Chapter 2: Flow Graphs and Path Testing

Introduction:

In this section we will look in detail about four specification-based or black box techniques. These techniques are needed to design test cases. The specification based technique of use case testing also shall be covered.

Test analysts and technical test analysts lay major attention to design, implement and execute tests using various testing techniques. Specification-based techniques are one of such techniques. Specification-based techniques are popular by more refined names like behavior-based techniques or Black Box test design techniques. These techniques can be used for any level of test activity. These are used by both test analysts and technical test analysts, but exploited mainly by the test analysts.

"Code is designed and developed from the software requirements"

Specifications (SRS) documents are the primary input documents, likewise specification-based test techniques too are based upon the test conditions & test cases derived from the SRS documents.

We can make our specifications that can be in the form of text documents, pictures, models, compilation of features, or any other document which could help us in understanding as to what we expect from the software & how it is going to accomplish that.

Test coverage is represented by the percentage of the specified items addressed by the designed tests. Coverage of all the specified items does not necessarily indicate complete test coverage, but it does indicate that we have addressed what was specified. For further coverage, we may need to look for additional information.

Sr.	Name of the Techniques	Brief Description	Coverage Criteria
1.	Equivalence Partitioning	It involves grouping of test conditions into various partitions, which are handled in the same way.	Number of Coverage Partitions / Total No. of Partitions
2.	Boundary Value Analysis (BVA)	It involves defining boundaries of the partitions & testing for them thereafter.	Number of Distinct Boundary Values covered / Total No. of Boundary Values
3.	Decision table testing / cause-effect graphing	It involves defining & testing for different combinations of conditions.	Number of combinations of conditions covered / Maximum No. of combinations of conditions
4.	State transition Tables	It involves Identification of all types of valid states & transitions, which need to be tested.	For single transitions, the coverage metrics is the percentage of all valid transitions exercised during test. This is known as 0-switch coverage. For n-transitions the coverage measure is the percentage of all valid sequences of n-transitions exercised during the test. This is known as (n-1) switch coverage
5.	All pair testing / Orthogonal Array Testing	It involves Identification of various combinations of configurations that need to be tested.	
6.	Classification Tree Method	It involves the use of graphical notations to describe the classes or test conditions and combinations handled by the test cases.	It depends upon the technique applied, e.g. pair wise is distinct from classification trees.
7.	Use Case Testing	It involves Identification of various usage scenarios & then testing accordingly.	No formal criteria apply.

Fig 1: Specification Based Test Design Techniques

Basics concepts of path testing:

Path Testing:

Definition:

Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications. It can be applied at different levels of granularity.

- The Specifications are accurate
- The Data is defined and accessed properly
- There are no defects that exist in the system other than those that affect control flow

Example:

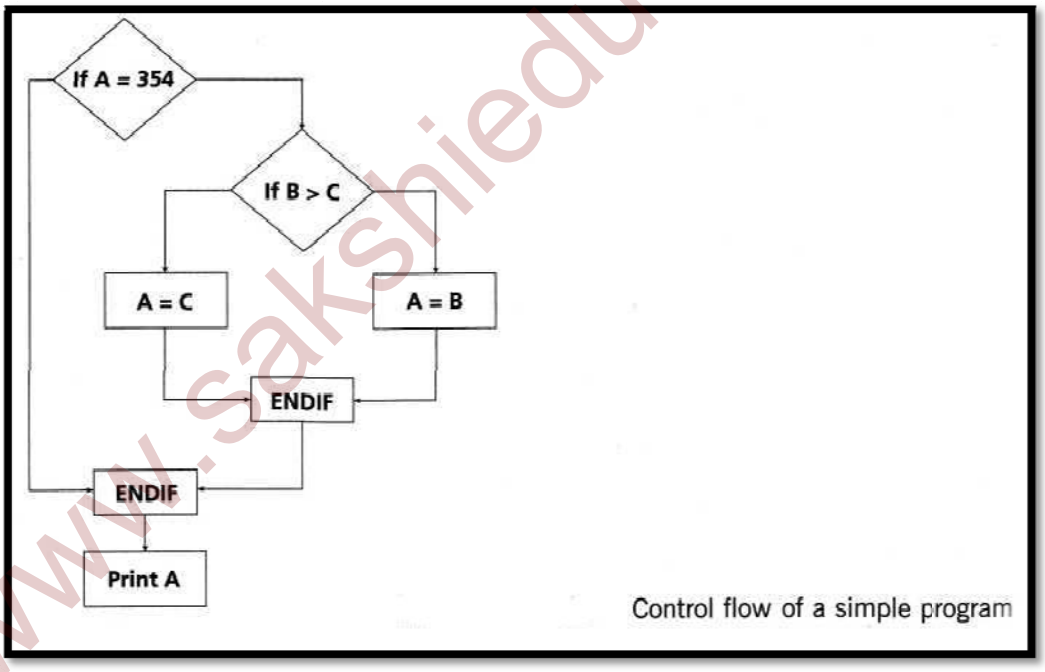

Fig 2: Sample Example for Path Testing

- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths are properly

chosen, then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.

