Chapter 5

Dynamic Testing: White-Box Testing Techniques

OBJECTIVES

After reading this chapter, you should be able to understand:

- White-box testing demands complete understanding of the program logic/structure
- Test case designing using white-box testing techniques
- Basis path testing method
- Building a path testing tool using graph matrices
- Loop testing
- Data flow testing method
- Mutation testing method

White-box testing is another effective testing technique in dynamic testing. It is also known as *glass-box* testing, as everything that is required to implement the software is visible. The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as *structural* or *development testing*.

In white-box testing, structure means the logic of the program which has been implemented in the language code. The intention is to test this logic so that required results or functionalities can be achieved. Thus, white-box testing ensures that the internal parts of the software are adequately tested.

5.1 Need of White-box testing

Is white-box testing really necessary? Can't we write the code and simply test the software using black-box testing techniques? The supporting reasons for white-box testing are given below:

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but

not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

5.2 Logic Coverage Criteria

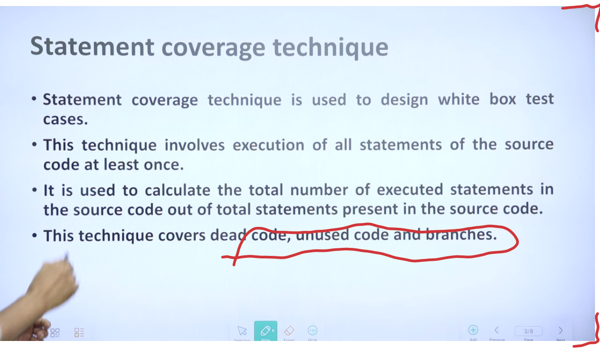
Definition

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. Discussed below are the basic forms of logic coverage.

Statement Coverage

The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Consider the following code segment shown in Fig. 5.1.



Statement coverage technique

- Statement coverage technique is used to design white box test cases.
- This technique involves execution of all statements of the source code at least once.
- It is used to calculate the total number of executed statements in the source code out of total statements present in the source code.
- This technique covers dead code, unused code and branches.

```
scanf (“%d”, &x);
scanf (“%d”, &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf (“x = ”, x);
 printf (“y = ”, y);
```

Figure 5.1 Sample code

If we want to cover every statement in the above code, then the following test cases must be designed:

Test case 1: $x = y = n$, where n is any number

Test case 2: $x = n$, $y = n'$, where n and n' are different numbers.

Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: $x > y$

Test case 4: $x < y$

These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, *statement coverage is a necessary but not a sufficient criteria for logic coverage.*

Decision or Branch Coverage

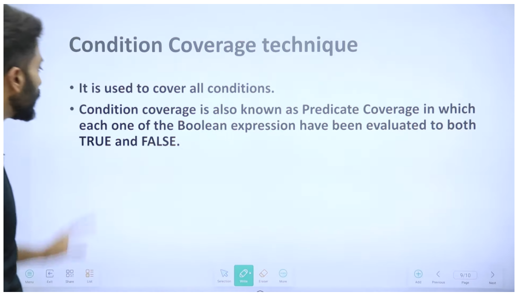
Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in Figure 5.1, *while* and *if* statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested. The test cases are designed as:

Test case 1: $x = y$

Test case 2: $x \,!= y$

Test case 3: $x < y$

Test case 4: $x > y$



Condition Coverage technique

- It is used to cover all conditions.
- Condition coverage is also known as Predicate Coverage in which each one of the Boolean expression have been evaluated to both TRUE and FALSE.

Condition Coverage

Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

while ((I ≤ 5) && (J < COUNT))

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

Test case 1: $I \le 5, J < \text{COUNT}$

Test case 2: $I < 5, J > \text{COUNT}$

Decision/condition Coverage

Condition coverage in a decision does not mean that the decision has been covered. If the decision *if (A && B)*

is being tested, the condition coverage would allow one to write two test cases:

Test case 1: A is True, B is False.

Test case 2: A is False, B is True.

But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once [2].

Multiple condition coverage In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there:

Test case 1: $A =$ True, $B =$ True

Test case 2: $A =$ True, $B =$ False

Test case 3: $A =$ False, $B =$ True

Test case 4: $A =$ False, $B =$ False

5.3 Basis Path Testing

Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing. Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors. But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths. Some criteria should be devised such that selected paths are executed for maximum coverage of logic. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test [9].
5. Choose enough paths in a program such that maximum logic coverage is achieved.

5.3.1 Control Flow Graph

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

- ***Node*** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- ***Edges or links*** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- ***Decision node*** A node with more than one arrow leaving it is called a decision node.
- ***Junction node*** A node with more than one arrow entering it is called a junction.
- ***Regions*** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

5.3.2 Flow Graph Notations for Different Programming Constructs

Since a flow graph is prepared on the basis of control structure of a program, some fundamental graphical notations are shown here (see Fig. 5.2) for basic programming constructs.

