**Module-3**

**Structural and Fault Based Testing**

**Overview:**

* This module gives knowledge about the different ways of deriving test cases with code.
* At the end of this module, one should be able to derive test cases following a strategy for any type of code given and should be able to derive values for various coverage metrics.

**What is Structural/white box/transparent or code-based Testing?**

* Test cases are derived from the software structure or internal implementation.
* Since the basis is absolute, structural testing methods are amenable to rigorous definitions, mathematical analysis and precise measurement.
* Under structural testing 4 strategies through which test cases can be derived.

1. Path Testing
2. DataFlow Testing
3. Slice Based Testing
4. Mutation/Fault Based Testing

* To understand the concepts of path and data flow testing, one must be aware of the program graph as it is the starting point for generating test cases.

**Program Graph:**

**Definition:**

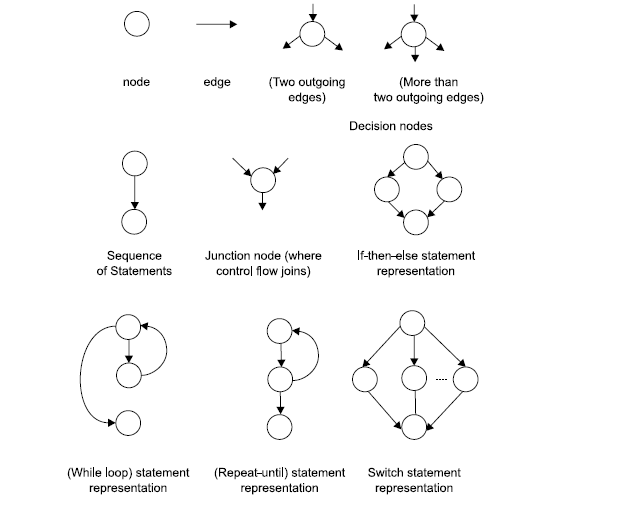
* Irrespective of the programming language, a program graph is constructed by using the program code.
* Program graph is a directed graph with nodes and edges.

1. Nodes🡪 Program statements
2. Edges🡪Flow of Control

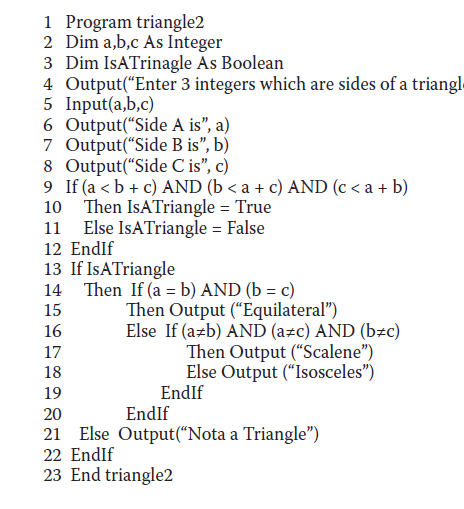
* Edge between two nodes can be formed portraying the sequence of execution.
* If *i* and *j* are nodes in the program graph, an edge exists from node *i* to node *j* if and only if the statement fragment corresponding to node *j* can be executed immediately after the statement fragment corresponding to node *i*.

**Style Choices of Program Graph:**

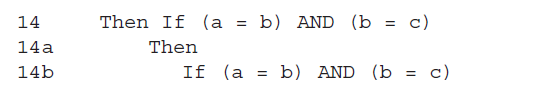
* Deriving a program graph from a given program is an easy process with the available constructs:

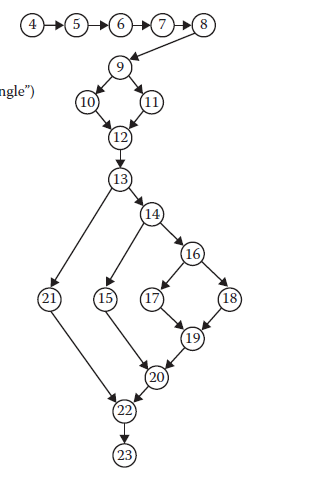


* Sometimes it is convenient to keep a fragment as a separate node; other times it seems better to include this with another portion of a statement.
* Consider the triangle problem pseudocode as given below:



* The line no 14 could be split into two lines as below:



* This latitude collapses onto a unique DD-path graph, so the differences introduced by differing judgments are moot.
* 
* **Sequence Nodes🡪 4 to 8**

**If then else construct🡪9 to 12**

**Nested if-then-else 🡪 13 to 22**

**Source Node🡪 4**

**Sink Node 🡪23No loops exist, so this is a directed acyclic graph.**

* The importance of the program graph is that program executions correspond

to paths from the source to the sink nodes.

* Because test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises.
* With the program graph, one can come to a conclusion if the program is structured or not .
* Size of the program is not something that we have to use to decide if its structural or not.
* A large program may be a structured program whereas a small program may be unstructured due to a loop in a program. If we have a loop in a program, large number of paths may be generated.
* Myers [MYER04] has shown 1014 paths in a very small program graph due to a loop that iterates up to 20 times.
* This shows how an unstructured program may lead to difficulties in even finding every possible path in a program graph. Hence, testing a structured program is much easier as compared to any unstructured program.

**Note: Given a program statement, you should be in a position to construct the program graph. We have done many examples in class…kindly go through those.**

**DD Path Graph:**

* The Decision to Decision (DD) path graph is an extension of a program graph. It is widely known as DD path graph.
* There may be many nodes in a program graph which are in a sequence.
* When we enter the first node of the sequence, we can exit only from the last node of that sequence.
* In DD path graph, such nodes which are in a sequence are combined into a block and are represented by a single node. Hence, the DD path graph is a directed graph in which nodes are sequences of statements and edges are control flow amongst the nodes.
* All programs have an entry and an exit, and the corresponding program graph has a source node and a destination node. Similarly, the DD path graph also has a source node and a destination node.
* We prepare a mapping table for the program graph and the DD path graph nodes.
* A mapping table maps nodes of the program graph to the corresponding nodes of the DD path graph. This may combine sequential nodes of the program graph into a block and that is represented by a single node in the DD path graph.
* This process may reduce the size of the program graph and convert it into a more meaningful DD path graph.

**Formal Definition:**

A *DD-path* is a sequence of nodes in a program graph such that

**Case 1: It consists of a single node with indeg = 0.**

**Case 2: It consists of a single node with outdeg = 0.**

**Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.**

**Case 4: It consists of a single node with indeg = 1 and outdeg = 1.**

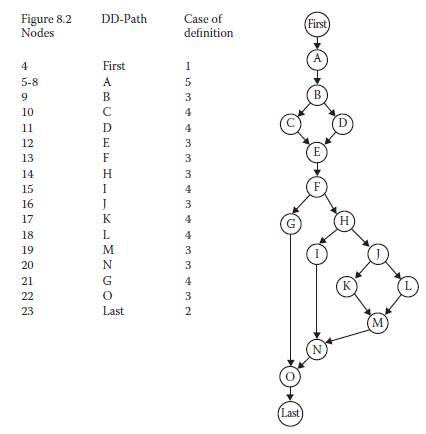
**Case 5: It is a maximal chain of length ≥ 1.**

Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured

program as initial and final DD-paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path. Case 4 is needed for “short branches”; it also preserves the one-fragment, one DD-path principle. Case 5 is the “normal case,” in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The “maximal” part of the case 5 definition is used to determine the final node of a normal (nontrivial) chain.

