UNIT 3: White Box Testing

Introduction, Need of white box testing, Testing types, Static testing by humans, Structural Testing – Control flow testing, Loop Testing, Design of test cases, Challenges in White box testing, Case-studies of White-Box testing.

WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

Basic Concepts

A white-box testing strategy can either be coverage-based or fault- based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

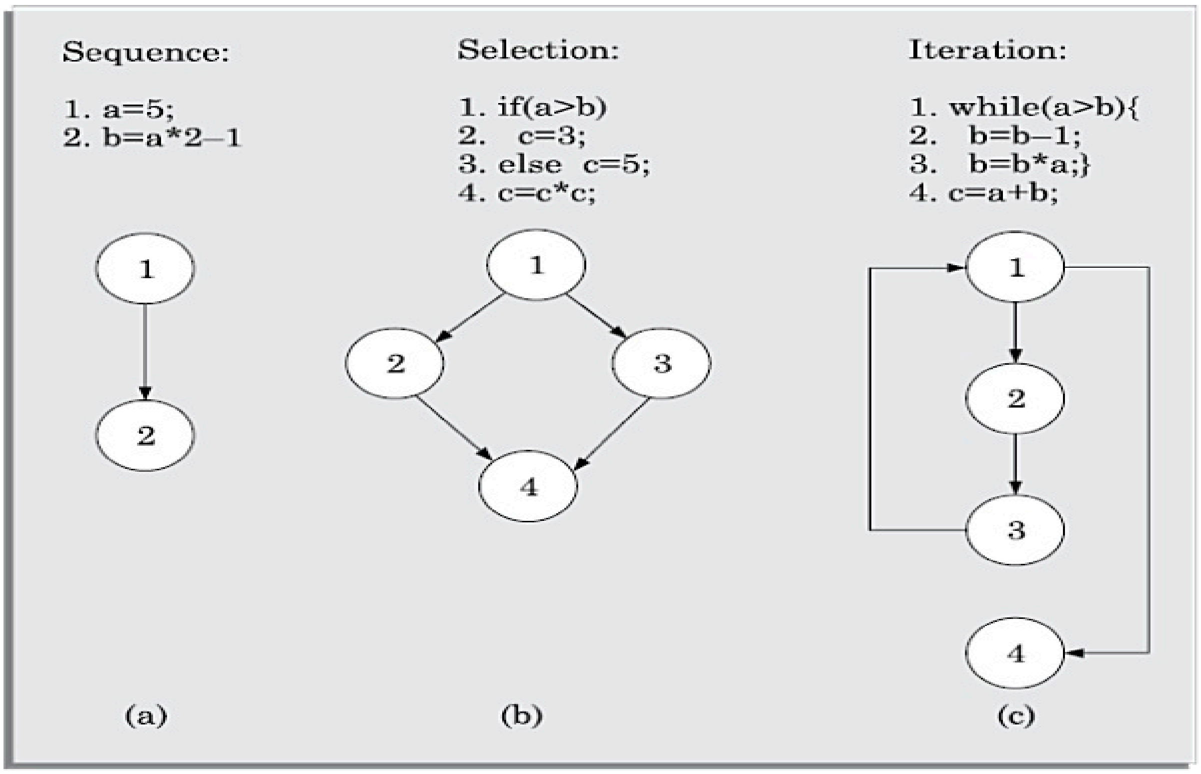
For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other

A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure . Observe in Figure (a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure (b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.



If a stronger testing has been performed, then a weaker testing need not be carried out.

A test suite should, however, be enriched by using various complementary testing strategies.

We need to point out that coverage-based testing is frequently used to check the quality of testing achieved by a test suite. It is hard to manually design a test suite to achieve a specific coverage for a non-trivial program.

Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.

It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.

Example : Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
int x,y;
{
1 while (x != y){
2 if (x>y) then
3 x=x-y;
4 else y=y-x;
5 }
6 return x;
}
```

Answer: To design the test cases for the statement coverage, the conditional expression of the while statement needs to be made true and the conditional expression of the if statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, all statements of the program would be executed at least once.

Branch Coverage

A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.

Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Example

For the following program , determine a test suite to achieve branch coverage.