- Path testing techniques are the oldest of all structural test techniques.
- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

Path Testing Techniques:

Control Flow Graph (CFG)

The Program is converted into Flow graphs by representing the code into nodes, regions and edges.

Decision to Decision path (D-D)

The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.

Independent (basis) paths

Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

Path Testing Concepts:

- Path is a sequence of statements starting at an entry, junction or decision and ending at another, or possibly the same junction or decision or an exit point.
- Link is a single process (block) in between two nodes.
- Node is a junction or decision.

- Segment is a sequence of links. A path consists of many segments.
- Path segment is a succession of consecutive links that belongs to the same path. (3,4,5)
- Length of a path is measured by # of links in the path or # of nodes traversed.
- Name of a path is the set of the names of the nodes along the path. (1,2,3 4,5, 6), (1,2,3,4, 5,6,7, 5,6,7, 5,6)
- Path-Testing Path is an “entry to exit” path through a processing block

Predicates:

The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: A>0, x+y>=90.....

Path Predicate:

A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", "x+y>=90", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

Multiway branches:

The path taken through a multi-way branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms. Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multi-way branches as an equivalent set of **if..then..else statements.**

For example a three way case statement can be written as: If case=1 DO A1 ELSE (IF Case=2 DO A2 ELSE DO A3 ENDIF) ENDIF.

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. **For example**, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types. The input for a particular test is mapped as a one dimensional array called as an Input Vector.

Predicate interpretation:

The simplest predicate depends only on input variables. For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.

Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignment statement y=x2+7. Although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.

The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called predicate interpretation. Sometimes the interpretation may depend on the path; for example,

```
INPUT X

ON X GOTO A, B, C, ...

 A: Z := 7 @ GOTO HEM

 B: Z := -7 @ GOTO HEM

 C: Z := 0 @ GOTO HEM

  .....
```

HEM: DO SOMETHING

.....

THEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends on the path we took through the first multi way branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0. The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

INDEPENDENCE OF VARIABLES AND PREDICATES:

The path predicates take on truth values based on the values of input variables, either directly or indirectly. If a variable's value does not change as a result of processing, that variable is independent of the processing. If the variable's value can change as a result of the processing, the variable is process dependent.

A predicate whose truth value can change as a result of the processing is said to be process dependent and one whose truth value does not change as a result of the processing is process independent. Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

Correlation of Variables and Predicates:

Two variables are correlated if every combination of their values cannot be independently specified. Variables whose values can be specified independently without restriction are called uncorrelated. A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.

For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select ‘X’ and ‘Y’ values to satisfy the first predicate, we might have forced the 2nd

predicate's truth value to change. Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

Path Predicate Expressions:

A path predicate expression is a set of Boolean expressions, all of which must be satisfied to achieve the selected path.

Example:

X1+3X2+17>=0

X3=17

X4-X1>=14X2

Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path. Sometimes a predicate can have an OR in it.

Example:

A: X5 > 0

B: X1 + 3X2 + 17 >= 0

C: X3 = 17

D: X4 - X1 >= 14X2

E: X6 < 0

B: X1 + 3X2 + 17 >= 0

C: X3 = 17

D: X4 - X1 >= 14X2

Boolean algebra notation to denote the Boolean expression:

ABCD+EBCD=(A+E)BCD

Predicate Coverage:

Compound Predicate: Predicates of the form A OR B, A AND B and more complicated Boolean expressions are called as compound predicates.

Sometimes even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17 . Or. X<17.

Predicate coverage is achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.

As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

Path Sensitizing:

- We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.
- Extract the programs control flow graph and select a set of tentative covering paths.
- For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
- Trace the path through, multiplying the individual compound predicates to achieve a Boolean expression such as

(A+BC) (D+E) (FGH) (IJ) (K) (l) (L).

