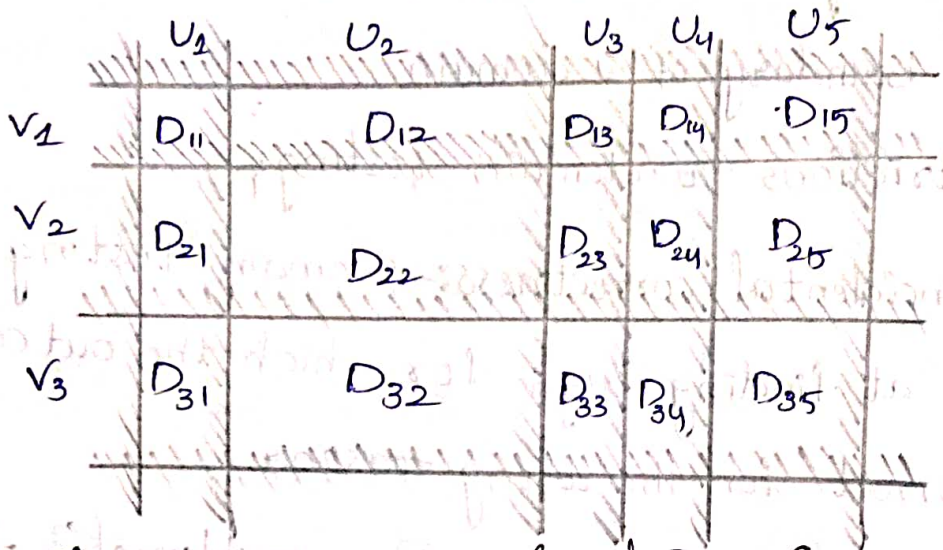


Fig: Nice Two-Dimensional Domains

* Linear and non linear boundaries:- Nice domain boundaries are defined by linear inequalities (or) equations.

* Complete Boundaries:- Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.

* systematic boundaries:- systematic boundary means that boundary in equalities related by a simple function such as a constant.

* orthogonal Boundaries:- Two boundary sets U and V are said to be orthogonal if every inequality in V is perpendicular to every inequality in U.

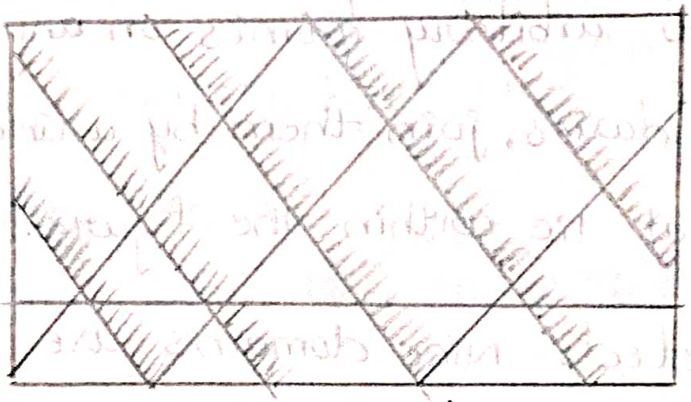


Fig: Tilted Boundaries.

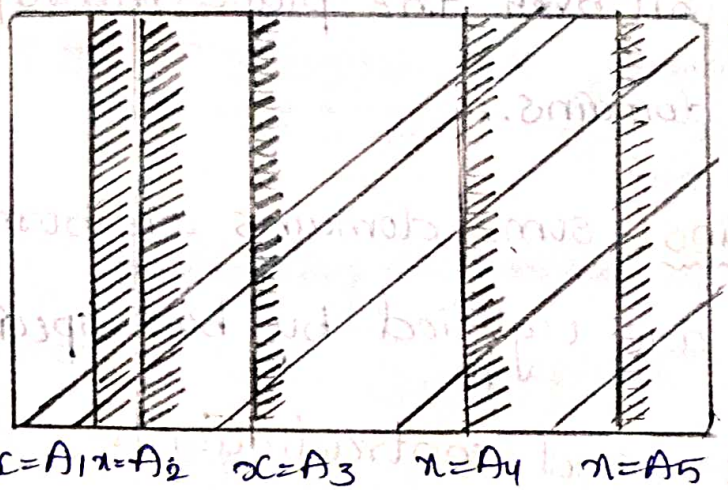


Fig: Linear, Non-orthogonal Domain Boundaries.

* closure consistency:- consistent closure means that there is a simple pattern to the closures for example, using the same relational operator for all boundaries of a set of parallel boundaries.

* convex:- A geometric figure is convex if you can take two arbitary points on any two different boundaries, join them by a line and all points on that lie within the figure.

* simply connected:- Nice domains are simply connected; that is, they are in one piece rather than pieces all over the place interspersed with other domains.,

* Ugly Domains:- some domains are born ugly and some are uglified by bad specifications.

* Ambiguities and contradictions:-

1). Domain ambiguities are holes in the input space.

2). The holes may lie with in the domains (or) in cracks between domains.

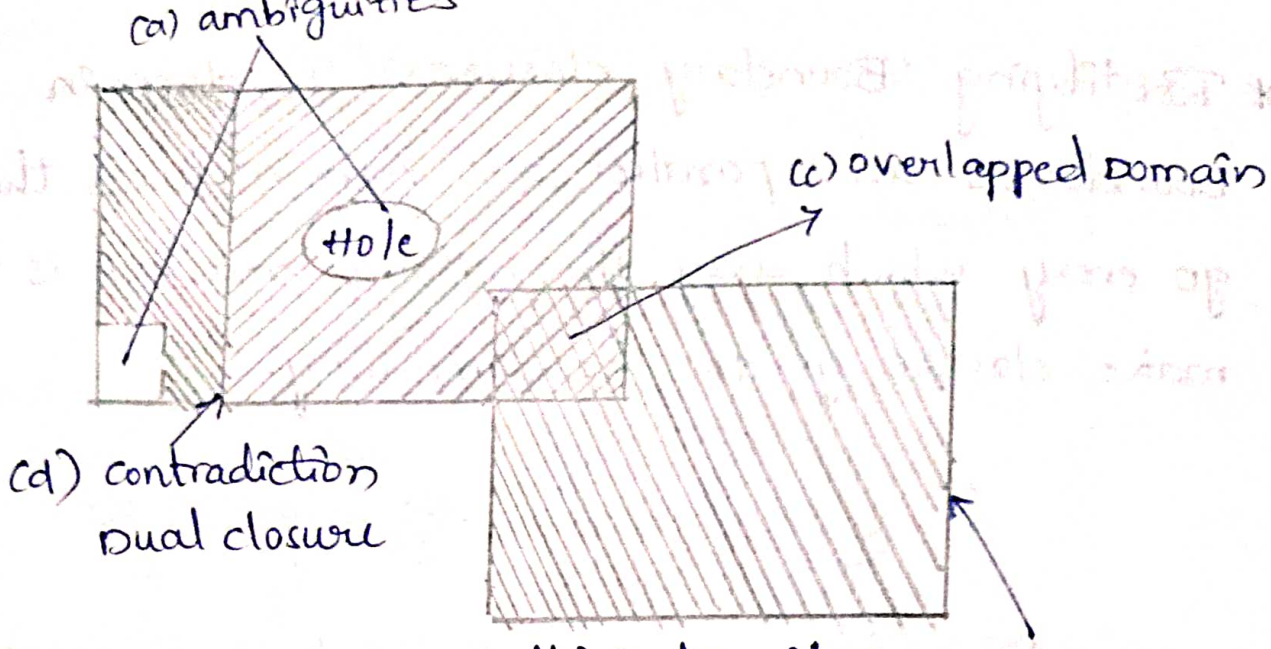
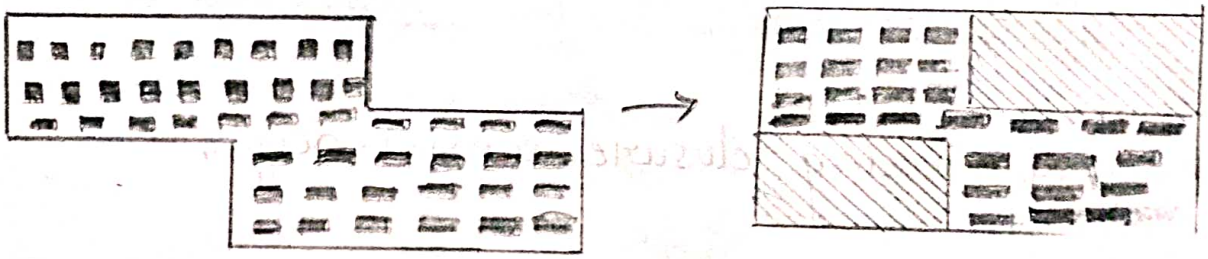
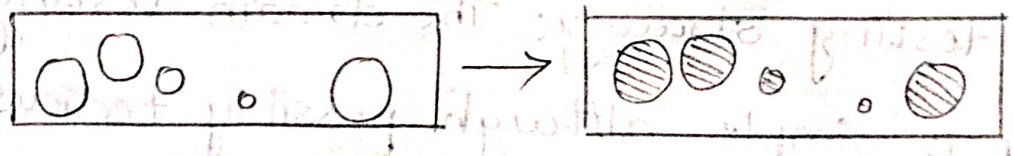


Fig: Domain Ambiguities and contradictions.

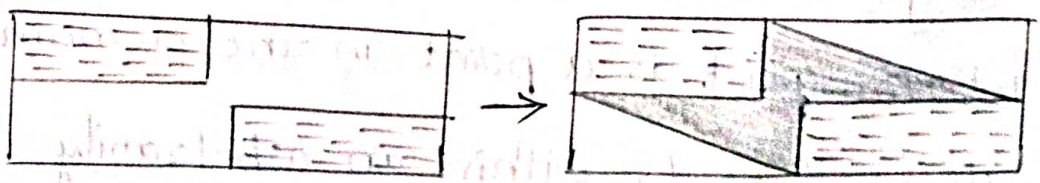
* simplifying the topology:- The programmer's and tester's reaction to complex domain is the same - simplify.



(a) Making it convex



(b) Filling in the Holes.



* Rectifying Boundary closures:- If domain boundaries are parallel but have closures that go every which way the natural reaction is to make closure go to the same way.

(a) consistent Direction.