```
int computeGCD(x,y)
int x,y;
{
1 while (x != y){
2 if (x>y) then
3 x=x-y;
4 else y=y-x;
5 }
6 return x;
}
```

Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage.

It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Theorem: Branch coverage-based testing is stronger than statement coverage-based testing.

Proof: We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.

(a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).

(b) To show that statement coverage does not ensure branch coverage, it would be sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

```
if(x>2) x+=1;
```

The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition (x > 2) is not made false by any test case in the suite.

Multiple Condition Coverage

In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Example : Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.

Answer: Consider the following C program segment:

```
if(temperature>150 || temperature>50)
setWarningLightOn();
```

The program segment has a bug in the second component condition, it should have been temperature<50. The test suite {temperature=160, temperature=40} achieves branch coverage. But, it is not able to check that setWarningLightOn(); should not be called for temperature values within 150 and 50

Path Coverage

A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

A control flow graph describes how the control flows through the program.

We can define a control flow graph as the following:

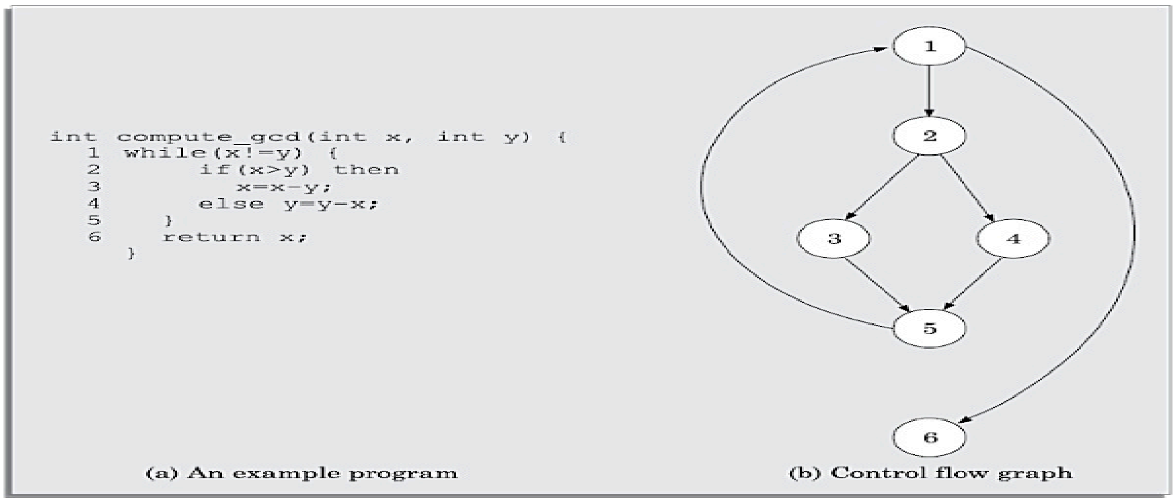
A control flow graph describes the sequence in which the different instructions of a program get executed.

In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph . There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.

We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only. Figure summarises how the CFG for these three types of constructs can be drawn. The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop(iteration) construct can be drawn. For iteration type of constructs such as the

while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop. Using these basic ideas, the CFG of the program given in Figure (a) can be drawn as shown in Figure (b).

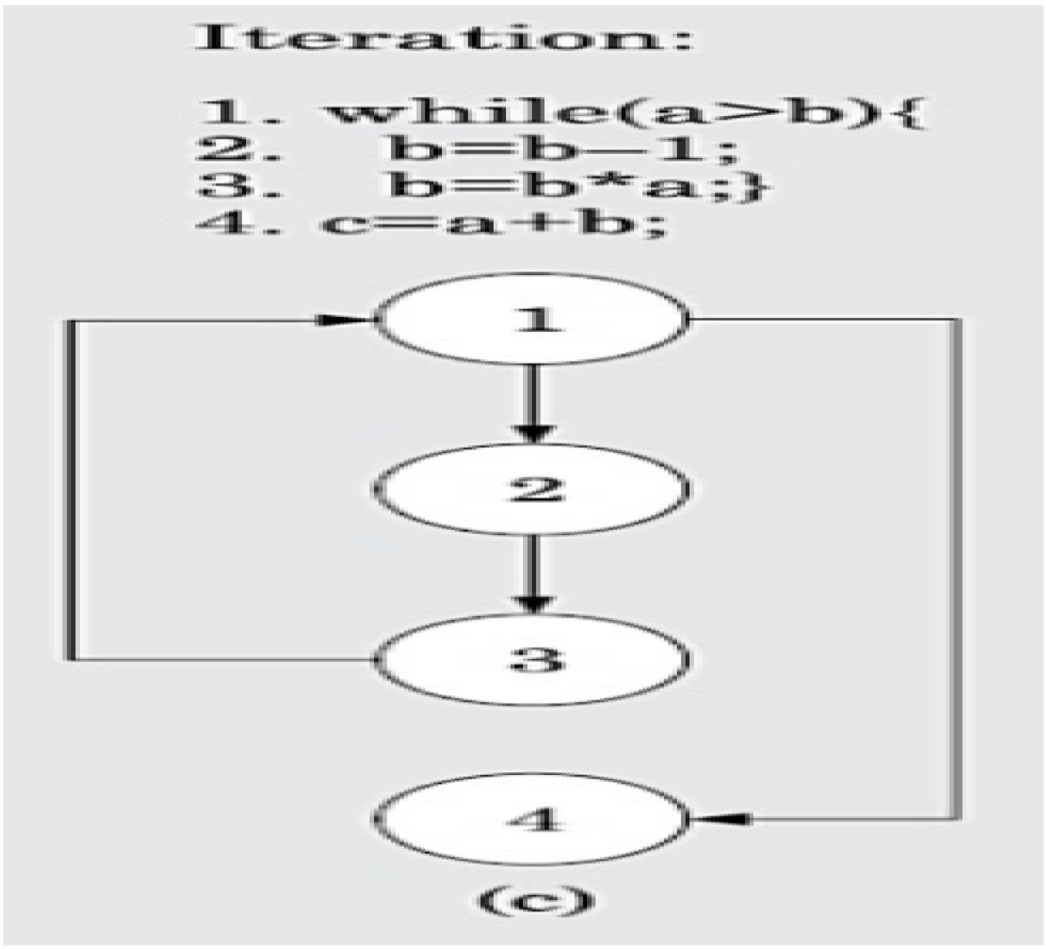