**Construction of DD graph for triangle problem:**

Consider the program graph depicted above for the triangle problem and the corresponding DD graph after applying the formal definition is depicted below:



* Node 4 is a case 1 DD path, this node in program graph is named as FIRST in DD graph
* Node 22 is a case 2 DD path, this node in program graph is named as LAST in DD graph.
* Nodes 5 through 8 are case5 DD paths. Node 8 is the last node in this path, and this is the node that preserves the 2-connectedness property of the chain.
* If we go beyond node 8 and try to include node 9 and 10, they hold only 1 connectedness so they can’t be included.
* Stopping at node 7 violates maximal criterion.
* Node 11 is a case 3 DD path which forces nodes 9 and 10 to be individual DD paths by case 4.
* Nodes 12 through 14 are case 5 DD-path as that of Node 5 and 8.
* Nodes 14 through 20 correspond to sequence of IF-THEN. Nodes 16 and 18 are both case 3 DD paths forcing 15,17 and 19 to be case 4 DD paths. Node 20 is case 3 and node 21 is case 4 DD path.

In short, DD graph is form of condensation graph in which the 2 connectedness components are collapsed into single node preserving the case5 definition. Single node DD paths are required to preserve the convention of being exactly in only one DD path. If these conventions are not followed, it might end up with a clumsy DD graph.

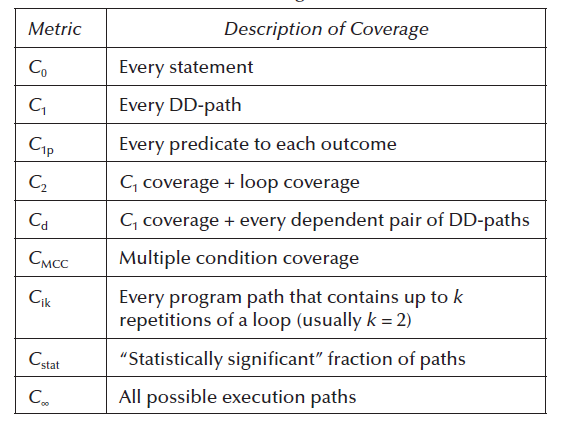
This process should not intimidate testers—high-quality commercial tools are available, which

generate the DD-path graph of a given program. The vendors make sure that their products work

for a wide variety of programming languages. In practice, it is reasonable to manually create DD-path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

**Test Coverage Metrics:**

* Some of the major problems with functional testing is its inability to identify the possibility of gaps and the extent of redundancy.
* We are unable to identify these flaws in functional testing due to the limited usage of coverage metrics.
* But in structural testing, the test coverage metrics plays a significant role in identifying the effectiveness of the test suite considered for testing.
* Test coverage metrics are a device to measure the extent to which a set of test cases covers a program.
* Sensibly managing the test process is possible only through the accepted form of test coverage metrics.
* Miller(1977) presented an organized view of the test metrics as depicted in the below table:



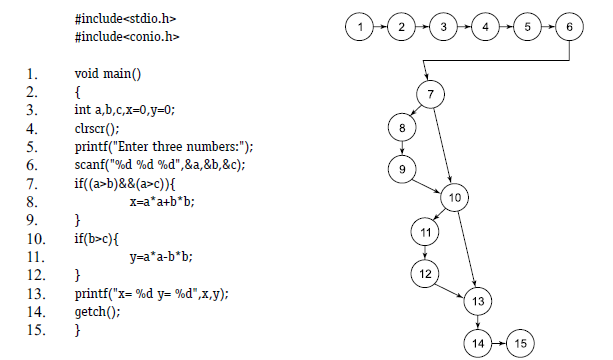
* The metrics mentioned in the table tells us what to test it but it gives zero knowledge on how to test it.
* Miller’s test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

**Statement Coverage:**

* Statement Coverage denotes the execution of every single line in a program to achieve 100% statement coverage.
* If some statements have not been executed by the set of test cases, there is clearly severe gap in the test coverage.
* Although less adequate than DD-path coverage, the statement coverage metric (*C*0) is still widely accepted: it is mandated by ANSI (American National Standards Institute) Standard 187B and has been used successfully throughout IBM since the mid-1970s.

**Example:**

Consider the below source code with program graph:



**If, we select inputs like:**

**a=9, b=8, c=7, all statements are executed, and we have achieved 100% statement coverage by only one test case.**

**DD-Path Testing: (C1 Metric)**

* C1 metric is exactly *G*chain metric otherwise referred to as chain coverage if executed assures every chain of length greater than or equal to 2 in the program graph is traversed.
* For if–then and if–then–else statements, this means that both the true and the false branche are covered (*C*1p coverage).
* For CASE statements, each clause is covered.
* Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. These should employ more exercise on functional tests.

**Example:**

Consider the same program mentioned in statement coverage, and

If we select a = 9, b =8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions):

Path = 1–15

We also want to select all false conditions with the following inputs:

a = 7, b = 8, c = 9, the path followed is

Path = 1–7, 10, 13–15

These two test cases out of four are sufficient to guarantee 100% branch / chain coverage.

The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

**Simple Loop coverage: ( C2 metric)**

* The *C*2 metric requires DD-path coverage (the *C*1 metric) plus loop testing.
* The simple view of loop testing is that every loop involves a decision, and we need to test both outcomes of the decision: one is to traverse the loop, and the other is to exit (or not enter) the loop and it is equivalent to *G*edge test coverage.

**Condition Coverage:**

* Condition coverage is better than branch coverage because we want to test every condition at

least once. However, branch coverage can be achieved without testing every condition.

* Consider the seventh statement of the program given in statement coverage.
* The statement number 7 has two conditions (a>b) and (a>c).
* There are four possibilities namely

(i) Both are true

(ii) First is true, second is false

(iii) First is false, second is true

(iv) Both are false

* If a > b and a > c, then the statement number 7 will be true (first possibility).
* However, if a< b, then second condition (a > c) would not be tested and statement number 7 will be false(third and fourth possibilities).
* If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition.
* Selected inputs may be given as:

(i) a = 9, b = 8, c = 7 (first possibility when both are true)

(ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)

(iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)

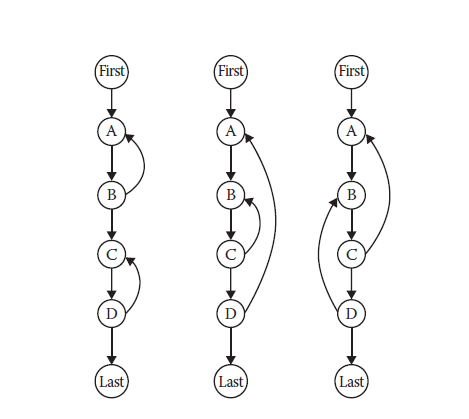
* Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

**Dependent Pairs of DD paths:**

* Identification of dependencies must be made at the code level. This cannot be done just by considering program graphs.
* The *C*d metric foreshadows the topic of data flow testing.

**Complex Loop Coverage:**

* Loops are a highly fault prone portion of source code and variety of loops like concatenated,nested and horrible loops exists.
* Miller’s *C*ik metric extends the loop coverage metric to include full paths from source to sink nodes that contain loops.



* Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Horrible loops can’t occur if structured programming percepts are followed. When it is possible to branch into (or out from) the middle of a loop, and these branches are internal to other loops, the result is Beizer’s knotted loop.
* If loops are knotted, it will be necessary to carefully analyze them in terms of the data flow methods.
* If the body of a simple loop is a DD-path that performs a complex calculation, this should also be tested, taking a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values.
* Once a loop has been tested, the tester condenses it into a single node.
* If loops are nested, this process is repeated starting with the innermost loop and working outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable.

**Multiple Condition Coverage:**

* *C*MCC metric addresses the question of testing decisions made by compound conditions.
* Compound conditions in DD-paths we should investigate the different ways that each outcome can occur.
* One possibility is to make a decision table: a compound condition of three simple

conditions will have eight rules yielding eight test cases.

* Another possibility is to reprogram compound predicates into nested simple if–then–else logic, which will result in more DD-paths to cover.

**Modified Condition Decision Coverage:**

*MCDC* requires

1. Every statement must be executed at least once.

2. Every program entry point and exit point must be invoked at least once.

3. All possible outcomes of every control statement are taken at least once.

4. Every nonconstant Boolean expression has been evaluated to both true and false outcomes.

5. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes.

6. Every nonconstant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

**Note:**

**Given a code fragment, you should be in a position to identify the number of test cases required to achieve 100% statement coverage, path coverage, condition and branch coverage.**

**Rule of Thumb:**

**Statement Coverage --Every statement should be executed.**

**Branch Coverage—Every Yes/No condition should be executed.**

**Condition Coverage—Every conditions true/false condition.**

**Path Coverage—Total number of independent paths should be checked with the test cases.**

**Path Based Testing:**

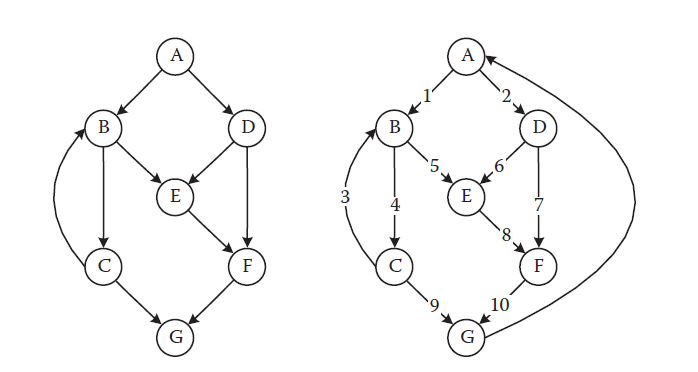
* A very important mathematical notion in structural testing is “Basis”
* Sets are defined by the basis and the basis has very important properties with respect to the entire set.
* Mathematically basis is defined by a structure called “Vector space”.
* Vector space consists of the set of elements and operations and every vector space possess Basis.
* Basis of a vector space is a set of vectors/elements that are independent of each other and “span” the entire vector space in the sense that any other vector in the space can be expressed in terms of the basis vectors.
* Thus, a set of basis vectors somehow represents “the essence” of the full vector space: everything else in the space can be expressed in terms of the basis, and if one basis element is deleted, this spanning property is lost.
* This mathematical theory is applied to perform structural testing considering program as a vector space, and then the basis for such a space is the set of elements to test.
* Thomas McCabe recognized this possibility in the mid-1970s.

**McCabe’s Basis Path Method:**

* McCabe’s basis path method is a measure to identify the path coverage in structural testing.
* It relies on Cyclometic complexity calculation in graph theory to identify the number of independent paths in the program graph.
* After the identification of independent paths, the number of test cases required for covering 100% path coverage is identified.
* McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number of a strongly connected graph is the number of linearly independent circuits in the graph.

**Example:**

Let us understand the concept of McCabe’s basis path method with the example:



* The directed graph shown above consists of single entry node “A” and single exit node “G”
* Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.
* We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node as shown in the connection from G to A.
* The number of independent paths in this graph is calculated using the formulae ***V*(*G*) = *e* – *n* + *p*** is the number of edges, *n* is the number of nodes, and *p* is the number of connected regions.
* The number of linearly independent circuits of the graph *V* (*G*) = *e* -*n* + *p* = 11- 7 +1 = 5.
* The cyclomatic complexity of the strongly connected graph in Figure above is 5; thus, there are five linearly independent circuits.
* If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths.
* Here, we identify paths as sequences of nodes:

p1: A, B, C, G

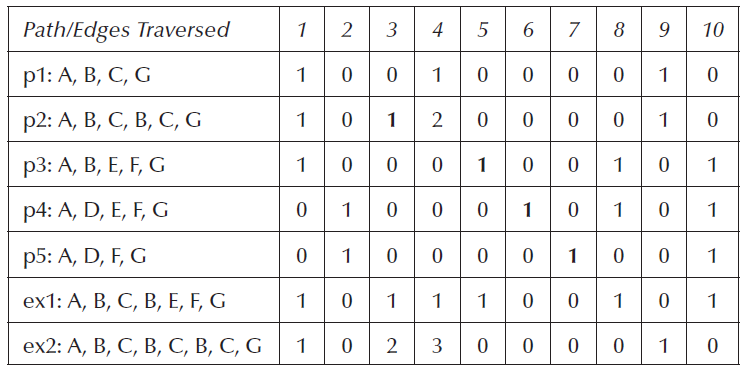
p2: A, B, C, B, C, G

p3: A, B, E, F, G

p4: A, D, E, F, G

p5: A, D, F, G

**Justification with Vector Space:**



* We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication.
* Path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths.
* Rows correspond to paths, and columns correspond to edges, The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9, while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Because edge 4 is traversed twice by path p2, that is the entry for the edge 4 column.
* We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must be independent.
* Path p1 is independent of all of these, because any attempt to express p1 interms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G.

**Algorithmic Procedure-Baseline method:**

* Another methodology used to identify the basis path in a program.
* The procedure starts of with the selection of baseline path which is selected randomly with the condition that the baseline path chosen should be normal execution.
* Next the baseline path is retraced and in turn each decision is flipped

Example:

Let us try to understand the baseline approach with the above example.

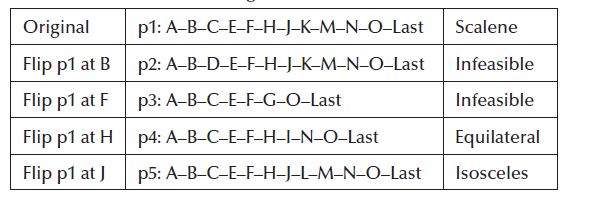
**Step 1: Selection of Baseline path so we are choosing A, B, C, B, E, F, G.**

**Step 2: Retracing and flipping**

**Step 3: Flipping generally happens when the retracing reaches a node with outdegree≥2.**

* The first decision node (outdegree ≥ 2) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1.
* We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible
* For the next path, we can follow the second path,and take the other decision outcome of node D, which gives us the path A, D, F, G.
* Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F,G and A, B, C, G.