Using the above notations, a flow graph can be constructed. Sequential statements having no conditions or loops can be merged in a single node. That is why, the flow graph is also known as *decision-to-decision-graph* or *DD graph.*

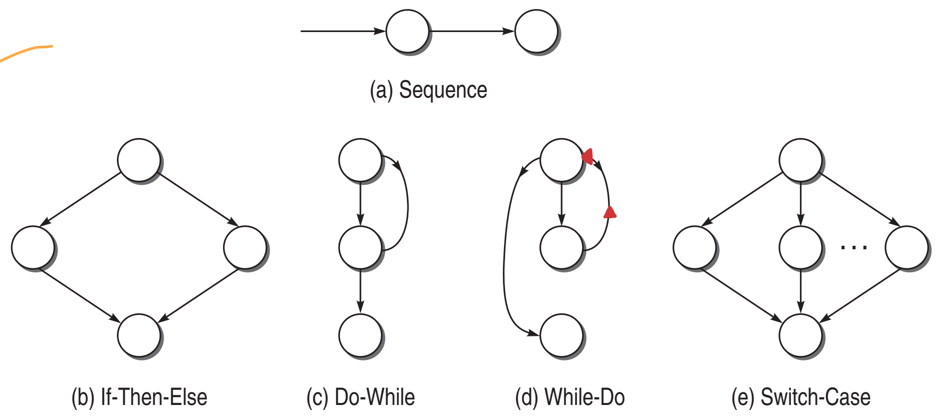


Figure 5.2

5.3.3 Path Testing Terminology

Path A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment A path segment is a succession of consecutive links that belongs to some path.

Length of a path The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined [9,28].

5.3.4 Cyclomatic Complexity

McCabe [24] has given a measure for the logical complexity of a program by considering its control flow graph. His idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, he considers only independent paths.

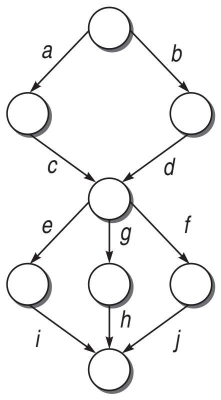


Figure 5.3 Sample graph

In the graph shown in Figure 5.3, there are six possible paths: *acei, acgh, acfh, bdei, bdgh, bdfj*.

In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4. In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

where n is the number of nodes and e is the number of arcs/edges.

However, it may be possible that the graph is not strongly connected. In that case, the above formula does not fit. Therefore, to make the graph strongly connected, we add an arc from the last node to the first node of the graph. In this way, the flow graph becomes a strongly connected graph. But by doing this, we increase the number of arcs by 1 and therefore, the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2$$

This is called the *cyclomatic number* of a program. We can calculate the cyclomatic number only by knowing the number of choice points (decision nodes) d in the program. It is given by

$$V(G) = d + 1$$

This is also known as *Miller's theorem.* We assume that a k-way decision point contributes for $k-1$ choice points.

The program may contain several procedures also. These procedures can be represented as separate flow graphs. These procedures can be called from any point but the connections for calling are not shown explicitly. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph. It is easy to demonstrate that, if p is the number of graphs and e and n are referred to as the whole graph, the cyclomatic number is given by

$$V(G) = e - n + 2p$$

And Miller's theorem becomes

$$V(G) = d + p$$

Formulae Based on Cyclomatic Complexity

Based on the cyclomatic complexity, the following formulae are being summarized.

Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G)$ = number of regions in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1$, where k is the number of arrows leaving the node.

Calculating the cyclomatic complexity number of the program having many connected components Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

$$V(G) = V(P) + V(X) + V(Y) + V(Z)$$

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$$V(G) = e - n + 2P$$

The complexity number derived collectively will be same as calculated above. Thus,

$$V(P \cup X \cup Y \cup Z) = V(P) + V(X) + V(Y) + V(Z)$$

Guidelines for Basis Path Testing

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example 5.1

Consider the following program segment:

Ex-1-6

```
main()
{
     int number, index;
1.   printf("Enter a number");
2.   scanf("%d, &number);
3.   index = 2;
4.   while(index <= number - 1)
5.   {
6.        if (number % index == 0)
7.        {
8.             printf("Not a prime number");
9.             break;
10.       }
11.       index++;
12.       }
13.     if(index == number)
14.         printf("Prime number");
15. } //end main
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the program, as shown in Fig. 5.4. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from the statement where a variable has been initialized.

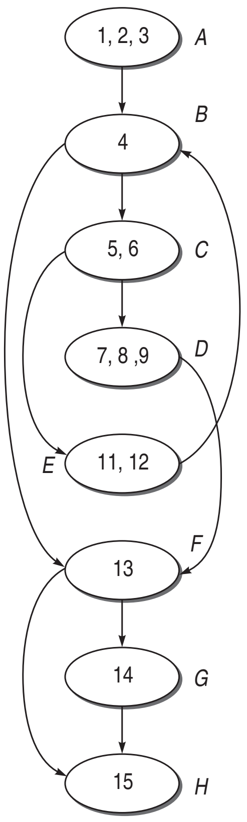


Figure 5.4 DD graph for Example 5.1

- Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
- Put the edges between the nodes according to their flow of execution.
- Put alphabetical numbering on each node like *A*, *B*, etc.

The DD graph of the program is shown in Figure 5.4.

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * p$

$= 10 - 8 + 2$

$= 4$

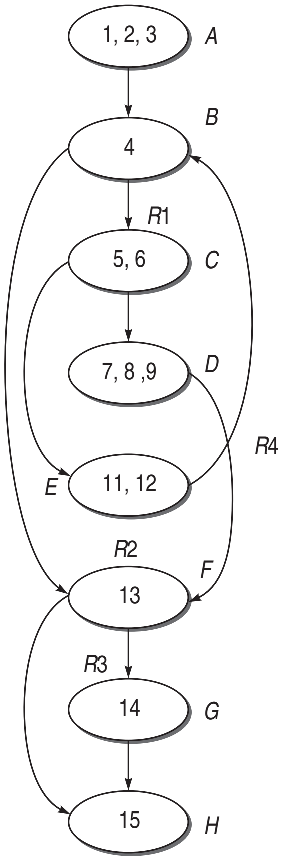


Figure 5.5 DD graph for Example 5.1 showing regions

(ii) $V(G)$ = Number of predicate nodes + 1

= 3 (Nodes *B*, *C*, and *F*) + 1

= 4

(iii) $V(G) = \text{Number of regions}$

$= 4(R1, R2, R3, R4)$

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H

(ii) A-B-F-G-H

(iii) A-B-C-E-B-F-G-H

(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Example 5.2

Consider the following program that reads in a string and then checks the type of each character.

```
     main()
     {
         char string [80];
         int index;
1.       printf(“Enter the string for checking its characters”);
2.       scanf(“%s”, string);
3.       for(index = 0; string[index] != ‘\0’; ++index)   {
4.           if((string[index] >= ‘0’ && (string[index] <=‘9’
5.                        printf(“%c is a digit”, string[index]);
6.           else if ((string[index] >= ‘A’ && string[index] <‘Z’))  ||
                      ((string[index] >= ‘a’ && (string[index] <‘z’)))
7.                    printf(“%c is an alphabet”, string[index]);
8.           else
9.                    printf(“%c is a special character”, string[index]);
10.          }
11.  }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.6.

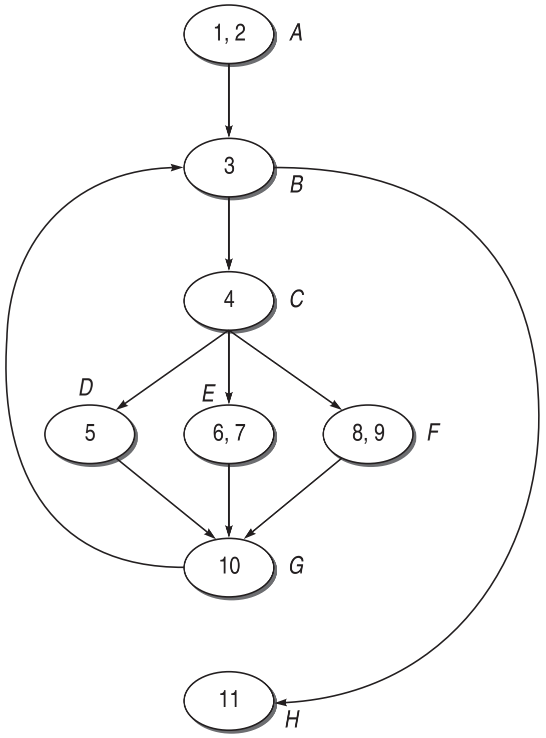


Figure 5.6 DD graph for Example 5.2

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * P$
$= 10 - 8 + 2$
$= 4$

(ii) $V(G)$ = Number of predicate nodes + 1
= 3 (Nodes *B*, *C*) + 1
= 4

Node *C* is a multiple IF-THEN-ELSE, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes

= Number of links out of main node − 1

= 3 – 1 = 2 (For node *C*)

