

Module 10

Coding and Testing

Lesson 25

White-Box Testing

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- In the context of white box testing strategy, differentiate between stronger testing and complementary testing.
- Design statement coverage test cases for a code segment.
- Design branch coverage test cases for a code segment.
- Design condition coverage test cases for a code segment .
- Design path coverage test cases for a code segment.
- Draw control flow graph for any program.
- Identify the linear independent paths.
- Compute cyclomatic complexity from any control flow graph.
- Explain data flow-based testing.
- Explain mutation testing.

White box testing

One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated in fig. 10.2.

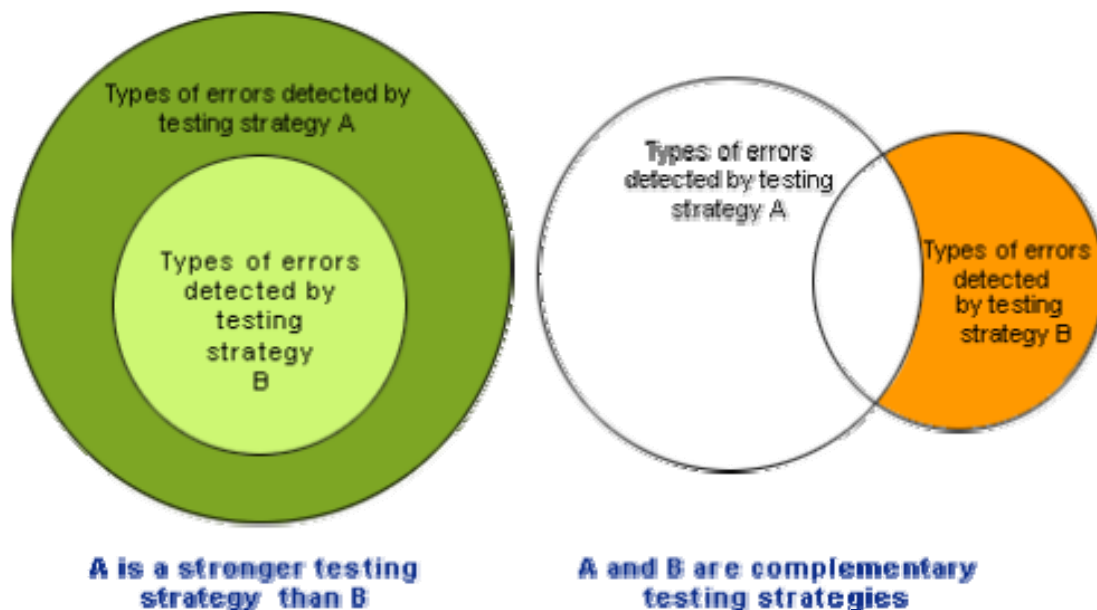


Fig. 10.2: Stronger and complementary testing strategies

Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm:

```
int compute_gcd(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;
    }
```

By choosing the test set $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$, we can exercise the program such that all statements are executed at least once.

Branch coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed at least once.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid's GCD computation algorithm, the test cases for branch coverage can be $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$.

Condition coverage

In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression $((c1.and.c2).or.c3)$, the components $c1$, $c2$ and $c3$ are each made to assume both true and false values. Branch testing is

probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Path coverage

The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 10.3). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 10.3 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in fig. 10.4.

Sequence:

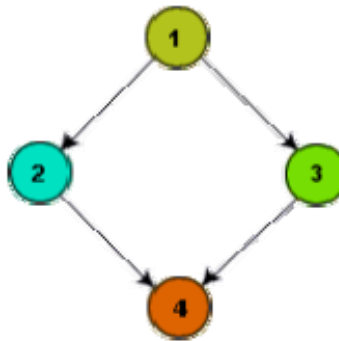
1. a=5;
2. b=a^2-1;



(a)

Selection:

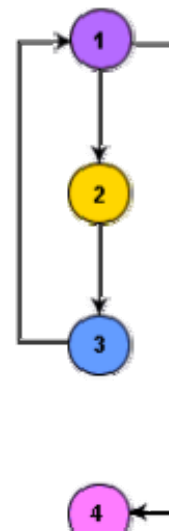
1. if(a>b)
2. c=3;
3. else c=5;
4. c=c^c;



(b)

Iteration:

1. while(a>b){
2. b=b-1;
3. b=b^a;}
4. c=a+b;



(c)

Fig. 10.3: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs

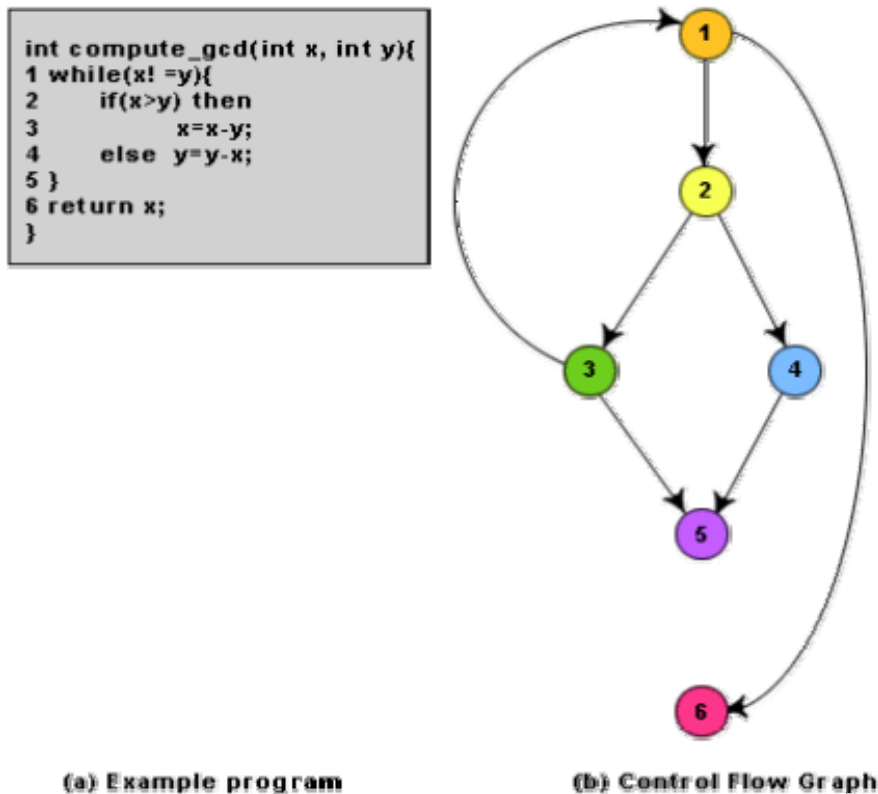


Fig. 10.4: Control flow diagram

Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Control flow graph

In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent path

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the 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. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. The answers computed by 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 fig. 10.4, $E=7$ and $N=6$. Therefore, the cyclomatic complexity = $7-6+2 = 3$.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

$$V(G) = \text{Total number of bounded areas} + 1$$

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. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield 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 cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in fig. 10.4, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing 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 other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

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

Data flow-based testing

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

For a statement numbered S , let

DEF(S) = {X/statement S contains a definition of X}, and
USES(S) = {X/statement S contains a use of X}

For the statement $s: a=b+c;$, DEF(S) = {a}. USES(S) = {b,c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

The *definition-use chain* (or DU chain) of a variable X is of form [X, S, S1], where S and S1 are statement numbers, such that $X \in \text{DEF}(S)$ and $X \in \text{USES}(S1)$, and the definition of X in the statement S is live at statement S1. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

Mutation testing

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.