```
int compute_gcd(int x, int y) {
  1  while(x!=y) {
  2       if(x>y) then
  3          x=x-y;
  4       else y=y-x;
  5     }
  6     return x;
    }
```

(a) An example program (b) Control flow graph

Path

A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Note that a program can have more than one terminal nodes when it contains multiple exit or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops. For example, in Figure (c), there can be an infinite number of paths



```
Iteration:
1. while(a>b){
2.    b=b—1;
3.    b=b*a;}
4. c=a+b;
```

(c)

such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (o r basis paths). Let us now discuss what are linearly independent paths and how to determine these in a program.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because, any path having a new node would automatically have a new edge. An alternative definition of a linearly independent set of paths [McCabe76] is the following:

If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.

According to the above definition of a linearly independent set of paths, for any path in the set, its subpath cannot be a member of the set. In fact, any arbitrary path of a program, can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the “basis” for all the paths of a program. Please note that there may not always exist a unique basis set for a program and several basis sets for the same program can usually be determined.

Even though it is straight forward to identify the linearly independent paths for simple programs, for more complex programs it is not easy to determine the number of independent paths. In this context, McCabe’s cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe’s cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe’s cyclomatic complexity is very simple to compute. Though the McCabe’s metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.

McCabe’s Cyclomatic Complexity Metric

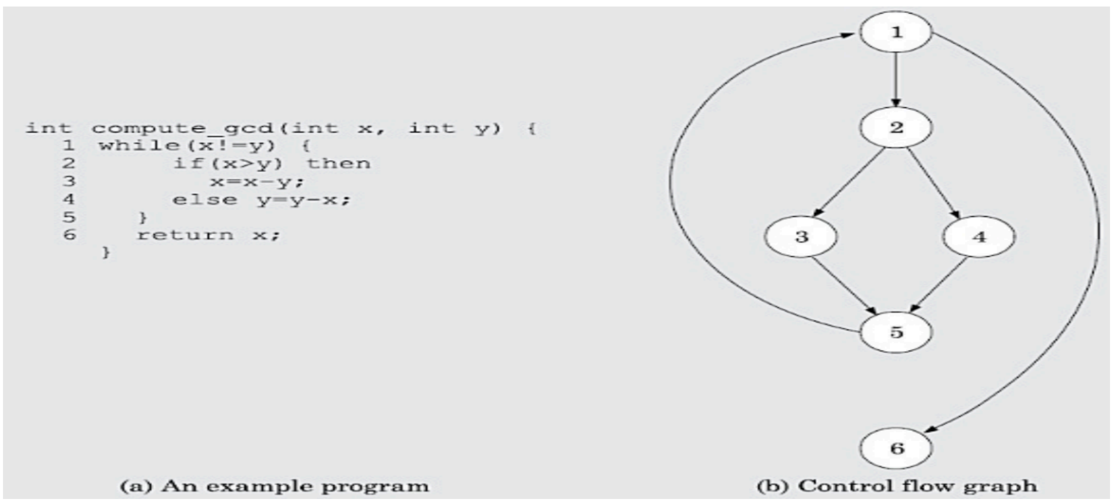
McCabe obtained his results by applying graph-theoretic techniques to the control flow graph ofa program. McCabe’s cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$V(G) = E – N + 2$

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure



```
int compute_gcd(int x, int y) {
 1  while(x!=y) {
 2      if(x>y) then
 3         x=x-y;
 4      else y=y-x;
 5    }
 6    return x;
  }
```

(a) An example program (b) Control flow graph

E = 7 and N = 6. Therefore,

the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

Method 2: An alternate way of computing the cyclomatic complexity of a

program is based on a visual inspection of the control flow graph is as follows

—In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:

V(G) = Total number of non-overlapping bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar (i.e., how ever you draw the graph, two or more edges always intersect). Actually,

it can be shown that control flow representation of structured programs always yields planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity does not apply.

The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program. Consider the CFG example shown in Figure . From a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computed with this method is also 2+1=3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the method for computing CFGs can easily be automated. That is, the McCabe's metric computations methods 1 and 3 can be easily coded into a program

that can be used to automatically determine the cyclomatic complexities of arbitrary programs.

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe's metric is equal to N + 1.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. repeat

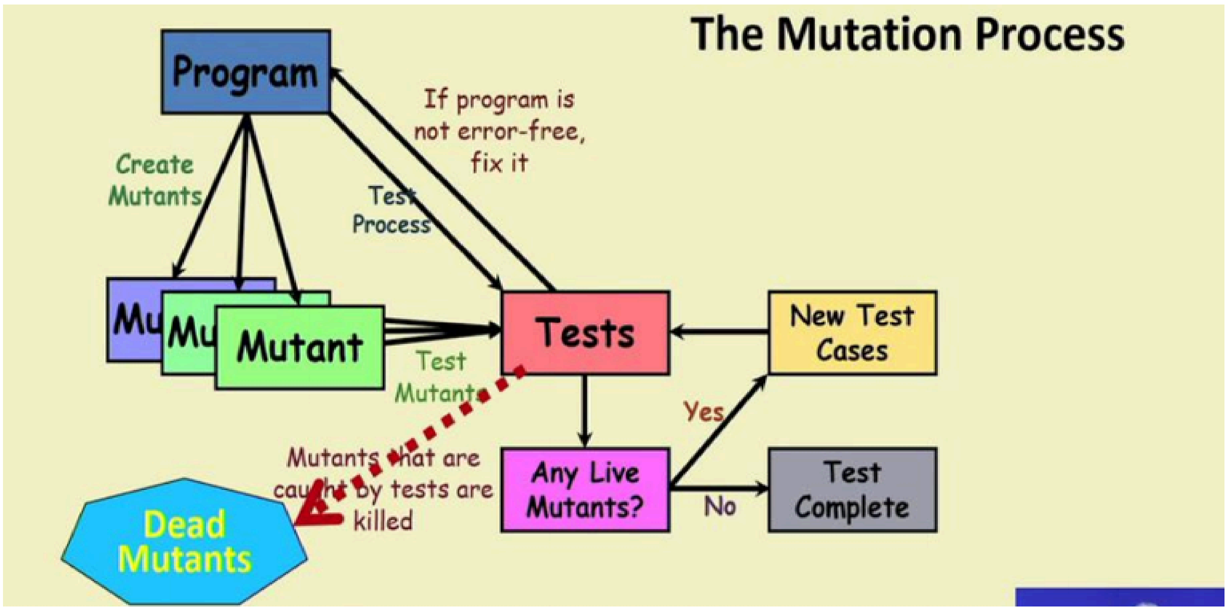
Test using a randomly designed set of test cases.

Perform dynamic analysis to check the path coverage achieved. until at least 90 per cent path coverage is achieved.

Mutation Testing

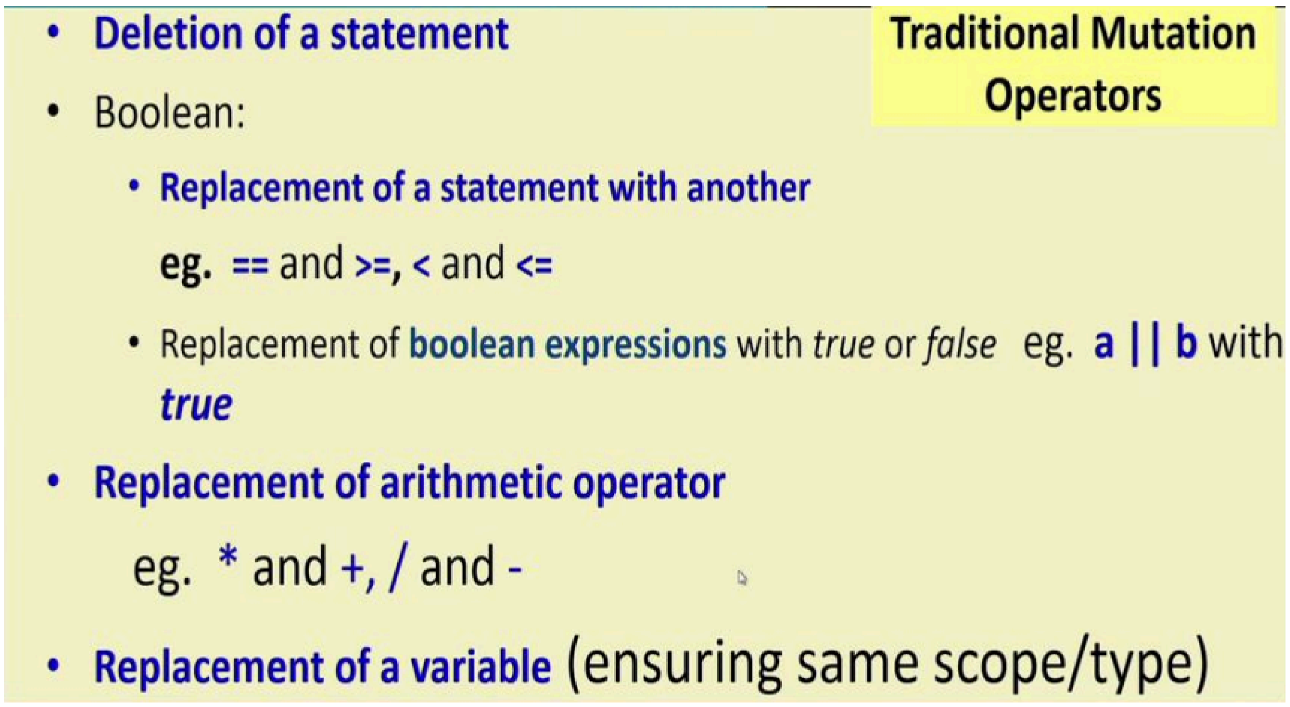
All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors. A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement. A mutant may or may not cause an error in the program. If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.



Block/Flow Diagram of Mutation Testing

A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant. However, it is not this straightforward. Remember that there is a possibility of a mutated program to be an equivalent program. When this is the case, it is futile to try to design a test case that would identify the error.



Traditional Mutation Operators

- Deletion of a statement
- Boolean:
  - Replacement of a statement with another
    eg. == and >=, < and <=
  - Replacement of boolean expressions with *true* or *false* eg. a || b with *true*
- Replacement of arithmetic operator
  eg. * and +, / and -
- Replacement of a variable (ensuring same scope/type)

Example:

```
Original Program
If (x>y)
Print "Hello"
Else
Mutant Program
If(x<y)
Print "Hello"
Else
Print "Hi"

If(x>y) replace x and y values
If(5>y) replace x by constant 5


If(x>y) replace x and y values

If(5>y) replace x by constant 5



                  ”
Print "Hi
```

If(x==y)

We can replace == into >= and have mutant program as

If(x>=y) and inserting ++ in the statement

If(x==++y)

An important advantage of mutation testing is that it can be automated to a great extent. The process of generation of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be simple program alterations such as—deleting a statement, deleting a variable definition, changing the type of an arithmetic operator (e.g., + to -), changing a logical operator (and to or) changing the value of a constant, changing the data type of a variable, etc. A major pitfall of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Mutation testing involves generating a large number of mutants. Also each mutant needs to be tested with the full test suite. Obviously therefore, mutation testing is not suitable for manual testing. Mutation testing is most suitable to be used in conjunction of some testing tool that should automatically generate the mutants and run the test suite automatically on each mutant. At present, several test tools are available that automatically generate mutants for a given program.

Disadvantages of Mutation Testing

Computationally very expensive.

A large number of possible mutants can be generated.

Certain types of faults are very difficult to inject.

Only simple syntactic faults introduced

Challenges in white box testing

1) Lack of Understanding Of Programming Language(S) Used For Testing:

We have seen too many testers who are not bothered to understand how the programs work or what is happening inside when they test an application or a website. To be precise, We are sure most of them do not know what is happening behind the scene when they are trying to test a website or an application. The majority of them are only bothered to do the testing to prove whether it works for them or not.

2) Lack Of Understanding The Logical Flow/Use Case :

Even if the tester understands how the program works or what is happening behind the scene, they may not be able to understand the logical flow of the program. Therefore, the tester should always try to analyze how things are connected earlier than White Box Testing them one by one.

3) Lack Of Patience To Go Thorough The Program:

Even if you have understood everything, made a use case and analyzed the logical flow, you might still not be able to go through with the program if you do not have patience. The majority of testers feel that they should test all the modules in a couple of hours to take out time for other things.

4) The Reality Of Testing:

We know this is an irony, but software testing services are something no one can be assured of getting right. There is always a chance that the program may not work as we expect it to work. Testing means proving the program works before releasing it to clients/customers for further use.

5) Copying The Existing Functionality:

We have seen many software testers copying the existing functionalities in the program; they add a few more things when they think some functionality is missing. This is because they want to satisfy their managers by completing the wish list they have asked for.

6) Allowing Clients/Managers To Influence:

Most of us do not like it when we are influenced by clients/managers. The majority of the testers give up their views and accept what clients or managers say. They want to maintain good relations with them so that their job will not be jeopardized.

7) Lack Of Patience:

We know this is again an irony, but most of us do not have patience. We want to get everything done quickly because we may feel that the client/manager will not allow us to take any further actions.

8) Not Being Honest Enough With The Client:

Most testers are always afraid to speak out their minds while White Box Testing an application or a website for clients/customers because they might lose their job if it goes wrong. Instead, they want to keep the client happy by showing them something is not happening.

9) Spending Long Hours In Testing:

We know many testers feel that they should spend long hours in testing because it will help them be more accurate. However, most of the time, testers want to avoid doing testing and spend time surfing swine on the internet.

10) Stress:

Last but not least, software testers usually suffer from stress in their life when they see something is going wrong in the application or the website. So they want to prove themselves efficient by running behind it 24×7, trying to find out what exactly happened.

To be precise, the majority of the testers are not bothered about what is happening behind the scene when they are testing anything. Instead, they want to go through with testing in a quick time because several things need their attention.

Questions

1. **At least how many test cases needed to achieve multiple condition coverage of the following code segment, Describe required test cases.**

```
If((a> 5) and (b<100) and (c>50))
x=x+1
```

2. **Consider the following program code. At least how many test cases are needed for basic condition coverage? Justify your answer with required test cases?**

```
int find_m(int i, int j, int k){
int m;
if(i>j) then m=j+k;
if(i>k) and (k<100) then m=i;
else m=k;
return(m);
}
```

3. **Consider the following program code. At least how many test cases are needed for basic condition coverage? Justify your answer with required test cases?**

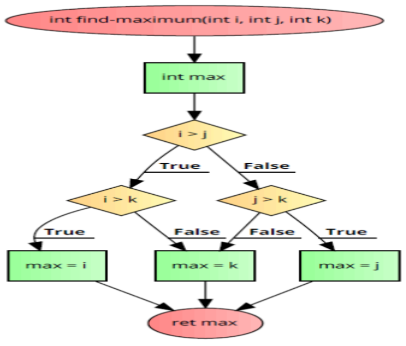
```
int find_m(int i, int j, int k){
int m;
if(i>j) then m=j+k;
if(i>k) and (k<100) then m=i;
else m=k;
return(m);
}
```

4. **Draw the CGF for the following function.**

Find Cyclomatic complexity ?