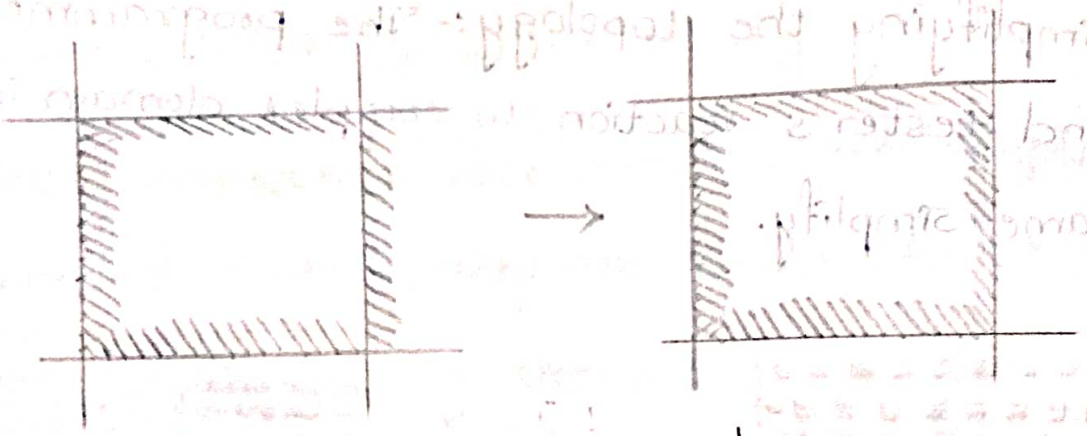


Fig: Forcing closure consistency.,

* Domain Testing:-

* Domain testing strategy:- The domain-testing strategy is simple, although possibly tedious.

* Domain bugs and how to test them:-

1). An interior point is a point in the domain such that all points within an arbitrarily distance are also in the domain.

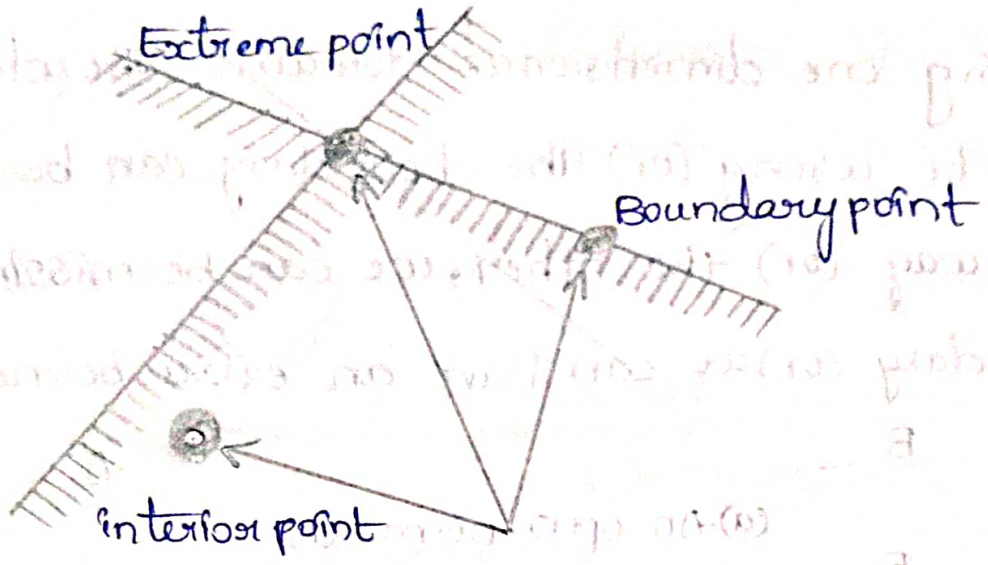


Fig: Interior, Boundary and Extreme points.

2). An onpoint is a point on the boundary, an off point is a point near the boundary but in the adjacent domain.

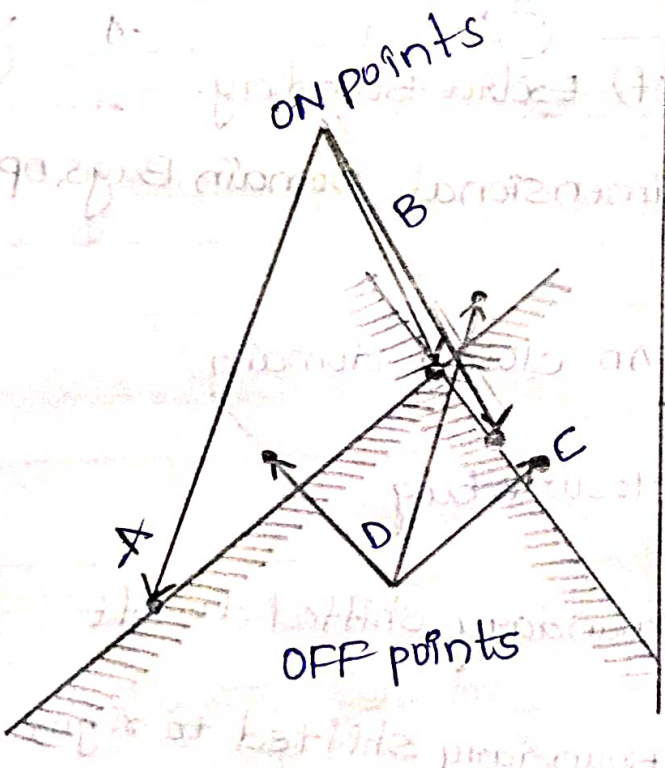


Fig: on points and off points.

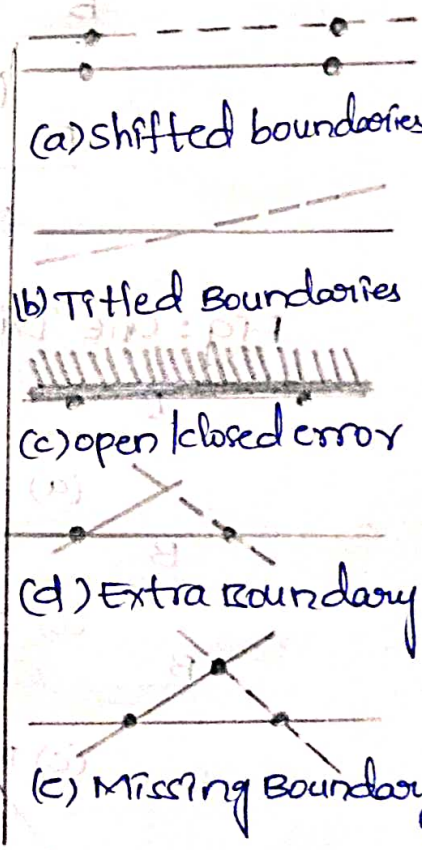


Fig: Generic Domain Bugs.

* Testing one dimensional domain:- The closure can be wrong (or) the boundary can be shifted one way (or) the other, we can be missing a boundary (or) we can have an extra boundary.

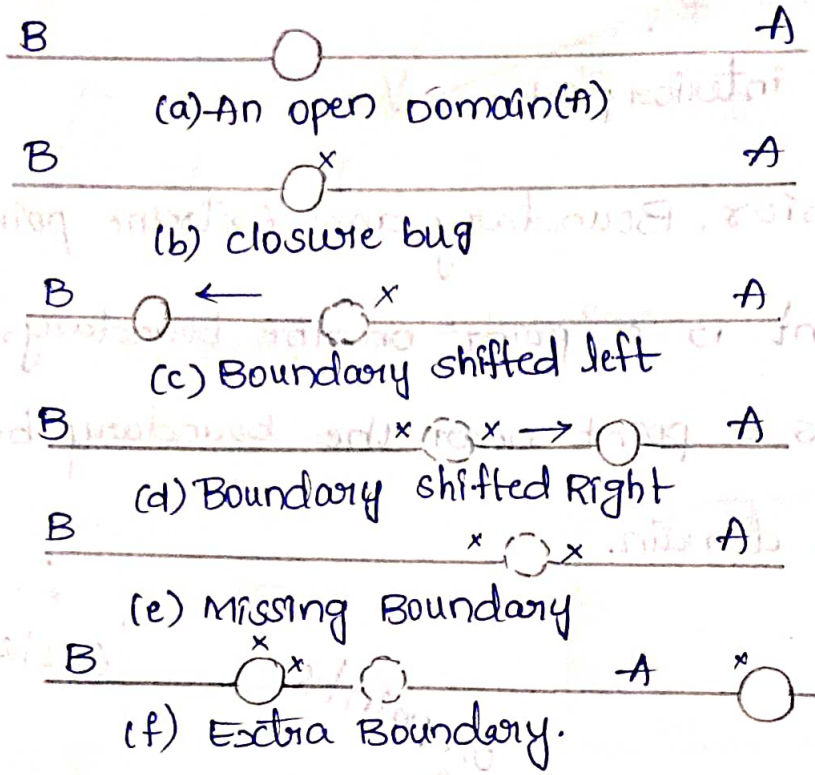


Fig: One Dimensional Domain Bugs, open Boundaries

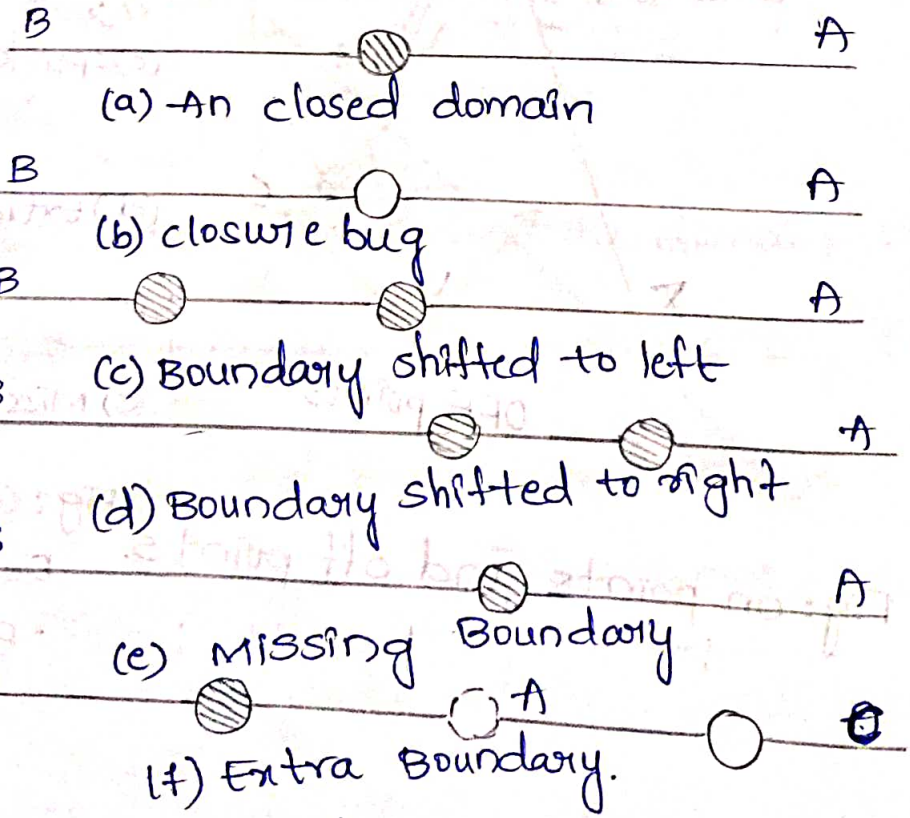
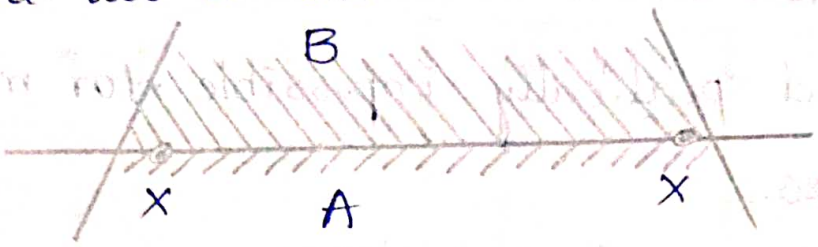
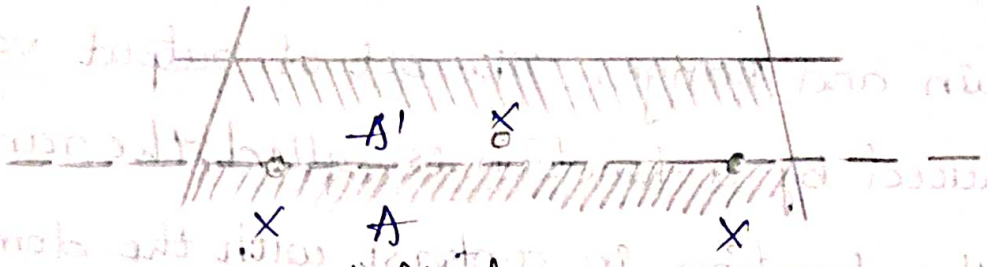


Fig: One dimensional Domain bugs, closed Boundaries.