**Contradictions in McCabe’s Basis Path:**

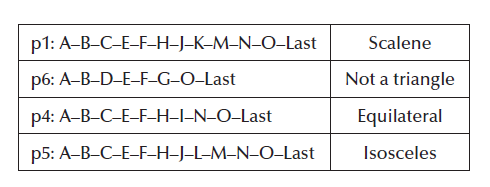


* Consider the table mentioned above, starting of path1 as baseline with test inputs for scalene as 3,4 and 5.
* This test case will traverse path and if we flip at node B path2 is obtained, flipping continuously at F,H and J provided p2,p3,p4 and p5.
* Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G.
* Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively.
* Notice that we do not have a basis path for the NotATriangle case.
* McCabe’s procedure successfully identifies basis paths that are topologically independent; however, when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible.
* One solution to this problem is to always require that flipping a decision results in a semantically feasible path.
* Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:

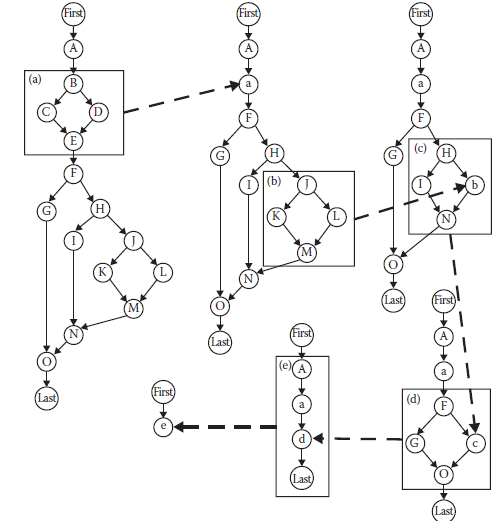
If node C is traversed, then we must traverse node H.

If node D is traversed, then we must traverse node G.

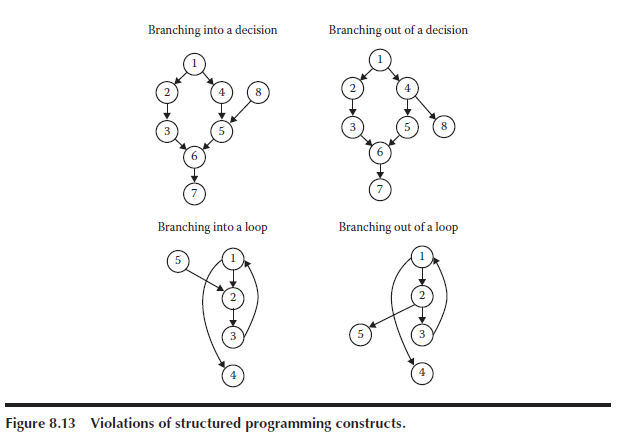
* Taken together, these rules, in conjunction with McCabe’s baseline method, will yield the following feasible basis path set.



* Irrespective of all the contradictory statements, McCabe’s work more on cyclometic complexity that can actually improve the coding than testing.
* If a program is well structured, it can always be reduced to a graph with one path.



* Consider the figure above which shows the program graph of triangle problem , and in the process of constructing the DD graph, the complete program graph is constructed to have only one path.
* This is always possible if the program written in highly structured.
* The bottom line for testers is this: programs with high cyclomatic complexity require more testing.
* Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; *V*(*G*) = 10 is a common choice.
* What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing.
* If the unit is well structured, its essential complexity is 1; so it can be simplified easily.
* If the unit has an essential complexity greater than 1, often the best choice is to eliminate the violations.



**Note:**

**As much as Path based testing is concerned, be clear with DD –path graph, coverage metrics and McCube’s complexity. Path testing we identify the number of paths using cyclometic complexity value and the number of test cases usually equals the cyclometic value.**

**DataFlow Testing:**

**Why Data Flow Testing?**

1.#include<stdio.h>

2.Void main()

3.{

4.Int a ,b,c;

5.a=b+c;

6.printf(“%d”, a);

7.}

* Observe the above code snippet and if we are been asked to write the output of the above code, we very well answer the value of “a” depends on the value of b and c.
* So if we are giving the value of b and c and 2 and 3 the value of “ a” would be 5.
* But we cannot expect the same result 5, sometimes it may give us garbage value, so this is something which is completely relying on the compiler.
* This demands a form of testing in which the variable usage is verified and that is what we are trying to do in dataflow testing.

**What is Dataflow Testing?**

* Dataflow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used.

Two points of concern are:

1. Statements where variables receive values(definition)

2. Statements where these values are used(referenced)

* In fact dataflow testing serves as a reality check for path testing.
* Dataflow testing looks cumbersome at unit level, however a very good form of testing in object oriented approach.
* There are two forms of dataflow testing:

1. One provides a set of basic definitions and a unifying structure of test coverage metrics.
2. Other is based on a concept called a “program slice”.

**Define/Reference Anomalies:**

* A variable that is defined but never used (referenced)
* A variable that is used before it is defined
* A variable that is defined twice before it is used

Each of these anomalies can be recognized from the concordance of a program. Because the concordance information is compiler generated, these anomalies can be discovered by what is known as static analysis: finding faults in source code without executing it.

**Definitions in Dataflow testing:**

1. **Defining Nodes:**

* Node *n* ∈ *G*(*P*) is a *defining node* of the variable *v* ∈ *V*, written as DEF(*v*, *n*), if and only if the value of variable *v* is defined as the statement fragment corresponding to node *n*.
* Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes.
* When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

1. **Usage Node:**

* Node *n* ∈ *G*(*P*) is a *usage node* of the variable *v* ∈ *V*, written as USE(*v*, *n*), if and only if the value of the variable *v* is used as the statement fragment corresponding to node *n*.
* Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes.
* When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

1. **Predicate Node usage:**

* A usage node USE(*v*, *n*) is a *predicate use* (denoted as P-use) if and only if the statement *n* is a predicate statement; otherwise, USE(*v*, *n*) is a computation use (denoted C-use).
* The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have an outdegree ≤ 1.

1. **Definition Use Path:**

* A *definition/use path* with respect to a variable *v* (denoted du-path) is a path in PATHS(*P*) such that, for some *v* ∈ *V*, there are define and usage nodes DEF(*v*, *m*) and USE(*v*, *n*) such that *m* and *n* are the initial and final nodes of the path.

1. **Definition Clear Path:**

* A *definition-clear path* with respect to a variable *v* (denoted dc-path) is a definition/use path in PATHS(*P*) with initial and final nodes DEF(*v*, *m*) and USE(*v*, *n*) such that no other node in the path is a defining node of *v*.

**Generating test cases adopting Dataflow Testing:**

Consider the greatest of three numbers program given below:

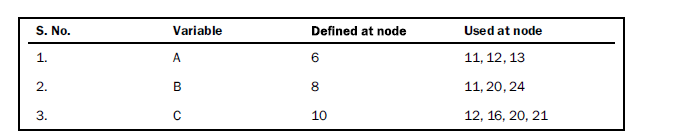
A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

* The given program possess three variables A,B and C and its



* Next the du-path construction table

A screenshot of a cell phone

Description automatically generated

* Identifying if the paths are definition clear

A screenshot of a cell phone

Description automatically generated

* Testing all du-paths test data

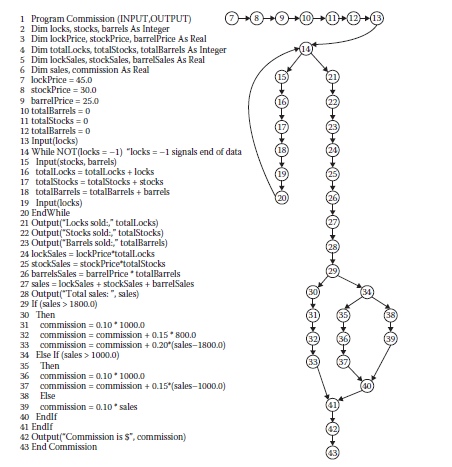
A screenshot of a cell phone

Description automatically generated

These are the steps that have to be performed for Data Flow testing. In your text book, specifically the dataflow testing for commission problem is considered so explaining the commission problem as well.