```
int find_m(int i, int j, int k){
int m;
if(i>j) then m=j+k;
if(i>k) and (k<100) then m=i;
else m=k;
return(m);
}
```

- Explanation: Using the control flow graph shown below, we compute Cyclomatic complexity for the given program using the formula E - N + 2 as (11 - 9 + 2) i.e. 4.



5. **If MC/DC coverage has been achieved on a unit under test, Statement coverage and Condition/decision coverage test coverage are implicitly implied. Justify?**

6. **For the following program statement, Design test suite to achieve basic condition/decision coverage. Justify your answer?**

```
if (a>50 || b<50)
   p++;
```

7. **At least how many test cases are required to achieve modified condition/decision coverage (MC/DC) of the following code segment:**

```
if((a>15) and (b<100) and (c>55))
  x=x*y+100;
```

   Justify your answer?

8. **Write 5 test cases which will achieve condition coverage for following code statement**
   If (a>10 && b< 50)
9. **Write test cases to achieve branch coverage for following function**

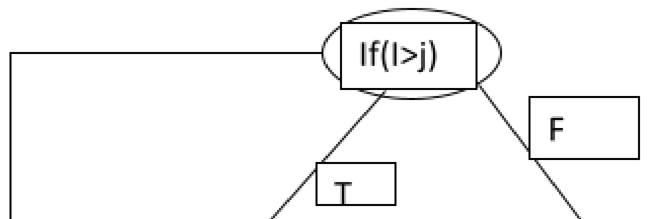
```
Int f(int x,int y)
{
while(x!=y){
If (x>y) then
x=x-y;
else y=y-x;
}
return x; }
```

10. **Demonstrate decision/Branch coverage with suitable example? Give formula to calculate coverage.**
11. **Illustrate Multiple condition coverage with suitable example?why it is practically not feasible?**
12. **Demonstrate process of mutation testing with diagram?**
13. Explain condition coverage with suitable example?Give formula to calculate coverage.
14. Demonstrate white box testing? List the various white box testing strategies.
15. Explain Modified Condition Decision coverage (MC/DC)with suitable example?

Explain statement coverage with suitable example?Give formula to calculate coverage.
Does basic condition coverage subsume decision coverage? Justify your answer with example.
What do you understand from short circuit evaluation, explain with suitable example?
Given that 1. Decision /decision coverage, 2. condition coverage ,3. Statement coverage, 4. Multiple condition coverage(MCC), 5. MC/DC
Arrange above coverage from strongest(first) coverage to weakest coverage(last)
Expalin data flow- based testing? Give example of DU Chain.
Describe coverage based and fault based white box testing strategies? Give examples.
Why Black-box and white-box testing are necessary?
Why is the all path testing strategy rarely used in practice? Justify.
Write test suit which will achieve Multiple Condition coverage for following code statement
If (a>50 OR b< 30)

Sketch the CFG for the following function and Find Cyclomatic complexity ?

```
int find_m(int i, int j, int k){
int m;
if(i>j) then m=j+k;
if(i>k) and (k<100) then m=i;
else m=k;
return(m);}
```



Nodes (N):

- Start
- Node 1 (if `i > j`)
- Node 2 (set `m = j + k`)
- Node 3 (if `i > k && k < 100`)
- Node 4 (set `m = i`)
- Node 5 (set `m = k`)
- Node 6 (return `m`)
- End

Total Nodes, N=8N = 8N=8

Edges (E):

- Start → Node 1
- Node 1 → Node 2 (if true)
- Node 1 → Node 3 (if false)
- Node 3 → Node 4 (if true)
- Node 3 → Node 5 (if false)
- Node 2 → Node 6
- Node 4 → Node 6
- Node 5 → Node 6
- Node 6 → End

Total Edges, E=7

Calculating Cyclomatic Complexity

Using the values we found Cyclomatic complexity
`CC = E - N + 2 =` 7−6+2=3