(iii) $V(G)$ = Number of regions

= 4

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) *A-B-H*

(ii) *A-B-C-D-G-B-H*

(iii) *A-B-C-E-G-B-H*

(iv) *A-B-C-F-G-B-H*

(d) Test case design from the list of independent paths

Test Case ID	Input Line	Expected Output	Independent paths covered by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H

Example 5.3

Consider the following program:

```
      main()
      {
          char chr;
1.        printf (“Enter the special character\n”);
2.        scanf (%c”, &chr);
3.        if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
             (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
             (chr != 56) && (chr != 57)
4.        {
5.            switch(chr)
6.          {
7.          Case '*': printf("It is a special character");
8.          break;
9.          Case '#': printf("It is a special character");
10.         break;
11.         Case '@': printf("It is a special character");
12.         break;
13.         Case '!':  printf("It is a special character");
14.         break;
15.         Case '%': printf("It is a special character");
16.         break;
17.         default : printf("You have not entered a special character");
18.         break;
19.         }// end of switch
20.     } // end of If
21.     else
22.         printf("You have not entered a character");
23. } // end of main()
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

Solution

(a) **DD graph**

The DD graph of the program is shown in Fig. 5.7.

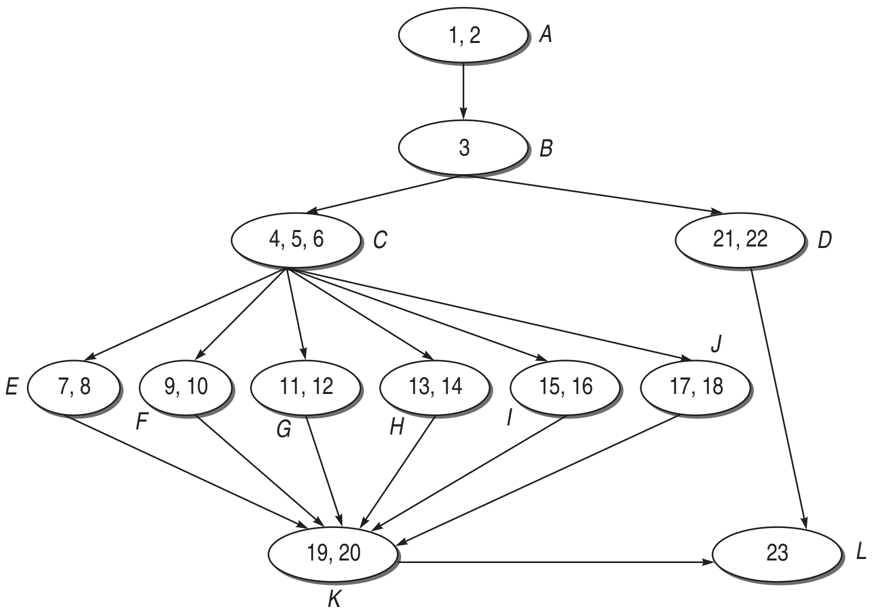


Figure 5.7 DD graph for Example 5.3

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2p$
$= 17 - 12 + 2$
$= 7$

(ii) $V(G) =$ Number of predicate nodes + 1
$= 2$ (Nodes B, C) + 1
$= 7$

Node C is a switch-case, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes

= Number of links out of main node −1
$= 6 - 1 = 5$ (For node C)

(iii) $V(G) =$ Number of regions = 7

(c) Independent paths

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

1. *A-B-D-L*
2. *A-B-C-E-K-L*
3. *A-B-C-F-K-L*
4. *A-B-C-G-K-L*
5. *A-B-C-H-K-L*
6. *A-B-C-I-K-L*
7. *A-B-C-J-K-L*

(d) Test Case Design from the list of Independent Paths

Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

Example 5.4

Consider a program to arrange numbers in ascending order from a given list of *N* numbers.

```
         main()
         {
         int num,small;
1.       int i,j,sizelist,list[10],pos,temp;
         clrscr();
         printf("\nEnter the size of list :\n ");
         scanf("%d",&sizelist);
2.       for(i=0;i<sizelist;i++)
         {
3.       printf("\nEnter the number");
         scanf ("%d",&list[i]);
         }
4.       for(i=0;i<sizelist;i++)
         {
5.       small=list[i];
         pos=i;
6.       for(j=i+1;j<sizelist;j++)
         {
7.       if(small>list[j])
         {
         small=list[j];
8.       pos=j;
         }
9.       }
         temp=list[i];
10.      list[i]=list[pos];
         list[pos]=temp;
11.      }
12.      printf("\nList of the numbers in ascending order : ");
13.      for(i=0;i<sizelist;i++)
14.      printf("\n%d",list[i]);
         getch();
15.      }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.8.