**Commission Problem:**

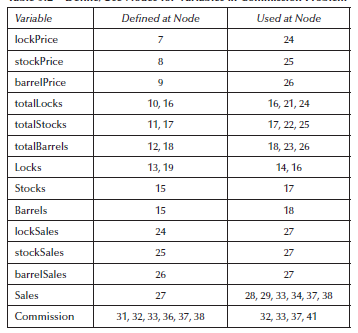
As the commission problem is already discussed in module 2, right away pasting the pseudo code and program graph as below:



**Steps that has to be performed:**

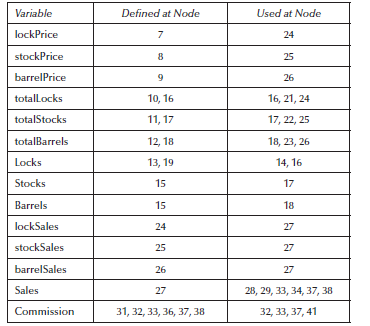
**Step 1:**

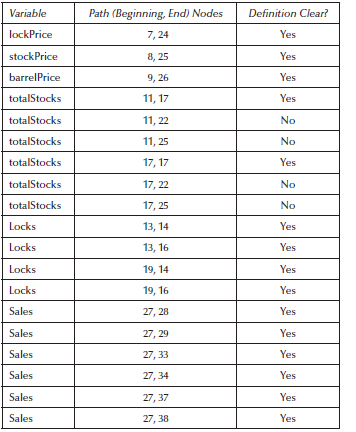
Identify all the variables and find where it is defined and where it is used. The tabulation below shows the step1 results for Commission problem.



**Step 2:**

**Identifying all the du-paths and dc-paths for all the variables:**

****

****

* **Tabulation shows only selected du-paths for the commission problem.**

**Understand the definition clear path properly:**

Let us consider the variable total locks. This variable consists of two defining nodes (DEF(totalLocks, 10) and DEF(totalLocks, 16)) and three usage nodes (USE(totalLocks,16), USE(totalLocks, 21), USE(totalLocks, 24)), we might expect six du-paths.

* Path p5 = <10, 11, 12, 13, 14, 15, 16> is a du-path in which the initial value of totalLocks (0) has a computation use. This path is definition clear.
* p6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21>
* Path p6 ignores the possible repetition of the while loop. We could highlight this by noting that the subpath <16, 17, 18, 19, 20, 14, 15> might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition clear.

Likewise for every variable, identify the paths which are definition clear and definition usage path. Based on these, the number of test cases varies; we can write test cases to test all du-paths, test cases to test all definitions, test cases only for definition clear.

**Define/Use Test coverage Metrics: (Rapps–Weyuker data flow metrics)**

In the following definitions, *T* is a set of paths in the program graph *G*(*P*) of a program *P*, with the set *V* of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths. This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

**Definition**

The set *T* satisfies the *All-Defs criterion* for the program *P* if and only if for every variable *v* ∈ *V*, *T*

contains definition-clear paths from every defining node of *v* to a use of *v*.

**Definition**

The set *T* satisfies the *All-Uses criterion* for the program *P* if and only if for every variable *v* ∈ *V*, *T* contains definition-clear paths from every defining node of *v* to every use of *v*, and to the successornode of each USE(*v*, *n*).

**Definition**

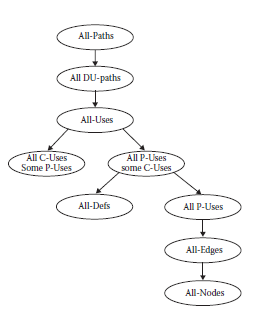
The set *T* satisfies the *All-P-Uses/Some C-Uses criterion* for the program *P* if and only if for every variable *v* ∈ *V*, *T* contains definition-clear paths from every defining node of *v* to every predicate use of *v*; and if a definition of *v* has no P-uses, a definition-clear path leads to at least one computation use.

**Definition**

The set *T* satisfies the *All-C-Uses/Some P-Uses criterion* for the program *P* if and only if for every variable *v* ∈ *V*, *T* contains definition clear paths from every defining node of *v* to every computation use of *v*; and if a definition of *v* has no C-uses, a definition-clear path leads to at least one predicate use.

**Definition**

The set *T* satisfies the *All-DU-paths criterion* for the program *P* if and only if for every variable *v* ∈ *V*, *T* contains definition-clear paths from every defining node of *v* to every use of *v* and to the successor node of each USE(*v*, *n*), and that these paths are either single loop traversals or they are cycle free.

****

**Note: The concept of dataflow testing is explained, you should be in a position to create test cases adopting data flow testing. Irrespective of whatever question been asked in exams w.r.to data flow testing make sure you write the definitions properly and proceed the steps for the given problem.**

**Slice Based Testing:**

* Another form of dataflow testing , again w.r.to variables is slice based testing.
* Slice is reduced form of a program that is also a executable program.

Let us look into the formal definition of slice.

Program *P* that has a program graph *G*(*P*) and a set of program variables *V*. Given a program *P* and a set *V* of variables in *P*, a *slice on the variable set V at statement n*, written *S*(*V*, *n*), is the set of all statement fragments in *P* that contribute to the values of variables in *V* at

node *n*.

**Basic Idea in Slice Based Testing:**

* Separate a program into components that has useful meaning.
* Slice captures the execution time behavior of a program w.r.to the variable in the slice.
* Eventually we will develop a lattice, in which nodes are slices and edges correspond to subset relationship.

**How to perform Slicing:?**

* All statements where variables are defined and redefined should be considered
* All statements where variables are receiving values externally should be considered
* All statements where output of a variable is printed should be considered
* The statement of all variables may be considered at last statement of a program.

***Example:***

*Let us under*stand the slice based testing with a concrete example:

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generated

Let us perform slicing for the above program.

The variable A is receiving value at line number 6 , so S(A,6) is the first slice.

**1.void main()**

**2.{**

**3.float A,B,C;**

**4.clrscr();**

**5.printf(“Enter the value for A:”);**

**6.scanf(“%f”,&A);**

**28.}**

The Slice S(A,6) itself is executable and its execution depends on the variable used.

Like wise the total number of slices that can be created for this example are as follows:

A close up of a sign

Description automatically generated

Above diagram shows the slices and the lines of statements that are part of the slice.