- Multiply out the expression to achieve a sum of products form:

  ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL

- Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
- Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
- If you can find a solution, then the path is achievable.
- If you can't find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called PATH SENSITIZATION.

o HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with un correlated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

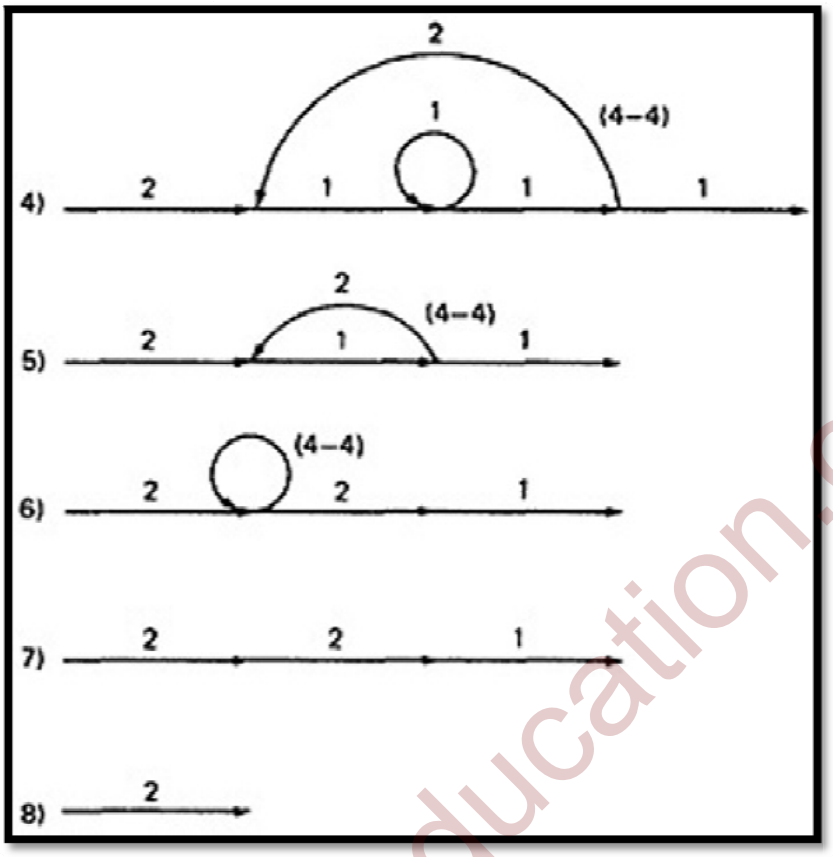

Fig 3: Path sensitizing

Path Instrumentation:

- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
- Co-incidental Correctness: The coincidental correctness stands for achieving the desired outcome for wrong reason.

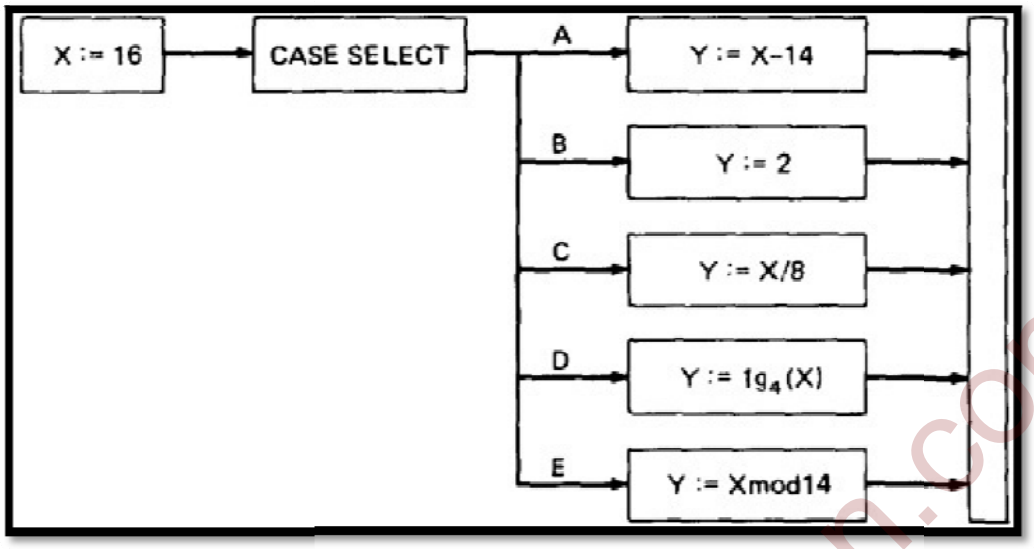

Fig 4: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. Path Instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.

The types of instrumentation methods include:

Interpretive Trace Program:

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the

path from its massive output dump is more work than simulating the computer by hand to confirm the path.