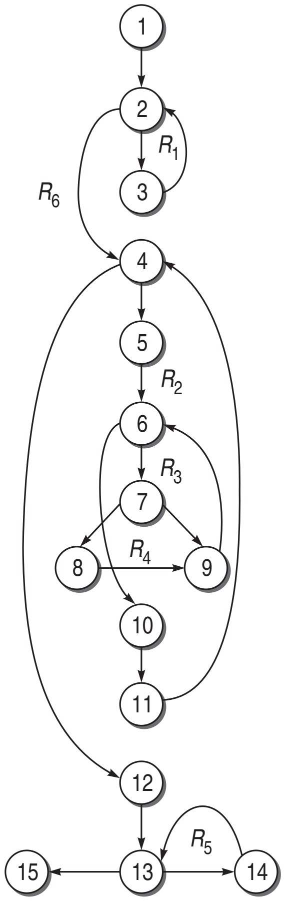


Figure 5.8 DD graph for Example 5.4

(b) Cyclomatic complexity

1. $V(G) = e - n + 2p$

$= 19 - 15 + 2$

$= 6$

2. $V(G) = \text{Number of predicate nodes} + 1$
$= 5 + 1$
$= 6$
3. $V(G) = \text{Number of regions}$
$= 6$

(c) Independent paths

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. 1-2-3-2-4-5-6-7-8-9-6-10-11-4-12-13-14-13-15
2. 1-2-3-2-4-5-6-7-9-6-10-11-4-12-13-14-13-15
3. 1-2-3-2-4-5-6-10-11-4-12-13-14-13-15
4. 1-2-3-2-4-12-13-14-13-15 (path not feasible)
5. 1-2-4-12-13-15
6. 1-2-3-2-4-12-13-15 (path not feasible)

(d) Test case design from the list of independent paths

Test Case ID	Input	Expected Output	Independent path covered by Test Case

1	Sizelist = 5
	List[] = {17,6,7,9,1}
		1,6,7,9,17	1

2	Sizelist = 5
	List[] = {1,3,9,10,18}
		1,3,9,10,18	2

3	Sizelist = 1
	List[] = {1}
		1	3

4	Sizelist = 0	blank	blank

Example 5.5

Consider the program for calculating the factorial of a number. It consists of main() program and the module fact(). Calculate the individual cyclomatic complexity number for main() and fact() and then, the cyclomatic complexity for the whole program.

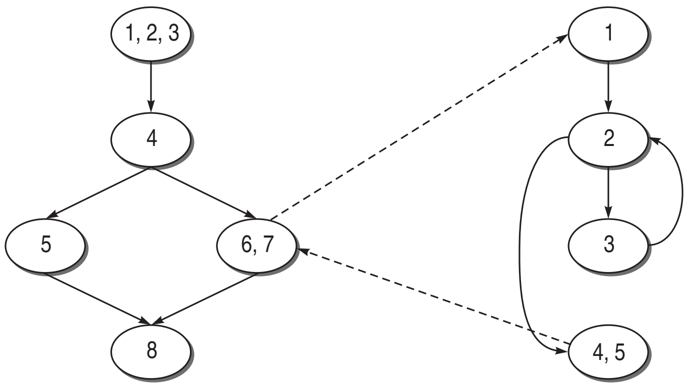
```
      main()
      {
          int number;
          int fact();
1.        clrscr();
2.        printf("Enter the number whose factorial is to be found out");
3.        scanf("%d", &number);
4.        if(number <0)
5.           printf("Facorial cannot be defined for this number);
6.        else
7.           printf("Factorial is %d", fact(number));
8.    }

      int fact(int number)
      {
          int index;
1.        int product =1;
2.        for(index=1; index<=number; index++)
3.           product = product * index;
4.        return(product);
5.    }
```

Solution

DD graph

The DD graph of the program is shown in Fig. 5.9



(a) Flow graph for main () (b) Flow graph for fact ()

Figure 5.9

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p$

$= 5 - 5 + 2$

$= 2$

(b) $V(M) = d + p$

$= 1 + 1$

$= 2$

(c) $V(M) =$ Number of regions $= 2$

Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

(b) $V(R) =$ Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R) =$ Number of regions $= 2$

Cyclomatic complexity of the whole graph considering the full program

(a) $V(G) = e - n + 2p$
$= 9 - 9 + 2 \times 2$
$= 4$
$= V(M) + V(R)$

(b) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(c) $V(G) =$ Number of regions
$= 4$
$= V(M) + V(R)$

5.3.5 Applications of Path Testing

Path testing has been found better suitable as compared to other testing methods. Some of its applications are discussed below.