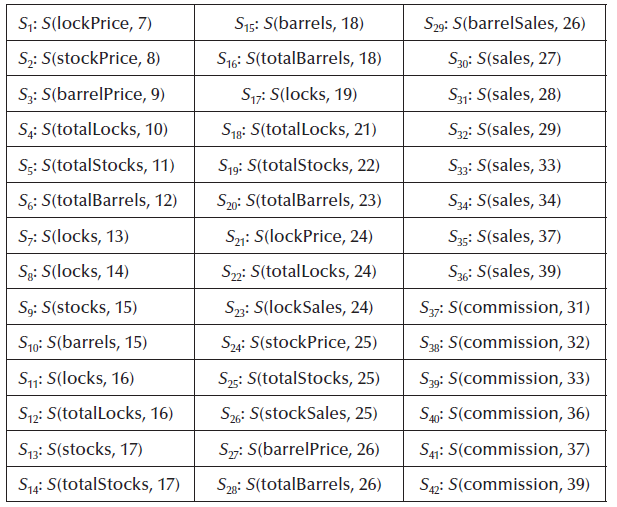
After the slices are generated, for each slice the test cases are created as below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S:No** | **Slice** | **Lines Covered** | **A** | **B** | **C** | **EO** |
| 1 | S(A,6) | 1-6,28 | 9 |  |  | No Output |
| 2 | S(A,13) | 1-14,18,27,28 | 9 | 8 | 7 | 9 |
| 3 | S(A,28) | 1-14,18,27,28 | 8 | 8 | 7 | 9 |
| 4 | S(B,8) | 1-4,7,8,28 |  | 9 |  | No output |
| 5 | S(B,24) | 1-11,18-20,22-28 | 7 | 9 | 8 | 9 |
| 6 | S(C,10) | 1-4,9,10,28 |  |  | 9 | No output |
| 7 | S(C,16) | 1-12,14-18,27,28 | 8 | 7 | 9 | 9 |
| 8 | S(C,21) | 1-11,18-22,26-28 | 7 | 8 | 9 | 9 |
| 9 | S(C,28) | 1-11,18-22,26-28 | 7 | 8 | 9 | 9 |

This is the procedure that has to be adopted for performing Slice Based Testing.

**Example 2: Commission Problem**

**The slices are as follows:**

****

Rapps and Weyuker (1985), but we need to refine these forms of variable usage. Specifically, the USE relationship pertains to five forms of usage:

P-use used in a predicate (decision)

C-use used in computation

O-use used for output

L-use used for location (pointers, subscripts)

I-use iteration (internal counters, loop indices)

Most of the literature on program slices just uses P-uses and C-uses. While we are at it, we identify two forms of definition nodes:

I-def defined by input

A-def defined by assignment

***S*1: *S*(lockPrice, 7) = {7}**

***S*2: *S*(stockPrice, 8) = {8}**

***S*3: *S*(barrelPrice, 9) = {9}**

***S*4: *S*(totalLocks, 10) = {10}**

***S*5: *S*(totalStocks, 11) = {11}**

***S*6: *S*(totalBarrels, 12) = {12}**

Slices 7 through 17 focus on the sentinel controlled while loop in which the totals for locks,

stocks, and barrels are accumulated.

The locks variable has two uses in this loop: a P-use at fragment 14 and C-use at statement 16.

It also has two defining nodes, at statements 13 and 19.

The stocks and barrels variables have a defining node at 15, and computation uses at nodes 17 and 18, respectively. Notice the presence of all relevant statement fragments in slice 8.

***S*7: *S*(locks, 13) = {13}**

***S*8: *S*(locks, 14) = {13, 14, 19, 20}**

***S*9: *S*(stocks, 15) = {13, 14, 15, 19, 20}**

***S*10: *S*(barrels, 15) = {13, 14, 15, 19, 20}**

***S*11: *S*(locks, 16) = {13, 14, 19, 20}**

***S*12: *S*(totalLocks, 16) = {10, 13, 14, 16, 19, 20}**

***S*13: *S*(stocks, 17) = {13, 14, 15, 19, 20}**

***S*14: *S*(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}**

***S*15: *S*(barrels, 18) = {12, 13, 14, 15, 19, 20}**

***S*16: *S*(totalBarrels, 18) = {12, 13, 14, 15, 18, 19, 20}**

***S*17: *S*(locks, 19) = {13, 14, 19, 20}**

Slices 18, 19, and 20 are output statements, and none of the variables is defined; hence, the corresponding statements are not included in these slices.

***S*18: *S*(totalLocks, 21) = {10, 13, 14, 16, 19, 20}**

***S*19: *S*(totalStocks, 22) = {11, 13, 14, 15, 17, 19, 20}**

***S*20: *S*(totalBarrels, 23) = {12, 13, 14, 15, 18, 19, 20}**

Slices 21 through 30 deal with the calculation of the variable sales. As an aside, we could simply write

***S*30: *S*(sales, 27) = *S*23 ∪ *S*26 ∪ *S*29 ∪ {27}.**

Slice *S*23 computes the total lock sales, *S*25 the total stock sales,

and *S*28 the total barrel sales. In a bottom–up way, these slices could be separately coded and tested, and later spliced together.

***S*21: *S*(lockPrice, 24) = {7}**

***S*22: *S*(totalLocks, 24) = {10, 13, 14, 16, 19, 20}**

***S*23: *S*(lockSales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}**

***S*24: *S*(stockPrice, 25) = {8}**

***S*25: *S*(totalStocks, 25) = {11, 13, 14, 15, 17, 19, 20}**

***S*26: *S*(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}**

***S*27: *S*(barrelPrice, 26) = {9}**

***S*28: *S*(totalBarrels, 26) = {12, 13, 14, 15, 18, 19, 20}**

***S*29: *S*(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}**

***S*30: *S*(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}**

Slices 31 through 36 are identical. Slice *S*31 is an O-use of sales; the others are all C-uses. Since none of these changes the value of sales defined at *S*30, we only show one set of statement fragment numbers here.

***S*31: *S*(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}**

The last seven slices deal with the calculation of commission from the value of sales. This is literally where it all comes together.

***S*37: *S*(commission, 31) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,30, 31}**

***S*38: *S*(commission, 32) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,30, 31, 32}**

***S*39: *S*(commission, 33) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,30, 31, 32, 33}**

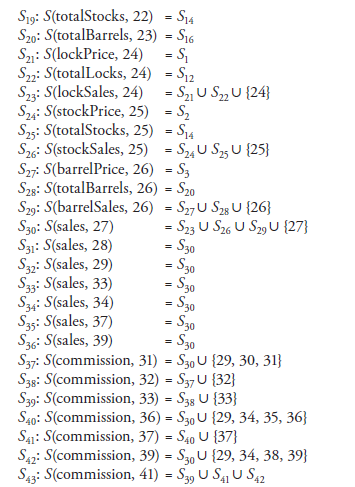
***S*40: *S*(commission, 36) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,34, 35, 36}**

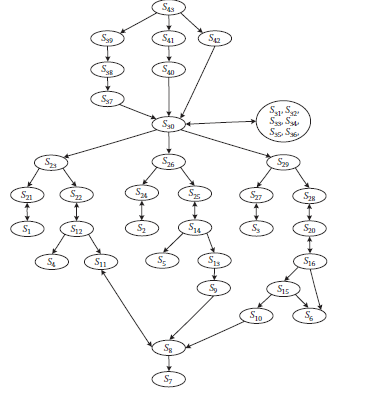
***S*41: *S*(commission, 37) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,34, 35, 36, 37}**

***S*42: *S*(commission, 39) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29,34, 38, 39}**

***S*43: *S*(commission, 41) = { 7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30,31, 32, 33, 34, 35, 36, 37, 39}**