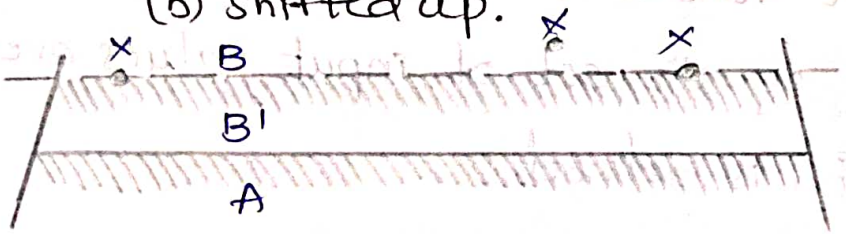
* Testing two dimensional domains:- The below figure shows possible domain boundary bugs for a two-dimensional domain.



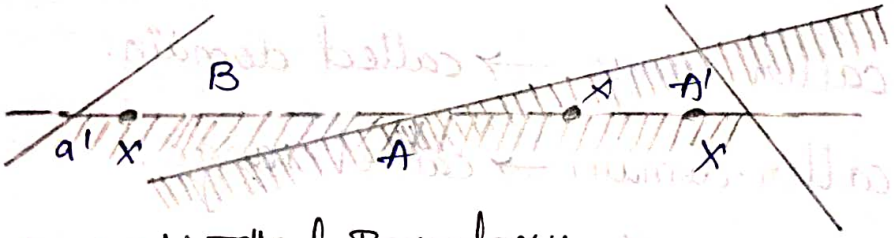
(a) closure Bug.



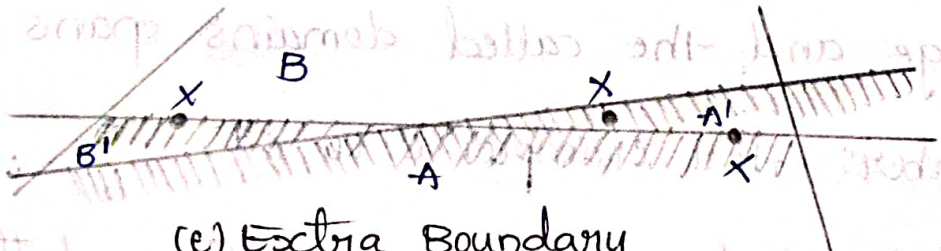
(b) shifted up.



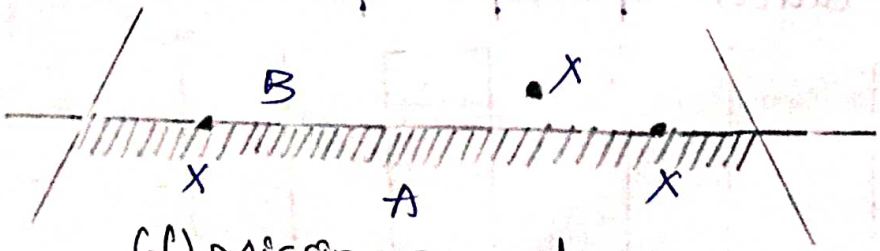
(c) shifted down.



(d) Tilted Boundary



(e) Extra Boundary



(f) Missing Boundary.

* procedure for testing:- The procedure is conceptually is straight-foward, It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.

* Domain and interface testing:-

* Domain and Range:- The set of output values produced by a function is called the range of the function, in contrast with the domain, which is the set of input values over which the function is defined.

caller domain → caller range

caller range → called domain.

caller domain → called range.

* closure compatibility:- 1). Assume that the caller's range and the called domains spans the same numbers - for example, 0 to 17.

caller called opentops openbottoms bothopen.

Fig: Range/Domain closure compatibility.

2). The below figure shows the twelve different ways the caller and the called can disagree about closure.

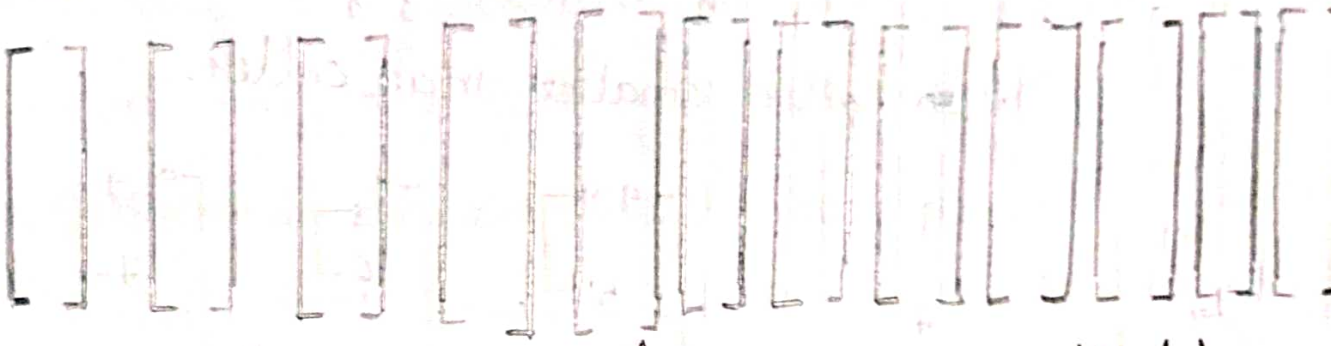


Fig: Equal-span Range/Domain compatibility Bugs.

* span compatibility:- 1). The below figure shows three possibly harmless span incompatibilities.

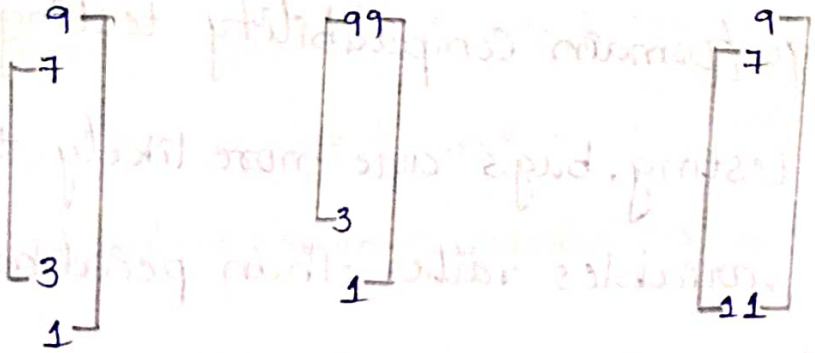
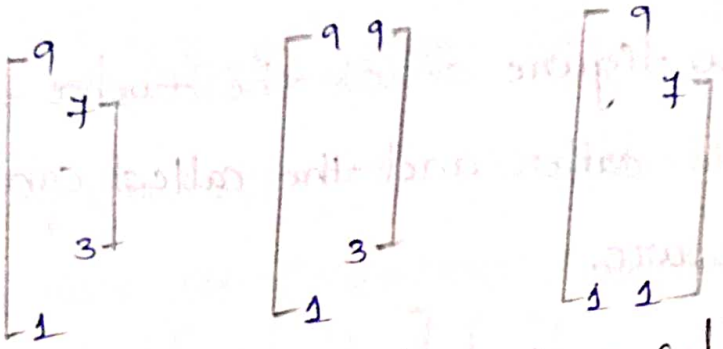
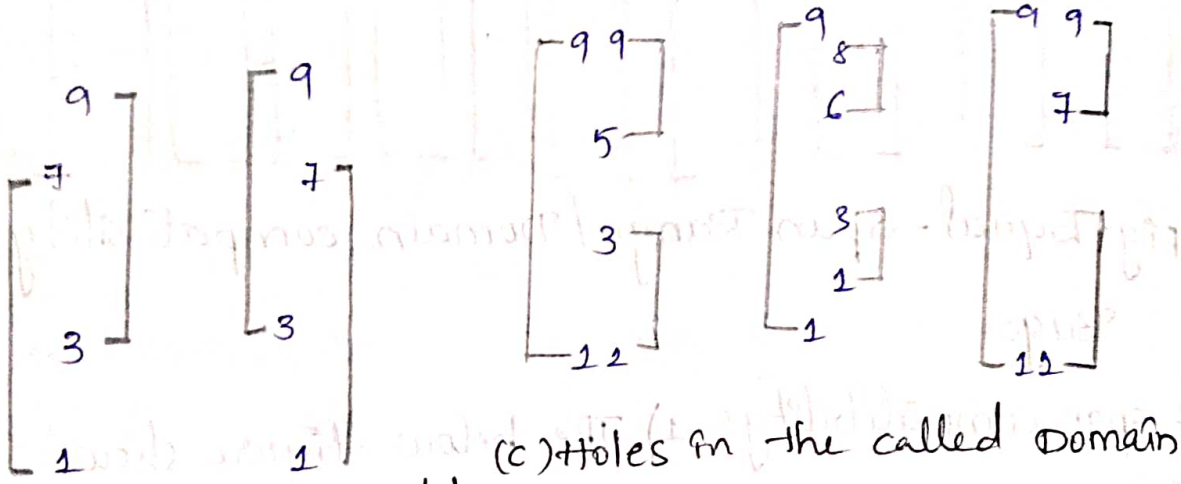


Fig: Harmless Range/Domain span incompatibility bug.

2). The below figure shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects.



(a) called smaller than caller.



(b) Domain Range Mismatch (c) Holes in the called Domain

Fig: Buggy Range / Domain Mismatches.

* Interface Range / Domain Compatibility testing :-

1). For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two (or) more variables.

2). Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable.

* Domains and testability:-

1). The best approach to do domain is testing is to stay away from it by making things simple such that the testing is not required.

2). Orthogonal domain boundaries, consistent closure independent boundaries, linear boundaries, etc.. make the domain testing easier.

3). Non-linear boundaries can be converted into equivalent linear boundaries.

4). Nice boundaries come in parallel sets.