Thorough testing / More coverage Path testing provides us the best code coverage, leading to a thorough testing. Path coverage is considered better as compared to statement or branch coverage methods because the basis path set provides us the number of test cases to be covered which ascertains the number of test cases that must be executed for full coverage. Generally, branch coverage or other criteria gives us less number of test cases as compared to path testing. Cyclomatic complexity along with basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

Unit testing Path testing is mainly used for structural testing of a module. In unit testing, there are chances of errors due to interaction of decision outcomes or control flow problems which are hidden with branch testing. Since each decision outcome is tested independently, path testing uncovers these errors in module testing and prepares them for integration.

Integration testing Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules. Path testing analyses all the paths on the interface and explores all the errors.

Maintenance testing Path testing is also necessary with the modified version of the software. If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing. In any case, path testing is still able to detect any security threats on the interface with the called modules.

Testing effort is proportional to complexity of the software Cyclomatic complexity number in basis path testing provides the number of tests to be executed on the software based on the complexity of the software. It means the number of tests derived in this way is directly proportional to the complexity of the software. Thus, path testing takes care of the complexity of the software and then derives the number of tests to be carried out.

Basis path testing effort is concentrated on error-prone software Since basis path testing provides us the number of tests to be executed as a measure of software cyclomatic complexity, the cyclomatic number signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

5.4 GRAPH MATRICES

Flow graph is an effective aid in path testing as seen in the previous section. However, path tracing with the use of flow graphs may be a cumbersome and time-consuming activity. Moreover, as the size of graph increases, manual path tracing becomes difficult and leads to errors. A link can be missed or covered twice. So the idea is to develop a software tool which will help in basis path testing.

Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing. The reason being the properties of graph matrices are fundamental to test tool building. Moreover, testing theory can be explained on the basis of graphs. Graph theorems can be proved easily with the help of graph matrices. So graph matrices are very useful for understanding the testing theory.

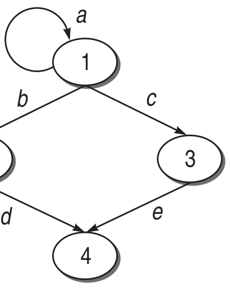
5.4.1 Graph Matrix

A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph. Each row and column identifies a particular node and matrix entries represent a connection between the nodes.

The following points describe a graph matrix:

- Each cell in the matrix can be a direct connection or link between one node to another node.
- If there is a connection from node 'a' to node 'b', then it does not mean that there is connection from node 'b' to node 'a'.
- Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

Example 5.6



Consider the above graph and represent it in the form of a graph matrix.

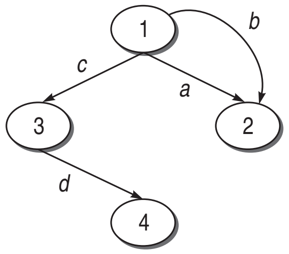
Solution

The graph matrix is shown below.

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

Example 5.7

Consider the graph and represent it in the form of a graph matrix.



Solution

The graph matrix is shown below.

	1	2	3	4
1		a+b	c	
2				
3				d
4				

5.4.2 Connection Matrix

Till now, we have learnt how to represent a flow graph into a matrix representation. But this matrix is just a tabular representation and does not provide any useful information. If we add link weights to each cell entry, then graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned a link weight which becomes the entry in the cell of matrix. The link weight provides information about control flow.

In the simplest form, when the connection exists, then the link weight is 1, otherwise 0 (But 0 is not entered in the cell entry of matrix to reduce the complexity). A matrix defined with link weights is called a connection matrix. The connection matrix for Example 5.6 is shown below.

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

The connection matrix for Example 5.7 is shown below.

	1	2	3	4
1		1	1	
2				
3				1
4				

5.4.3 Use of Connection Matrix in Finding Cyclomatic Complexity Number

Connection matrix is used to see the control flow of the program. Further, it is used to find the cyclomatic complexity number of the flow graph. Given below is the procedure to find the cyclomatic number from the connection matrix:

Step 1: For each row, count the number of 1s and write it in front of that row.

Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

The cyclomatic number calculated from the connection matrix of Example 5.6 is shown below.

	1	2	3	4	
1	1	1	1		**3 – 1 = 2**
2				1	**1 – 1 = 0**
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 2+1 = 3				

The cyclomatic number calculated from the connection matrix of Example 5.7 is shown below.

	1	2	3	4	
1		1	1		**2 – 1 = 1**
2					
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 1+1 = 2				

5.4.4 Use of Graph Matrix for Finding Set of all Paths

Another purpose of developing graph matrices is to produce a set of all paths between all nodes. It may be of interest in path tracing to find *k*-link paths from one node. For example, how many 2-link paths are there from one node to another node? This process is done for every node resulting in the set of all paths. This set can be obtained with the help of matrix operations. The main objective is to use matrix operations to obtain the set of all paths between all nodes. The set of all paths between all nodes is easily expressed in terms of matrix operations.