Traversal Marker or Link Marker:

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

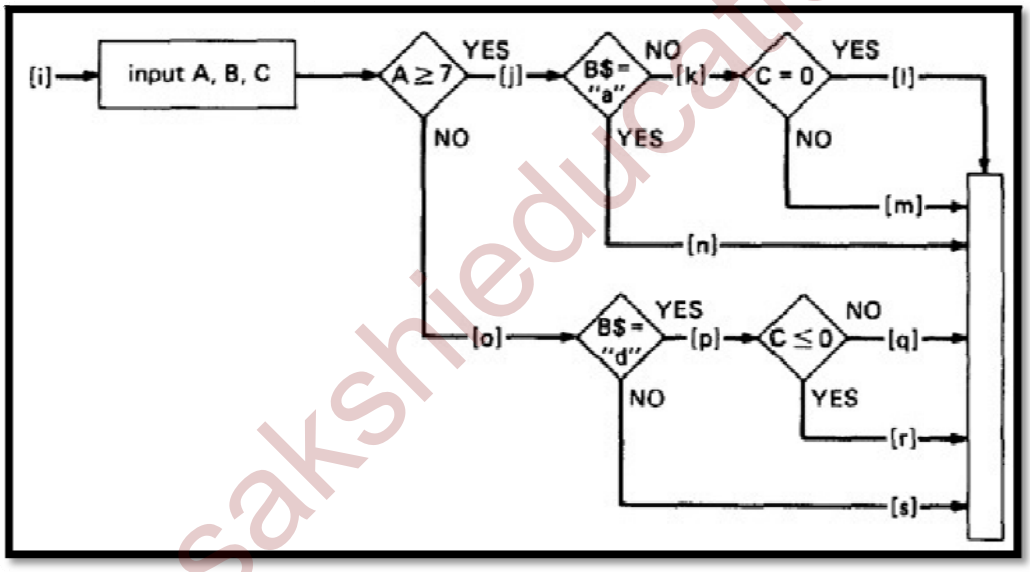

Fig 5: Single Link Marker Instrumentation

We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

Two Link Marker Method:

- The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and on at the end.

- The two link markers now specify the path name and confirm both the beginning and end of the link.

Link Counter:

A less disruptive (and less informative) instrumentation method is based on counters.

- Path testing based on structure is a powerful unit-testing tool. With suitable interpretation, it can be used for system Instead of a unique link name to be pushed into a string when the link is traversed; we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to the double link counter functional tests.
- The objective of path testing is to execute enough tests to assure that, as a minimum, C1 + C2 have been achieved.
- Select paths as deviations from the normal paths, starting with the simplest, most familiar and most direct paths from the entry to the exit. Add paths as needed to achieve coverage.
- Add paths to cover extreme cases for loops and combinations of loops: no looping, once, twice, one less than the maximum, the maximum. Attempt forbidden cases.
- Find path-sensitizing input-data sets for each selected path. If a path is unachievable, choose another path that will also achieve coverage. But first ask yourself why seemingly sensible cases lead to unachievable paths.
- Use instrumentation and tools to verify the path and to monitor coverage.
- Incorporate the notion of coverage (especially C2) into all reviews and inspections. Make the ability to achieve C2 a major review agenda item.
- Design test cases and path from the design flow graph or PDL specification but sensitize paths from the code as part of desk checking. Do covering test case designs either prior to coding or concurrently with coding.
- Document all tests and expected test results as copiously as you would document code. Put test suites under the same degree of configuration control used for the software it tests. Treat each path like a subroutine. Predict and document the outcome for the

stated inputs and the path trace (or name by links). Also document any significant environmental factors and preconditions.

- Your tests must be reproducible so that they can serve a diagnostic purpose if they reveal a bug. An undocumented test cannot be reproduced. Automate test execution.
- Be creatively stupid when conducting tests. Every deviation from the predicted outcome or path must be explained. Every deviation must lead to either a test change, a code change, or a conceptual change.
- A test that reveals a bug has succeeded, not failed.

Application of Path Testing

Path testing methods are mainly used in unit testing. However to create an environment in order to provide required inputs and also to receive the outputs from such units, we need to do test harness in order to create environment with required test stubs and test drivers.

In order to perform testing in sub routines that need to be integrated, we have to think about paths within the sub routine; then to achieve test coverage both statement coverage (C1) and branch coverage (C2) we need to create control flow graph, arrive at equivalent predicate notation representation, list predicate paths, identify predicate, identify predicate values for each of identified paths and finally select appropriate input values that would result in required predicate values and their by execute corresponding path.