5). Testing can be divided into several steps that can be merged and made small that can be converted into a canonical program form..

UNIT - III.

Part-I: path products and Regular Expressions.

⇒ path products and path expression.

⇒ Reduction procedure.

⇒ Applications

⇒ Regular expressions and flow anomaly detection.

Part-II: Logic Based Testing

⇒ overview

⇒ decision tables.

⇒ path expressions.

⇒ kv charts.

⇒ specifications.

Part-I: path products and Regular Expressions

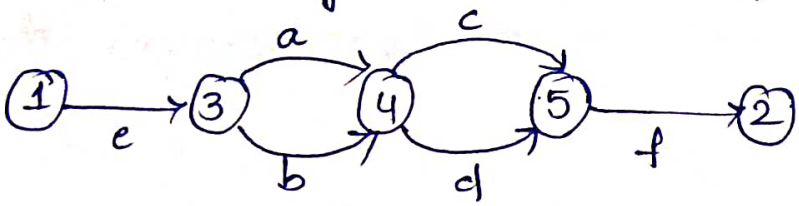
* **path products:-** The name of a path that consists of two successive path segments is conveniently expressed by the contatenation (or) path product of the segment names.

* **Examples:**

1). If x and y are defined as x = abcde, y = fghij, then path corresponding to x followed by y is denoted by,

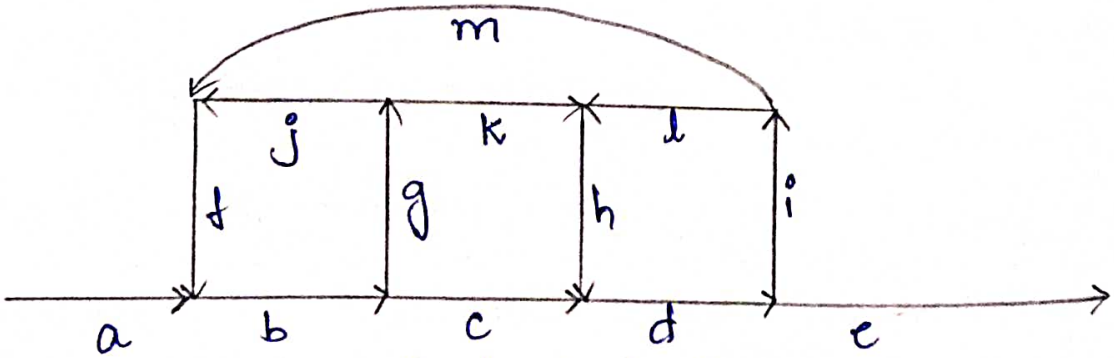
xy = abcdefghij.

2). The below diagram shows the path products.



eacf, eadf, ebcf, ebdf

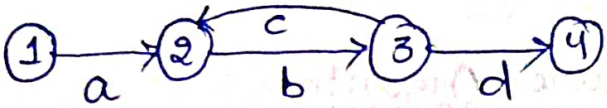
(a)



abcde, abgjfbcde, abcdimfbcde (b).

ac, abc, abbc, abbbc, abbbbc.

(c).



abd, abcbd, abcbcbd, abcbcbcbd.

(d). "

* path Expression:-

1). consider a pair of nodes in a graph and the set of paths between those node.

2). Denote that set of paths by upper case letter such as X, Y.

3). From the above figure the members of the path set can be listed as follows:-

ac, abc, abbc, abbbc, abbbbc.....

4). Alternatively, the same set of paths can be denoted by:

ac + abc + abbc + abbbc + abbbbc + ...

5). The + sign is understood to mean "or" between the two nodes of interest, Paths ac (or) abc, (or) abbc, and so on can be taken.

6). Any Expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "path Expression".

* Reduction procedure Algorithm:-

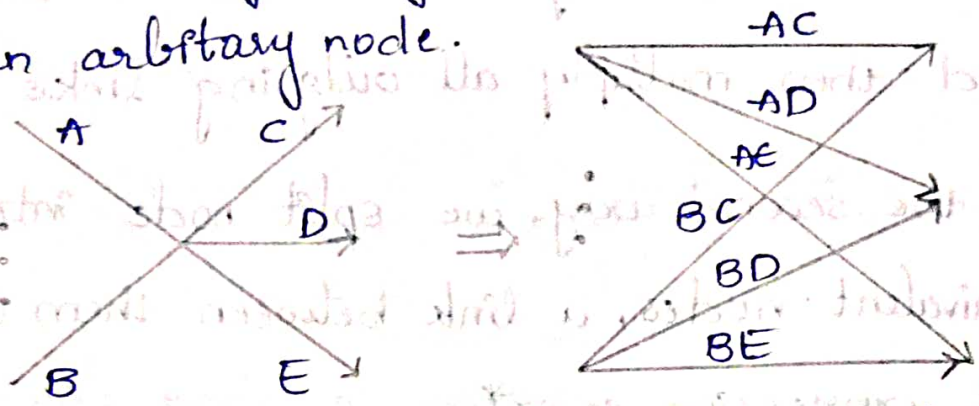
* step-1:- combine all serial links by multiplying their path expressions.
* step-2:- combine all parallel links by adding their path expressions.
* step-3:- Remove all self-loops by replacing them with a link of the form x*, where x is the path expression of the link in the loop.
* step-4 (CROSS-TERM STEP):-

1). The cross-term step is the fundamental step of the reduction algorithm.

2). It removes a node, thereby reducing the number of nodes by one.

3). successive applications of this step eventually get you down to one entry and one exit node.

4). The following diagram shows the situation at an arbitary node.

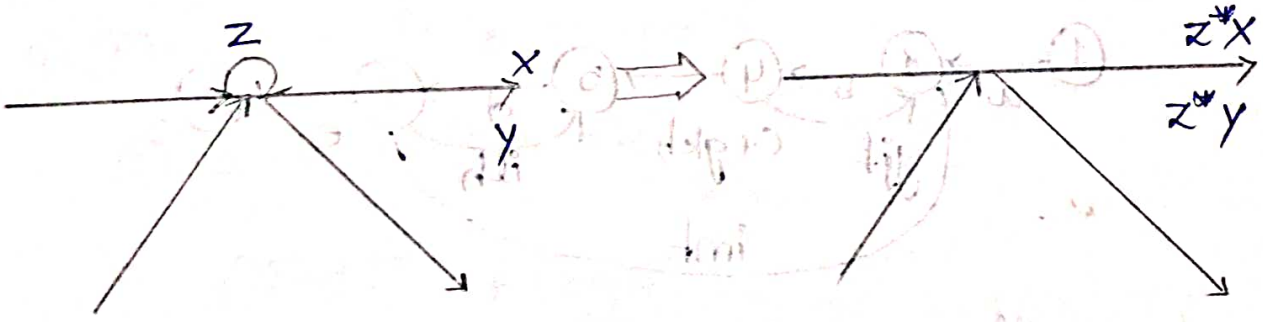


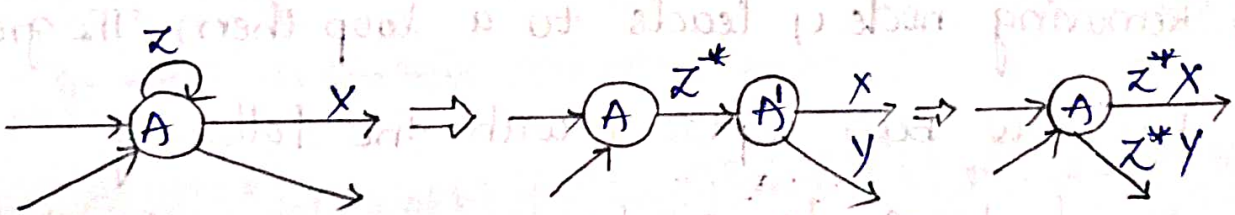
5). From the above diagram, one can infer:

$$(a+b)(c+d+e) = ac+ad+ae+bc+bd+be.$$

* step (Loop Removal operations):-

1). There are two ways of looking at the loop-removal operation:





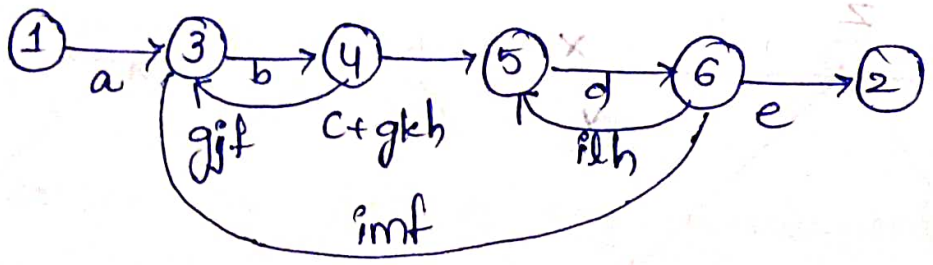
2). In the first way, we remove the self-loop and then multiply all outgoing links by z^*.

3). In the second way, we split node into two equivalent nodes, a link between them whose path expression is z^*.

* step-6 (parallel Term):-

1). Removal of node 3 above led to a pair of parallel links between nodes 4 and 5.

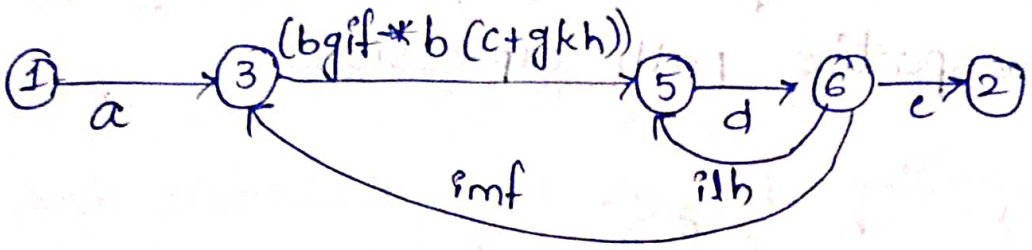
2). combine them to create a path expression for an equivalent link whose path expression is.