The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive (mostly, relations used in testing are transitive). For example, the square of matrix represents path segments that are 2-links long. Similarly, the cube power of matrix represents path segments that are 3-links long.

Generalizing, we can say that mth power of the matrix represents path segments that are m-links long.

Example 5.8

Consider the graph matrix in Example 5.6 and find 2-link paths for each node.

Solution

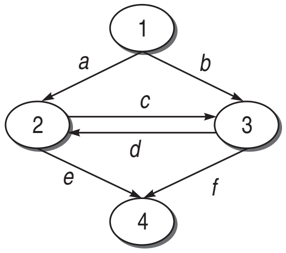
For finding 2-link paths, we should square the matrix. Squaring the matrix yields a new matrix having 2-link paths.

$$\begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a^2 & ab & ac & bd + ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The resulting matrix shows all the 2-link paths from one node to another. For example, from node 1 to node 2, there is one 2-link, i.e., *ab*.

Example 5.9

Consider the following graph. Derive its graph matrix and find 2-link and 3-link set of paths.



Solution

The graph matrix of the graph is shown below.

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

First we find 2-link set of paths by squaring this matrix as shown below:

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Next, we find 3-link set of paths by taking the cube of matrix as shown below:

$$\begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & acd & bdc & bde+acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

It can be generalized that for n number of nodes, we can get the set of all paths of $(n-1)$ links length with the use of matrix operations. These operations can be programmed and can be utilized as a software testing tool.

5.5 Loop Testing

Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint. If loops are not tested properly, bugs can go undetected. This is the reason that loops are covered in this section exclusively. Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module. Sufficient test cases should be designed to test every loop thoroughly.

There are four different kinds of loops. How each kind of loop is tested, is discussed below.

Simple loops Simple loops mean, we have a single loop in the flow, as shown in Fig. 5.9.

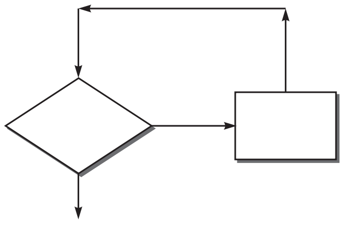


Figure 5.9 (a)

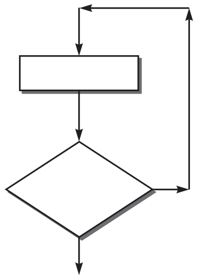


Figure 5.9 (b)

The following test cases should be considered for simple loops while testing them [9]:

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min–1, max–1, max, and max+1 number of iterations through the loop.

Nested loops When two or more loops are embedded, it is called a nested loop, as shown in Fig. 5.10. If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically. Thus, the strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered [9].

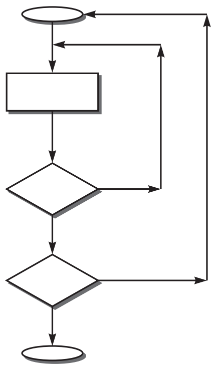


Figure 5.10 Nested loops

Concatenated loops The loops in a program may be concatenated (Fig. 5.11). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit. If the two loops are not on the same path, then they are not concatenated. The two loops on the same path may or may not be independent. If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

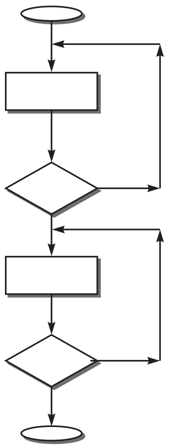


Figure 5.11 Concatenated loops

Unstructured loops This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

5.6 Data Flow Testing

In path coverage, the stress was to cover a path using statement or branch coverage. However, data and data integrity is as important as code and code integrity of a module. We have checked every possibility of the control flow of a module. But what about the data flow in the module? Has every data object been initialized prior to use? Have all defined data objects been used for something? These questions can be answered if we consider data objects in the control flow of a module.