**Upon the way the slices are combines, the lattice structure can be formed as follows:**

****

****

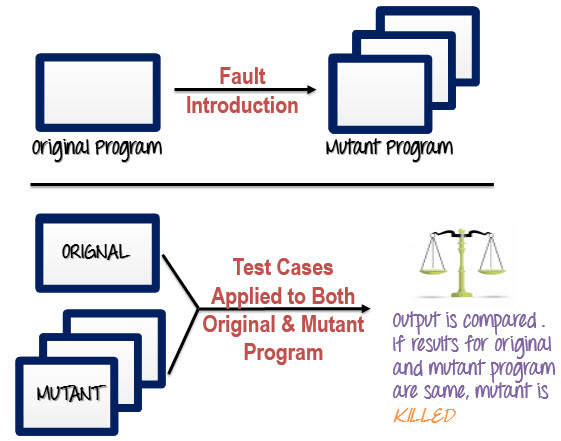
**Fault Based Testing:**

**Why fault based testing?**

* Nothing teaches more than failures and one can become successful if the reasons for failure are evaluated properly.
* Testing is no exception to this and if a tester wants to evaluate the effectiveness of the test suite he/she has considered, fault based testing is a way out to identify the answer.
* Experience with common software faults sometimes leads to improvements in design methods and programming languages. For eg: Automatic Memory Management in Java is not to spare the programmer the trouble of releasing unused memory, but to prevent the programmer from making the kind of memory management errors (dangling pointers, redundant deallocations, and memory leaks) that frequently occur in C and C++ programs.

**What is Fault Based Testing?**

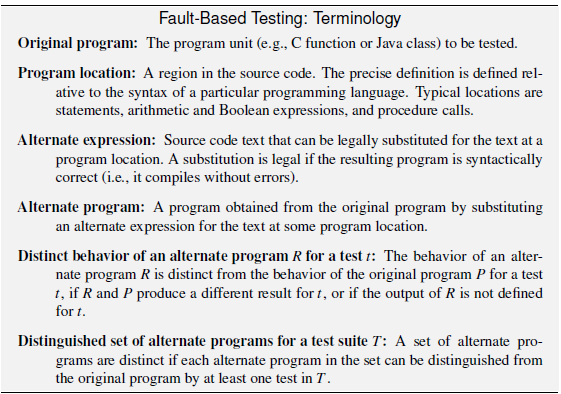
* Mutation analysis is the most common form of software fault-based testing.
* A fault model is used to produce hypothetical faulty programs by creating variants of the program under test.
* Variants are created by “seeding” faults, that is, by making a small change to the program under test following a pattern in the fault model.
* The patterns for changing program text are called mutation operators, and each variant program is called a mutant.

****

**Assumptions in Fault Based Testing:**

1. The effectiveness of fault-based testing depends on the quality of the fault model and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice, the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program. This is framing **competent programmer hypothesis**.
2. Some program faults are indeed simple typographical errors, and others that involve deeper errors of logic may nonetheless be manifest in simple textual differences. Sometimes, though, an error of logic will result in much more complex differences in program text. This may not invalidate fault-based testing with a simpler fault model, provided test cases sufficient for detecting the simpler faults are sufficient also for detecting the more complex fault. This is known as the **coupling effect.**

**Terminologies in Fault Based Testing:**

****

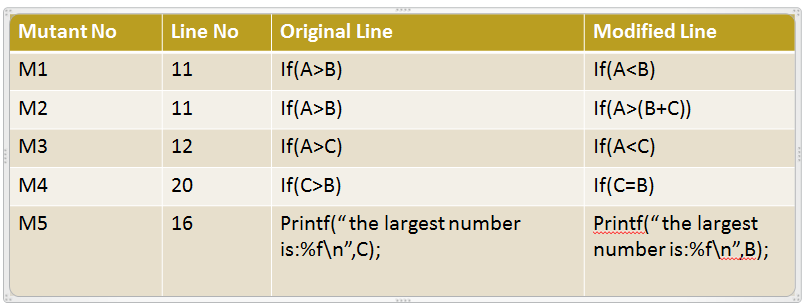
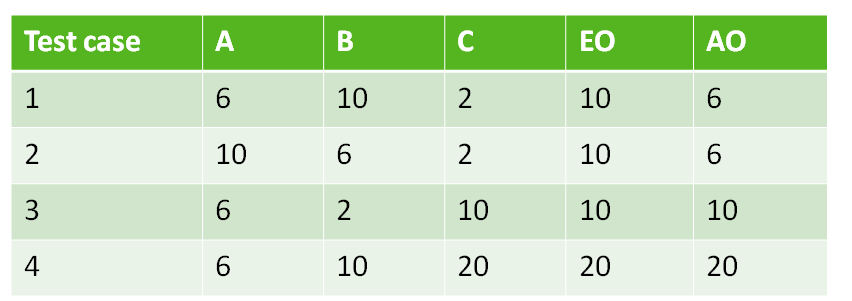
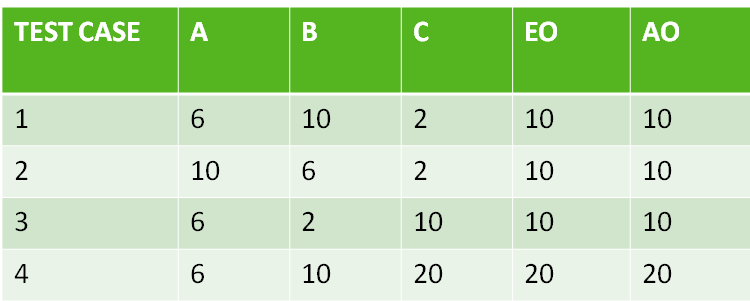
**Mutation Analysis:**

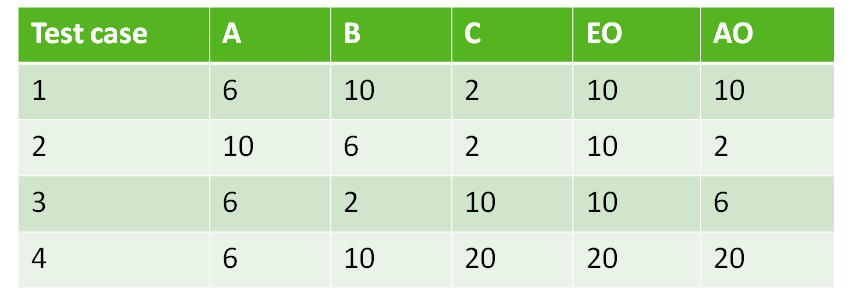
* Mutation analysis is the most common form of software fault-based testing.
* A fault model is used to produce hypothetical faulty programs by creating variants of the program under test.
* Variants are created by “seeding” faults, that is, by making a small change to the program under test following a pattern in the fault model.
* The patterns D mutation for changing program text is called mutation operators, and each variant program is operator called a mutant.