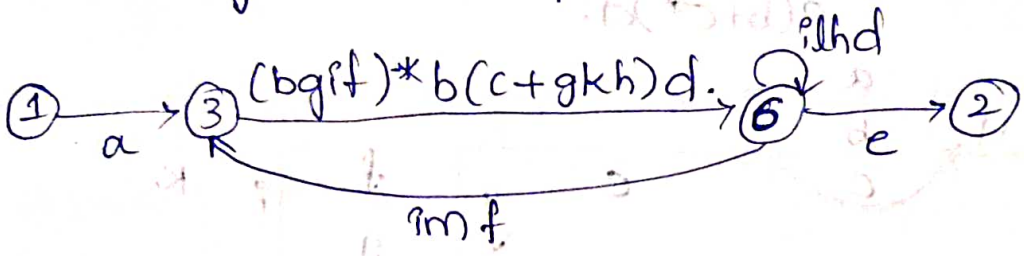
* step-7 (loop Term):-

1). Removing node 4 leads to a loop them. The graph has now been replaced with the following equivalent simpler graph:

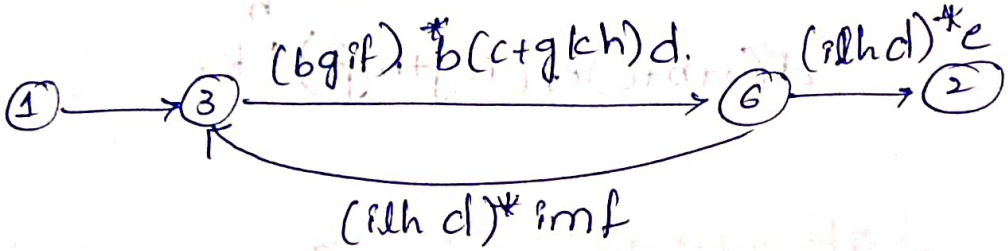
2). continue the process by applying the loop-removal step as follows:



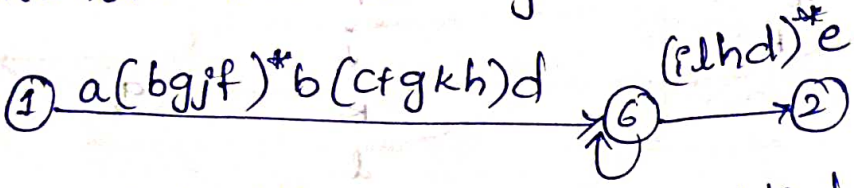
3). Removing node 5 produces:-



4). Remove the loop at node 6 to yield:-



5). Remove node 3 to yield.

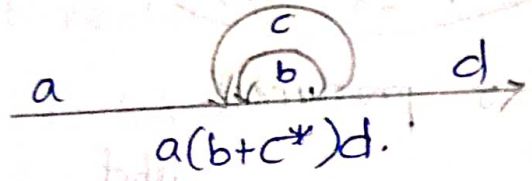


(ilhd)*imf (bgif)*b(c+gkh)d.

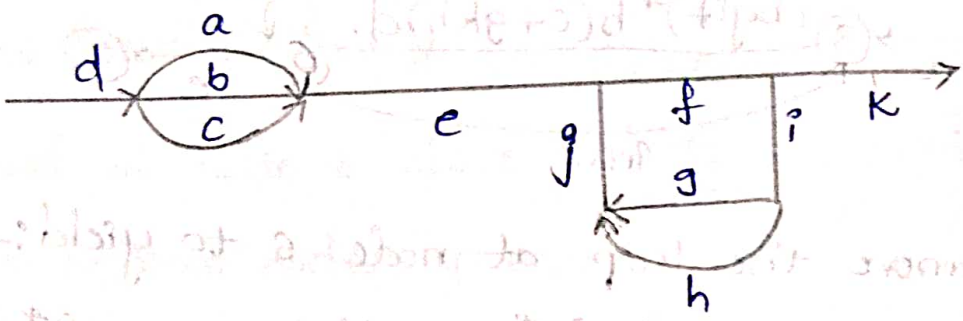
6). Removing the loop and then node 6 results in the following expression:-

a(bgif)*b(c+gkh)d (ilhd)*imf (bjgf)*b(c+gkh)d)* (ilhd)*e.

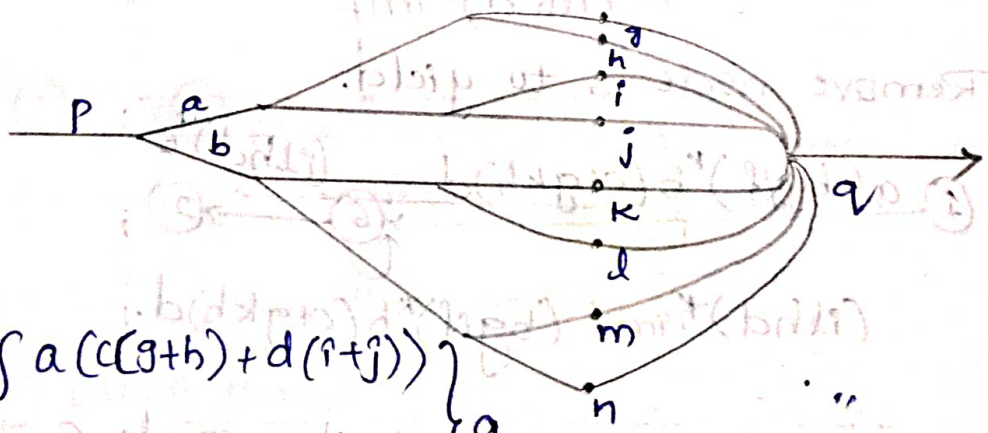
8). You can pratice by applying the algorithm on the following flowgraphs and generate their respective path expressions:



$a(b+c^*)d$.



$d(a+b+c)ef(i(g+h)jf)^*k$.



$$P\begin{cases} a(c(g+h)+d(i+j)) \\ b(e(k+l)+f(m+n)) \end{cases} q$$

* Applications:-

1). The purpose of the node removal algorithm is to present one very generalized concept the path expression and way of getting it.

2). Every application follows this common pattern:

i). convert the program (or) graph into a path expression.

ii). Identify a property of interest and device an appropriate set of "arithmetic" rules that characterizes the property.

* Maximum path count Arithmetic:-

case	Path expression	weight expression
parallels	A+B	$w_A + w_B$
series	AB	$w_A w_B$
Loop	A^2	$\sum_{j=0}^{n} w_A^j$

* Lower path count Arithmetic:-

case	Path expression	weight expression
parallels	A+B	$w_A + w_B$
series	AB	$\max(w_A, w_B)$
loop	A^n	$1, w_1$

* Regular Expressions and flow anomaly Detection:-

* The problem:-

1). The generic flow-anomaly detection problem is that of looking for a specific sequence of options considering all possible paths through a routine.

2). let the operations be SET and RESET, denoted by s and r respectively, we want to know if there is a SET followed immediately a SET (or) a RESET followed immediately by a RESET.

* The Method:-

1). Annotate each link in the graph with the appropriate operator (or) the null operator 1.

2). simplify things to the extent possible, using the fact that $a+a=a$ and $1^2=1$.

3). you now have a regular expression that denotes all the possible sequences of operators in that graph.

4. you can now examine that regular expression for the sequences of interest.

* Limitations:-

1). Huang's theorem can be easily generalized to cover sequences of greater length than two characters.

2). There are some nice theorems for finding sequences that occur at the beginnings, and ends of strings but no nice algorithms for finding strings buried in an expression.

Part-II: Logic Based Testing

* Overviews-

1). "Logic" is one of the most often used words in programmer's vocabularies but one of their least used techniques.

2). Logic has been, for several decades, the primary tool of hardware logic designers.

* Knowledge Based systems-

1). The knowledge-based system has become the programming construct of choice for many applications that were once considered very difficult.

2). Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law (or) Civil engineering into a database.

* Decision tables:-

1). Decision tables are used in various engineering fields to represent complex logical relationships.

2). This testing is a very effective tool in testing the software and its requirements management

* parts of Decision Tables:-

In software testing, the decision table has 4 parts which are divided into portions and are given below:-

	stubs	Entries.
condition	c_1 c_2 c_3	
Action	a_1 a_2 a_3 a_4	

1). condition stubs:-

The condition are listed in this first upper left part of the decision table that is used to determine a particular action (or) set of actions.

2). Action stubs:- All the possible actions are given in the lower left portion of the decision table.

3). condition Entries:- In the condition entry the values are inputted in the upper right portion of the decision table.

4). Action Entries:- In the action entry, every entry has some associated action (or) set of actions in the lower right portion of the decision table and these values are called outputs.

* Types of Decision Tables:-

1). Limited Entry:- In the limited entry decision tables, the condition entries are restricted to binary values.

2). Extended Entry:- In the extended entry decision table, the condition entries have more than two values.

* path Expressions:-

* General:-

1). Logic-based testing is structural testing when it's applied to structure it's functional testing when it's applied to a specification.

2). In logic-based testing we focus on the truth values of control flow predicates.

* Boolean Algebra:-

* steps:-

1). Label each decision with an Uppercase letter that represents the truth value of the predicate.

2). The YES (or) TRUE branch is labelled with a letter and the NO (or) FALSE branch with the same letter overscored.

3). The truth value of a path is the product of the individual labels.

4). concatenation (or) products mean "AND".

* KV charts:-

* Definition:- KV (karnaugh-veitch) charts are used to determine which cases are interesting and which combination ~~are interesting~~ and which comb of predicate values should be used to reach which node.

* single variables:-

	A=0	A=1
0	0	0

The function is never true.

	A=0	A=1
A	0	1

The function is true when A is true.

	A=0	A=1
$\bar{A}$	1	0

The function is true when A is false

	A=0	A=1
1	1	1

The function is always true

1). The above image shows all the boolean functions of a single varrable and their and their equivalent representation as, KV chart.

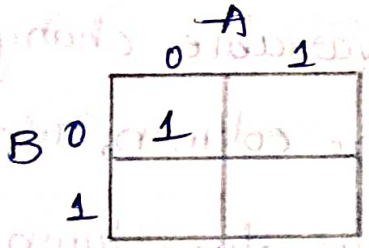
2). The chart show all possible truth values that the variable A can have A "1" means the variables value is "1" (or) TRUE.

3). A "0" means that variables value is 0 (or) FALSE.

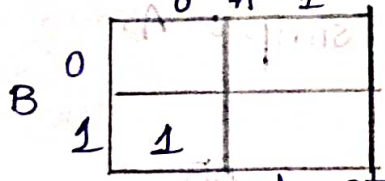
4). The entry in the box (0 or 1) specifies is true (or) false for the value of the variable.

5). Do not explicitly put in 0 entries but specify only the conditions under which the function is true.

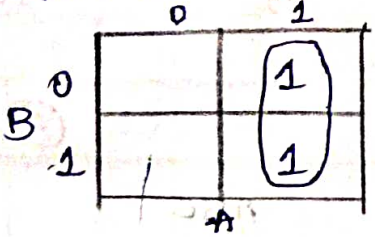
* Two variables:-

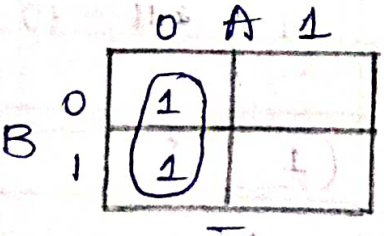


$\bar{A}B$ – NAND

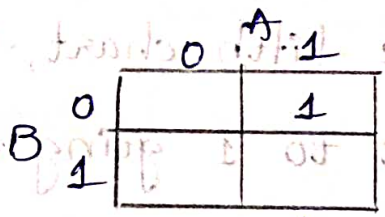


$\bar{A}B$ – B and not A

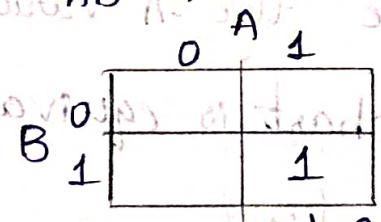




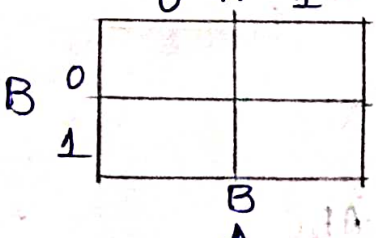
$\bar{A}$

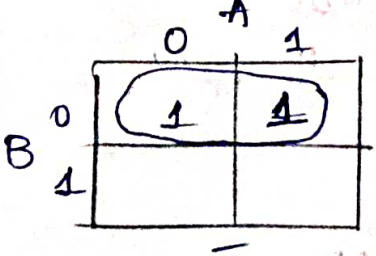


$\bar{A}B$ – A and not B



$A\bar{B}$ – B and A





$\bar{B}$

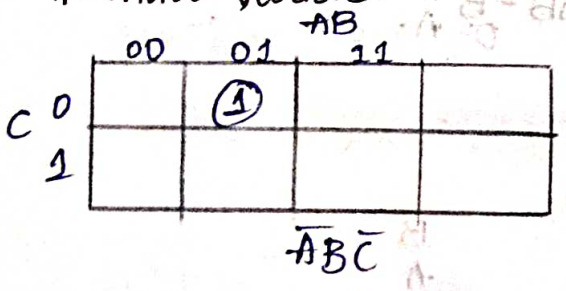
1). The above image shows eight of the sixteen possible functions of two variables.

2). Each box corresponds to the combination of values of the variables for the row and column of that box. A pair may be adjacent either horizontally (or) vertically but not diagonally.

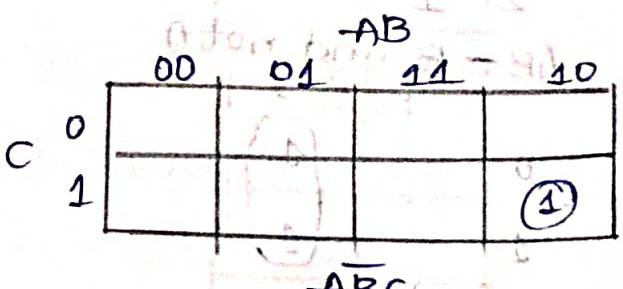
3). Any variable that changes in either the horizontal (or) vertical direction does not appear in the expression.

4). In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variables value for column is 1, the chart is equivalent to simple A.

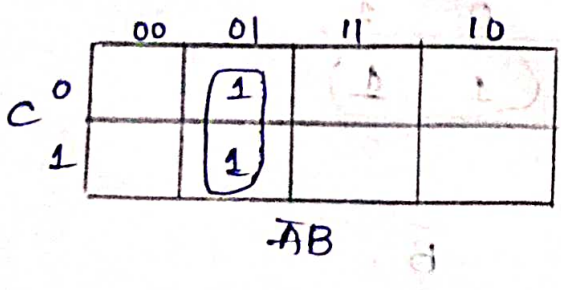
* Three variables:-



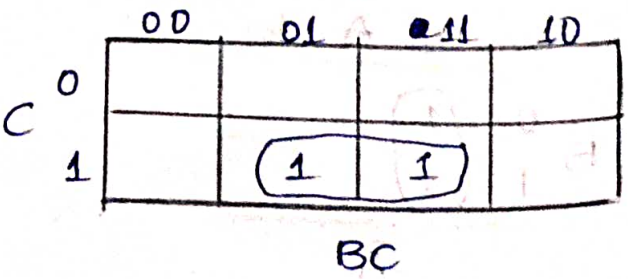
$\bar{A}B\bar{C}$



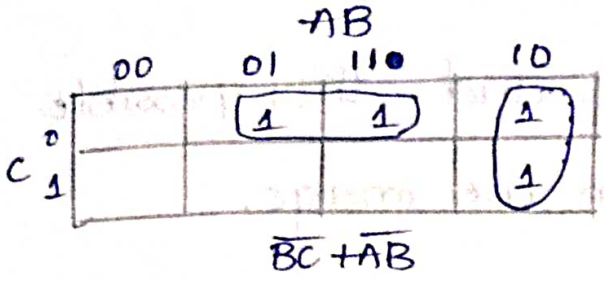
$A\bar{B}C$

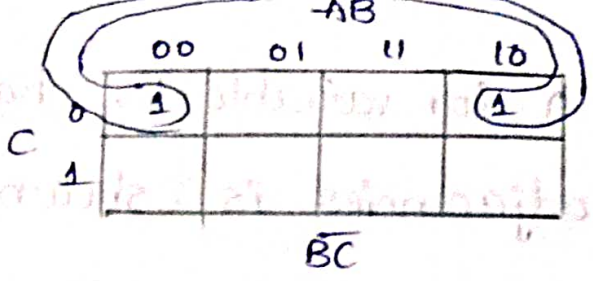


$\bar{A}B$



BC





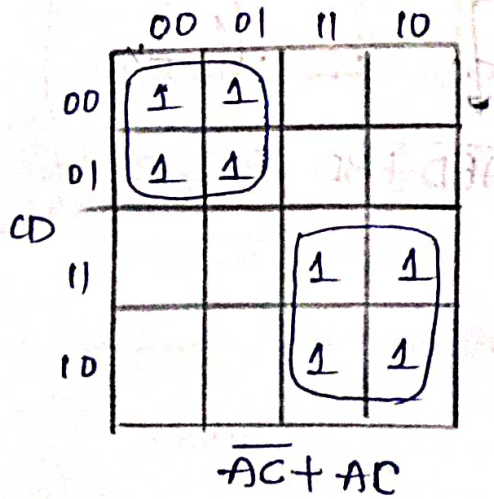
1). The above images shows KV charts for three variables.

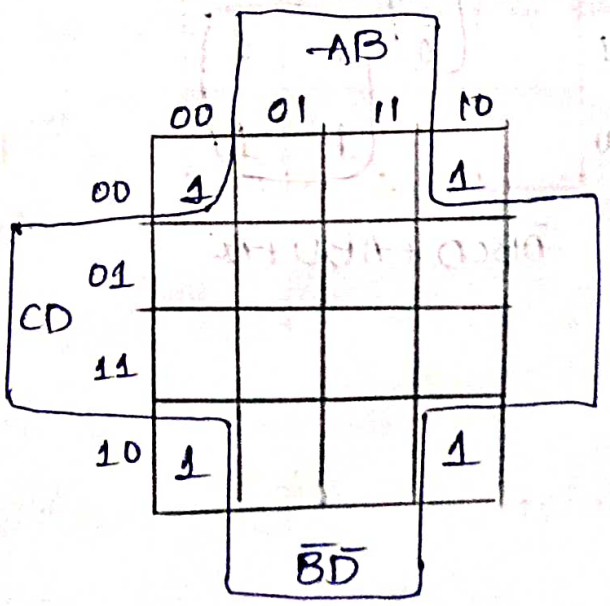
2) As, before, each box represents an elementary term of three variables with a bar appearing according to whether the row-column heading for that box is 0 (or) 1.

3). A three variable chart can have grouping of 1, 2, 4, and 8 boxes.

4). A few examples will illustrate the principles Notice that there are several ways to circle the boxes into maximum sized covering groups.

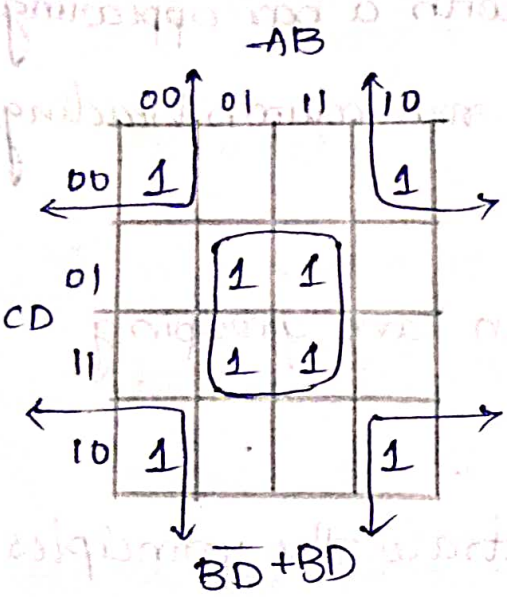
* four variables-

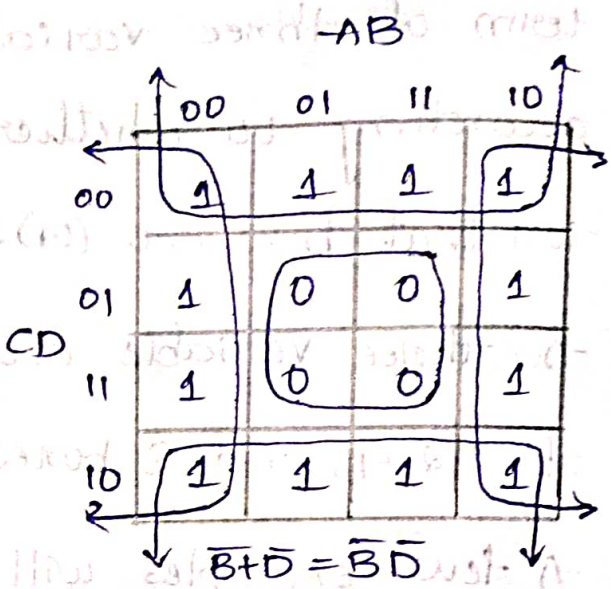


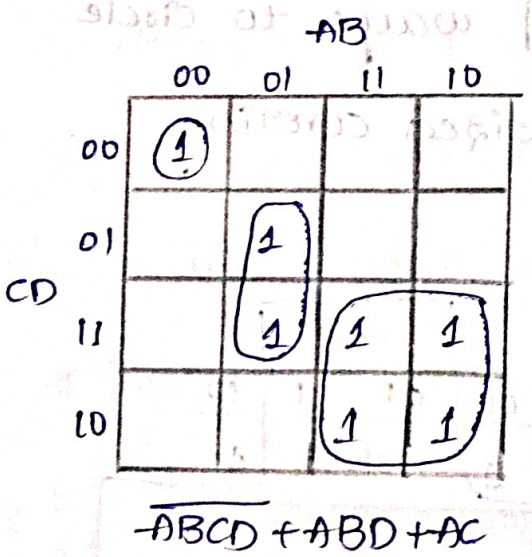


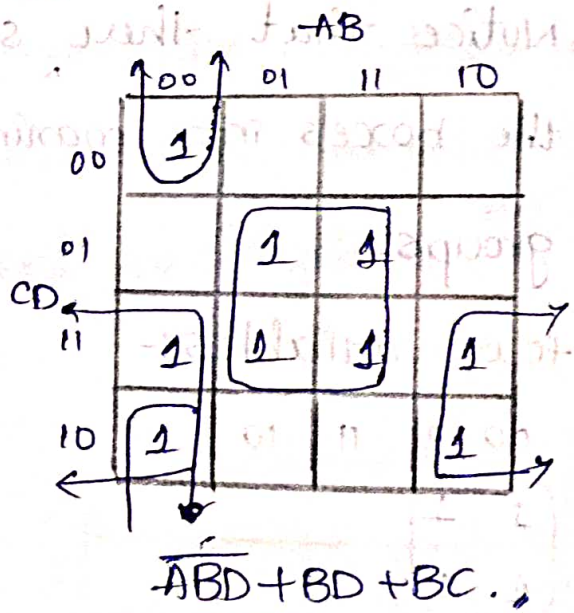
1). A for variable KV chart and other possible adjacencies is shown in the image.