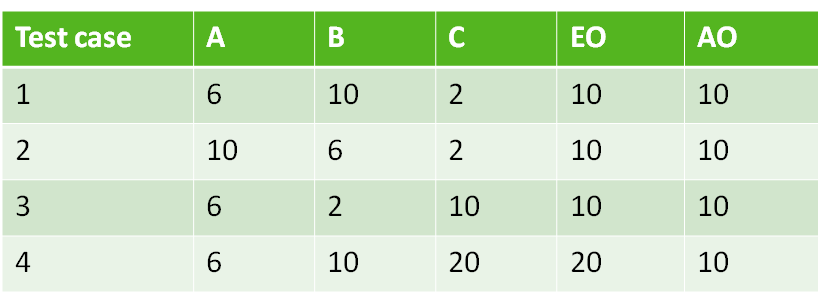
**Example:**

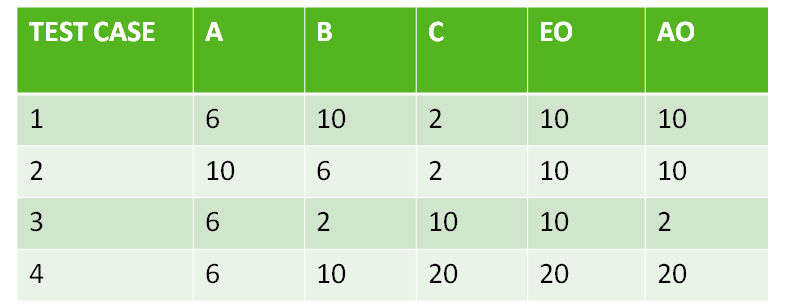
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S:No** | **A** | **B** | **C** | **Expected Output** |
| **1** | **6** | **10** | **2** | **10** |
| **2** | **10** | **6** | **2** | **10** |
| **3** | **6** | **2** | **10** | **10** |
| **4** | **6** | **10** | **20** | **20** |

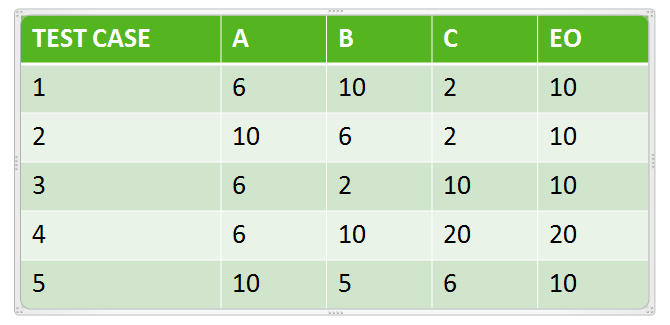
**Assume that the test suite selected for checking the greatest of three numbers program as below:**

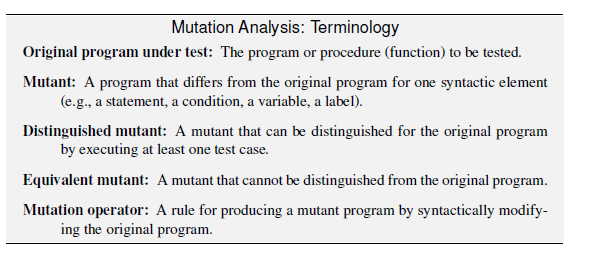
* **Now we need to measure the effectiveness of the test suite we have selected for which we will perform mutation analysis by creating mutants by introducing mutation operators as below:**
* ****
* This input value is then fed up after the changes is been made and the outputs are observed for mutant1,2,3,4 and 5 as below:
* ****
* ****

****

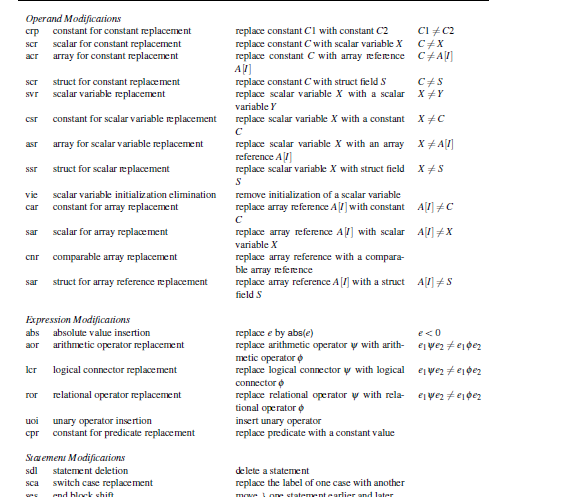
****

****

* Now if we observe the actual and expected output of all the 5 mutants only the mutant 2 matches so mutant 2 is killed …remaining 4 mutants are alive tat means we need to add more test cases to increase the effectiveness of the suite the tester has selected. Mutation score of 1 denotes the suite is good.
* **The test cases that has to be added are as follows:**
* ****
* This produces a mutation score of 4/5 ( no of mutant alive/Total number of mutants )
* This explains the concept of fault based testing, however there are some terminologies associated.



**Sample set of Mutation Operators:**



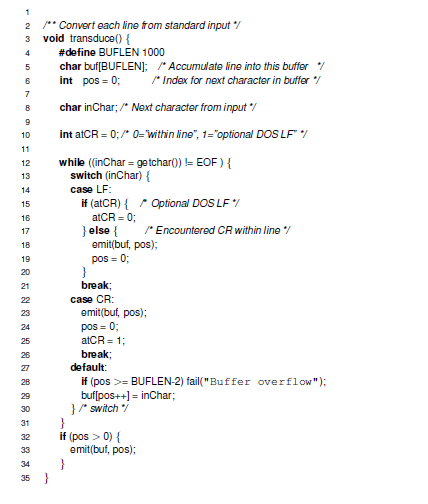
**Fault Based Adequacy Criteria:**

Given a program and a test suite T, mutation analysis consists of the following steps:

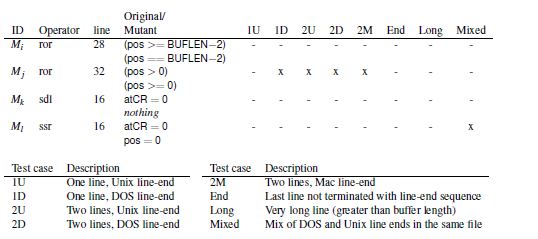
**Select mutation operators:** If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

**Generate mutants**: Mutants are generated mechanically by applying mutation operators to the original program.

**Distinguish mutants**: Execute the original program and each generated mutant with the test cases in T. A mutant is killed when it can be distinguished from the original program.



For this program, the mutants formed are



**Test suite TS**

**TS = {1U,1D,2U,2D,2M,End,Long}**

kills Mj, which can be distinguished from the original program by test cases 1D, 2U,

2D, and 2M.

Mutants Mi, Mk, and Ml are not distinguished from the original program by any test in TS. We say that mutants not killed by a test suite are live.

A mutant can remain live for two reasons:

* The mutant can be distinguished from the original program, but the test suite T does not contain a test case that distinguishes them
* The mutant cannot be distinguished from the original program by any test case
* Given a set of mutants SM and a test suite T, the fraction of nonequivalent mutants killed by T measures the adequacy of T with respect to SM.
* Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not or by counting some equivalent mutants among the remaining live mutants.
* The adequacy of the test suite TS evaluated with respect to the four mutants of Figure above is 25%.
* However, we can easily observe that mutant Mi is equivalent to the original program (i.e., no input would distinguish it).
* Conversely, mutants Mk and Ml seem to be nonequivalent to the original program: There should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of TS, measured after eliminating the equivalent mutant Mi, is 33%.
* Mutant Ml is killed by test case Mixed, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines.
* We would expect that Mixed would also kill Mk, but this does not actually happen: Both Mk and the original program produce the same result for Mixed. This happens because both the mutant and the original program fail in the same way.1
* The use of a simple oracle for checking the correctness of the outputs (e.g., checking each output against an expected output) would reveal the fault. The test suite TS2 obtained by adding test case Mixed to TS would be 100% adequate (relative to this set of mutants) after removing the fault.

**Note: Go through ppts if problem in understanding and also go thorugh the classwork we have done numerous example problems.**