2). Adjacencies can now consist of 1,2,4,8 and 16 boxes and the terms resulting will have 4, 3, 2, 1 and 0's in them respectively

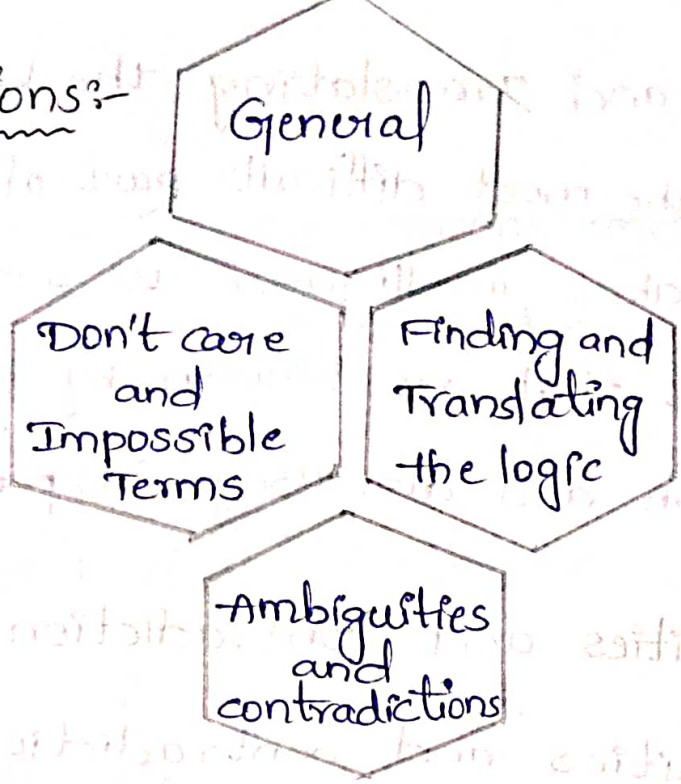








* specifications:-



1). General:- If the specification is not logically consistent and complete, there is no use in getting into design and implementation until an automated specification system is developed much of the specification analysis can be done using KV chart, paper and pencil.

2). Don't care and Impossible Terms:-

Impossible cases can be used to simplify logic and further to simplify programs implementing that logic, the external world can also contain some "impossible and "mutually exclusive" cases such as blind editors, consistent specification, honest politicians etc...

3. Finding and Translating the Logic:-

This is the most difficult part of the job as it takes intelligance to understand the intentions that are hidden by ambiguities in English and also usage (or) poor english.

4. Ambiguities and contradictions:-

Ambiguities and contradictions can be identified when we map the specfications onto a kv chart.

UNIT-IV.

Part-I: State, State Graphs and Transition testing.

⇒ State graphs

⇒ good and bad state graphs

⇒ State testing.

⇒ Testability tips.

Part-I: State, State Graph & Transition testing

* State graphs:-

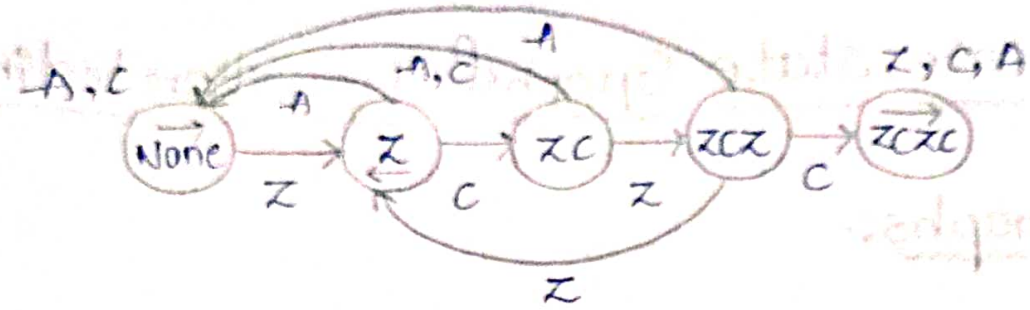
1). States:-

1). state is a condition (or) situation during which an object undergoes throughout its life time.

2). states are numbered (or) identified by characters (or) words (or) whatever else is convenient.

* Example:- A program that detects the character sequence ZCZC can be in the following states.

1. Neither ZCZC nor any part of it has been detected.
2. Z has been detected.
3. ZC has been detected.
4. ZCZ has been detected.
5. ZCZC has been detected.



2). Inputs and Transitions:-

1). whatever is being modeled is subjected to inputs.

2). As a result of those inputs, the state changes, (or) is said to have made a Transition.

3). Transitions are denoted by links that join the states.

3). outputs:-

1). Outputs are based on the input values.

2). when an input is applied to a state it is processed in order to produce an output.

4). Finite state Machine:

1). A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of

states and a finite number of transitions between states.

2). An output can be associated with any link

5). State Tables:-

1). If state graph has a large number of states and transitions, then it is difficult to follow them.

2). The state table for the tape control system is shown below.

STATE	OK	ERROR.
1	1/NONE	2/REWRITE
2	1/NONE	4/REWRITE
3	1/NONE	2/REWRITE
4	3/NONE	5/ERASE
5	1/NONE	6/ERASE
6	1/NONE	7/OUT
7	....	.....

6). Time versus sequence:-

1). State graph don't represent time - they represent sequence.

2). A transition might take microseconds (or) centuries.

7). Software implementation:-

i). Implementation and operation:

There four tables are involved.

1). INPUT_TABLE_CODE

2) TRANSITION_TABLE

3) OUTPUT_TABLE

4) DEVICE_TABLE.

ii). Input encoding and input alphabet:-

1). The input encoding here is for OTHER=0, for Z=1, for C=2.

2). The different encoded input values are called the input alphabet.

iii). output encoding and output Alphabet:-

1). A single character output for a link is rare.

2). These can be encode into a convenient output alphabet.

iv). State codes and state-symbol products:-

The term state-symbol product is used to convert the combined state and input code into a pointer to compact table.

v). Application comments for Designers:-

An explicit state table implementation is advantageous when either the control function is likely to change in the future.

vi). Application comments for Testers:-

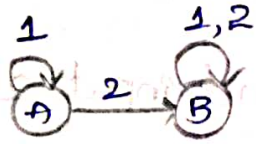
Independent testers are not usually taken with either implementation details (or) the economics of this approach.

* Good and bad state Graphs:-

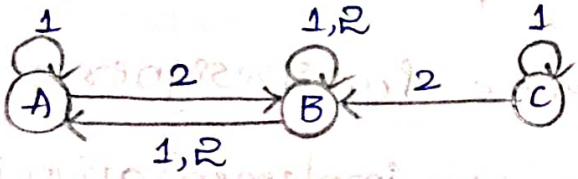
1). General:-

1). In testing we deal with a good state graph and also with a bad one.

2). The following figure shows examples of improper (or) bad state graphs.

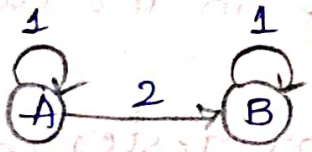


In state B the initial state can never be entered again.

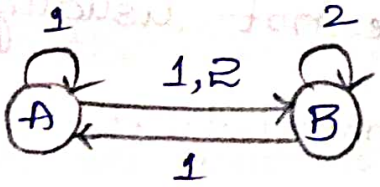


state C cannot be entered.

state A, B are not reachable.



No transition is specified for an input of 2, when in state B.



Two transitions are specified for an input of 1 in state A.

2). State Bugs:-

1). The bugs in states are called state bugs.

2). The state bugs arise due to the following reasons.

i). Number of states:

1). A state graph consists of the number of states.

2). It represents behaviour of the system.

ii). Impossible states:

1). A state that is not possible is called impossible states.

iii). Equivalent states:

Two states A, B are equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs.

3). Transition Bugs:-

1). The connectivity between two (or) more states is known as transition.

2). The bug in transition is called Transition Bug.

4). Unreachable states:-

1). An Unreachable state is like Unreachable code.

2). A state that no input sequence can reach.

5). Dead states:-

1). A dead state is a state that once entered cannot be left.

2). This is not necessarily a bug but it is suspicious.

6). output Errors:-

1). The states, transitions, and the inputs could be correct, there could be no dead (or) Unreachable states, but the output for the transition could be incorrect.

2). Output actions must be verified independ -ntly of state and transitions.

*Differentiate Between Good state Graph and Bad state Graphs-

S.NO	Good state Graph	Bad state Graph.
1.	A state graph is said to be good, when every state, input, transition and output is specified clearly and under standable.	A state graph is said to be bad, when every state, input, transition and output is not specified clearly and difficult to understand.
2.	In good state graph the sequence of inputs is specified for every state inorder to perform some action and that will help the systems to get back to the initial state.	In bad state graph, there is no sequence of inputs specified and this result to be the incorrect output.
3.	In good state graph, the bugs are less and easy to identify	In bad state graphs the bugs are more and difficult to identify.,

* State Testing :-

1). Impact of Bugs:

1). Let us say that a routine is specified as a state graph that has been verified as correct in all details.

2). From the following the bugs may occur.

i). wrong number of states.

ii). wrong transition.

2). principles:

1). state testing is defined as a functional testing technique to test the functional bugs in the entire system.

2). A set of tests consists of three sets of sequences.

i) Input sequences

ii) corresponding transitions

iii) output sequences.

3). Limitations and extensions :-

1). The limitation is state transition coverage in a state graph does not guarantee complete testing.

2). The extension is chow defines a hierarchy of path and methods for combining paths.

4). What to model:-

1). combination of hardware & software can be modeled sufficiently complicated state graph.

2). The state graph is behavioural model that it is functional rather than structural.

5). Getting the data:- Here labor intensive data gathering is needed and needs more meetings to resolves issues.

6). Tools:- Tools for hardware logic designs are needed.

* Testability tips:-

1). A balm for programmers:-

1). The key to testability design is easy that is we can easily build explicit finite state machines.

2). How big How small:-

1) For two finite state machines there are only eight good and bad ones.

2). For three finite states machines there are eighty possible good and bad one.

3). switches, Flags and unachievable paths:-

1). The functionality of switches and flags are almost similar in the state testing.

2). switches (or) flags are used as essential tool for state testing to test the finite state machine in every possible state.

4). Essential and inessential finite state behavior:-

1) To understand an essential and inessential

finite state behavior, we need to know the concept of finite state machines and combinational machines.

2). There is a difference between finite state machines and combinational machines in terms of quality.

5). Design guidelines:-

1). Finite state machine is represented by a state graph having a finite number of states and a finite number of transitions between states.

2). Finite state machine(FSM) is a functional testing tool and programming testing tool.,

UNIT-V.

Part-I: Graph Matrices & Application.

⇒ Motivational overview.

⇒ Matrix of graph

⇒ Relations

⇒ power of a matrix

⇒ Node Reduction Algorithm.

⇒ Building tools (student should be given an exposure to a tool like JMeter (or) win-runner).

Part-I: Graph Matrices and Applications

* Matrix of a Graph:-

1). The matrix in which every node of a graph is represented by one row and one column is called a graph matrix.

2). Graph is an abstract representation of a software structure.

3). Graphs can also be represented using matrices, the matrix when represents the structure of a graph is known as graph is known as graph matrix, graph matrices are introduced to overcome the problems faced using pictorial graphs.

4). A graph matrix is purely based on matrix methods which are ordered, structured and are more dependable than path tracing in a pictorial graph.

* constructing Test Tools:- Matrix methods of a graph matrix are the basic for constructing the test tools using analysis routines, moreover it is much more difficult to generate test cases for a pictorial graph than it is for a graph matrix.

* Implementing Testing Theory:-
In theoretical terms, graphs are the simple structures but when used in theorem proving, then matrix representations (i.e., graph matrix) are used.

* Basic Algorithmic Toolkit:- The basic algorithm toolkit contains certain methods. They are:-

1). Partition Method:- It is used for elimination loops from the graphs.

2). Multiplication of Matrix:-
It is used to derive path expressions between all possible node pairs.

3). Matrix Determinant:- It is used to derive path expressions between any pair of nodes.

* problems with pictorial graphs:-

1). Tracing a path in a pictorial graph is a difficult task.

2). There is every possibility of having an error while tracing i.e., one may miss an important and mandatory link (or) may include some links more than once.

3). yellow marking may not be considered as reliable because once the concentration is lost during marking one may lose link to be marked which ultimately leads to confusion.

4). It is very difficult to generate test cases for a pictorial graph and hence a considerable amount of time is wasted if pictorial graphs are used.

* cyclomatic complexity :-

1). cyclomatic complexity is also known as conditional complexity, it is a metrix used for computing the complexity of source code. and for counting the number of linearly independent paths comprising of the program.

2). The general formula to compute cyclomatic complexity is,

$$M = E - N + 2P.$$

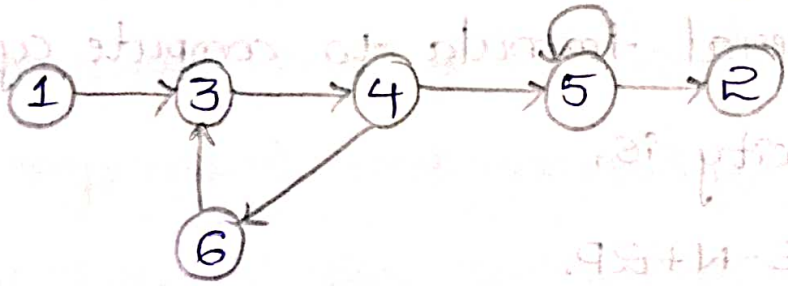
* cyclomatic complexity value assists the tester in number of ways :-

1). It is used to develop white box test cases.

2). It provides an estimation of testability of a module..

* connection Matrix: A connection matrix is also known as a graph matrix, it is a matrix of a graph in which the number of rows and columns correspond to the number of nodes within that graph.

* Example:- consider the following graph.



The connection matrix for this graph is,

Node \ connection node	1	2	3	4	5	6	
1	0	0	1	0	0	0	1-1=0
2	0	0	0	0	0	0	0
3	0	0	0	1	0	0	1-1=0
4	0	0	0	0	1	1	2-1=0
5	0	1	0	0	1	0	2-1=1
6	0	0	1	0	0	0	1-1=0.

$$M = 2+1 = 3.$$

⇒ In the above fig, '1' represents a connection and '0' represents no connection.

* Relations:-

* Definition:- The property by which two nodes are interconnected is known as a relation, it can be represented by a link connecting the nodes.

* Relation properties:-

1). Transitive Relations:-

1). For three nodes, a, b and c transitive property states that if node 'a' relates node 'b' and 'b' relates node 'c' then 'a' relates 'c'.

2). A relation is said to be a transitive relation if it satisfies transitive property.

* Examples:- If a>b and b>c

$\Rightarrow a > c.$

2). Reflective Relations:-

1). A relation is said to be reflexive, if for every 'a' there exists an equivalent 'a' of the form aRa.

2). Reflexive relation is similar to relation having self loop at all nodes.

3). symmetric Relations:-

1). A relation is said to be symmetric, if it satisfies the symmetric property.

2). This property states that, if there exists a relation aRb then bRa also exists for all a and b.

4). Antisymmetric Relations:-

A Relation 'R' is said to be antisymmetric if the relations aRb and bRa exists for all a and b such that both a and b are equal.

* Example:- if $a \leq b$ and $b \leq a \Rightarrow a = b$.

* Equivalence Relations:-

1). A relation is said to be an equivalence relation if it satisfies transitive, reflexi -ve and symmetric properties.

2). An equivalence class is a collection of objects that satisfies all the three properties of equivalence relation.

* partial ordering Relations:-

1). A relation is said to be partially ordered, if it satisfies transitive, reflexive and antisymmetric properties.

2). A graph which shows partial ordereding relation between its nodes is said to be partially ordered graph.

* power of a matrix:-

1). Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry.

2). The square of the matrix represents all path segments two links long.

* Matrix power and products:-

1). Given a matrix whose entries are a_{ij}, the square of that matrix is obtained by replacing every entry with.

$$\cdot \sum_{k=1}^{n} a_{ik} a_{kj}$$

2). More generally, given matrices A and B with entries a_{ik} and b_{kj}, respectively, their product is a new matrix C, whose entries are c_{ij}, where:-

$$\cdot \sum_{k=1}^{n} a_{ik} b_{kj}$$

* partitioning Algorithm:-

1). consider any graph over a transitive relation.

2). The graph may have loops.

3). such a graph is partially ordered.

4). Many graphs with loops are easy to analyze if you know where to break the loops.,

* Node Reduction Algorithm:-

1). The matrix powers usually tell us more than we want to know about most graphs.

2). The advantage of matrix reduction method is that it is more methodical than the graphical method called as node by node removal algorithm.

i) select a node for removal; replace the node by equivalent links that bypass that node and add those links the links they parallel..

ii). combine the parallel terms and simplify as you can.

iii) observe loop terms and adjust the out links of every node that had a self loop to account for the effect of the loop.

iv). The result is a matrix whose size has been reduce by 1.

v) continue until only the two nodes of interest exist.

* Building Tools:-

1). 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.

2). It is hard to build algorithms over visual graphs so the properties (or) graph matrices are fundamental to tool building.

* JMeter:-

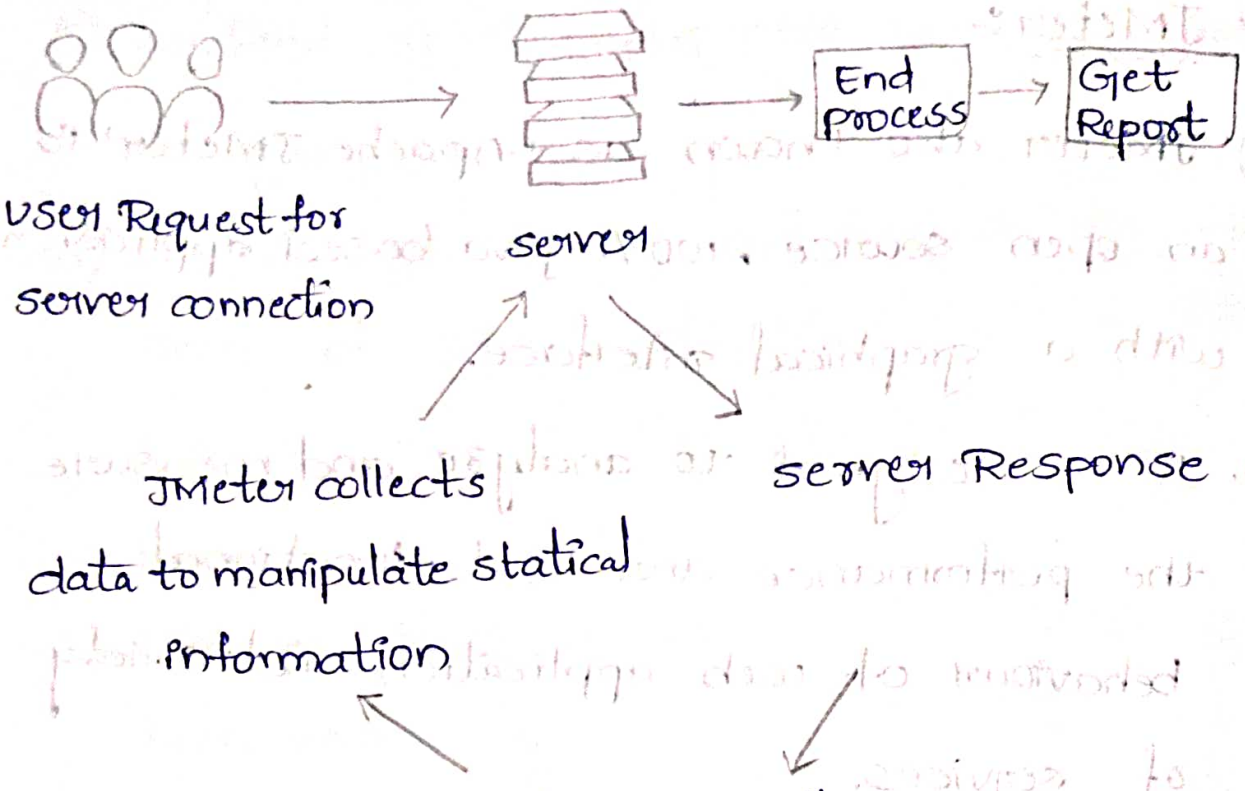
1). JMeter also known as 'Apache JMeter' is an open source, 100% java based application with a graphical interface.

2). It is designed to analyze and measure the performance and load functional behaviour of web application and variety of services.

* Working of JMeter:-

1). JMeter sends requests to a target server by simulating a group of users.

2). Subsequently, data is collected to calculate statistics and display performance metrics of target server through various formats.



* Win-Runner:-

The five steps involved in performing a test using win-Runner are:-

1). Defining the way of performing object Recognitions:-

1). object recognition is important in order to perform operations on application repeatedly.

2). The recognized objects are located with logical names and stored in a GUI Map file

2). Creating test:-

1). In this step, the test script is generating that perform testing.

2). The output generated from this process is a script file carrying data written in TSL (Test script Language).

3). Debugging test:-

1). In this step, the errors (or) bugs are identified by performing the test multiple times.

2). The errors usually are syntactic (or) semantic.

4). Running test:-

1). This step is initiated only when debugging step indicates no bugs / errors.

2). The test in this step is executed in special mode called verify Run mode.

5). Analyzing Test:

1). In this step, the results generated from the test are analyzed using Test Result viewer tool.

2). with this, a detailed analysis can be done on test results.