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers. For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate. For example,

```
int a;
if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used. For example, if an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen. The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced. Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere. Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

5.6.1 State of a Data Object

A data object can be in the following states:

- ***Defined (d)*** A data object is called *defined* when it is initialized, i.e. when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on [9].

- ***Killed/Undefined/Released (k)*** When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.
- ***Usage (u)*** When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either *computational use* (c-use) or *predicate use* (p-use).

5.6.2 Data-Flow Anomalies

Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code. An anomaly is denoted by a two-character sequence of actions. For example, 'dk' means a variable is defined and killed without any use, which is a potential bug. There are nine possible two-character combinations out of which only four are data anomalies, as shown in Table 5.1.

Table 5.1 Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur. In addition to the above two-character data anomalies, there may be single-character data anomalies also. To represent these types of anomalies, we take the following conventions:

$\sim x$: indicates all prior actions are not of interest to x.

$x\sim$: indicates all post actions are not of interest to x.

All single-character data anomalies are listed in Table 5.2.

Table 5.2 Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

5.6.3 Terminology Used in Data Flow Testing

In this section, some terminology [9,20], which will help in understanding all the concepts related to data-flow testing, is being discussed. Suppose *P* is a program that has a graph *G*(*P*) and a set of variables *V*. The graph has a single entry and exit node.

Definition node Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Usage node It means the variable has been used in some statement of the program. Node *n* that belongs to *G*(*P*) is a usage node of variable *v*, if the value of variable *v* is used at the statement corresponding to node *n*. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.
A usage node can be of the following two types:

(i) Predicate Usage Node: If usage node *n* is a predicate node, then *n* is a predicate usage node.

(ii) Computation Usage Node: If usage node *n* corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Loop-free path segment It is a path segment for which every node is visited once at most.

Simple path segment It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.

Definition-use path (du-path) A du-path with respect to a variable *v* is a path between the definition node and the usage node of that variable. Usage node can either be a *p*-usage or a *c*-usage node.

Definition-clear path(dc-path) A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.

The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons. Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths. The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in code to trace the bug. So the du-paths which are not dc-paths need more attention.

5.6.4 Static Data Flow Testing

With static analysis, the source code is analysed without executing it. Let us consider an example of an application given below.

Example 5.10

Consider the program given below for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary. Calculate his gross salary.

```
main()
{
    float bs, gs, da, hra = 0;
    printf("Enter basic salary");
    scanf("%f", &bs);
    if(bs < 1500)
    {
        hra = bs * 10/100;
        da = bs * 90/100;
    }
    else
    {
        hra = 500;
        da = bs * 98/100;
    }
    gs = bs + hra + da;
    printf("Gross Salary = Rs. %f", gs);
}
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Solution

For variable 'bs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'gs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

From the above static analysis, it was observed that static data flow testing for the variable 'hra' discovered one bug of double definition in line number 1.

Static Analysis is not Enough

It is not always possible to determine the state of a data variable by just static analysis of the code. For example, if the data variable in an array is used as an index for a collection of data elements, we cannot determine its state by static analysis. Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence, not completely anomalous. Thus, all anomalies using static analysis cannot be determined and this problem is provably unsolvable.

5.6.5 Dynamic Data Flow Testing

Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code. The test cases are designed in such a way that every definition of data variable to each of its use is traced and every use is traced to each of its definition. Various strategies are employed for the creation of test cases. All these strategies are defined below.

All-du Paths (ADUP) It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy, since it is a superset of all other data flow testing strategies. Moreover, this strategy requires the maximum number of paths for testing.

All-uses (AU) This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

All-p-uses/Some-c-uses (APU + C) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.

All-c-uses/Some-p-uses (ACU + P) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every computational use. If there are definitions of the variable with no c-use following it, then add predicate use (c-use) test cases as required to cover every definition.

All-Predicate-Uses (APU) It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-Computational-Uses (ACU) It is derived from the strategy ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-Definition (AD) It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example 5.11

Consider the program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
     main()
     {
     int work;
0.   double payment =0;
1.   scanf("%d", work);
2.   if (work > 0) {
3.       payment = 40;
4.   if (work > 20)
5.   {
6.       if(work <= 30)
7.           payment = payment + (work - 25) * 0.5;
8.       else
9.       {
10.          payment = payment + 50 + (work -30) * 0.1;
11.              if (payment >= 3000)
12.                  payment = payment * 0.9;
13.      }
14.  }
15.  }
16.  printf("Final payment", payment);
```

Solution

Figure 5.12 shows the control flow graph for the given program.

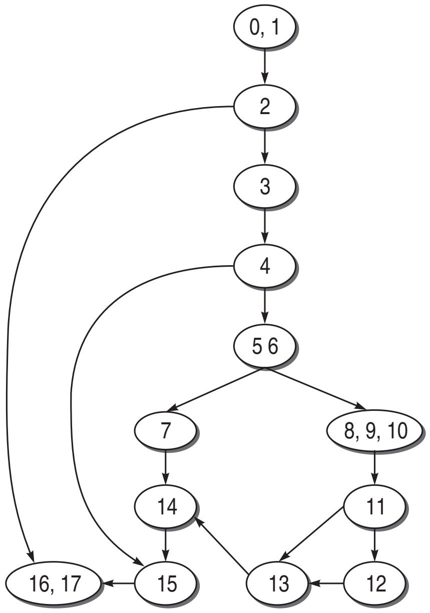


Figure 5.12 DD graph for Example 5.11

Figure 5.13 shows the data flow graph for the variable 'payment'.

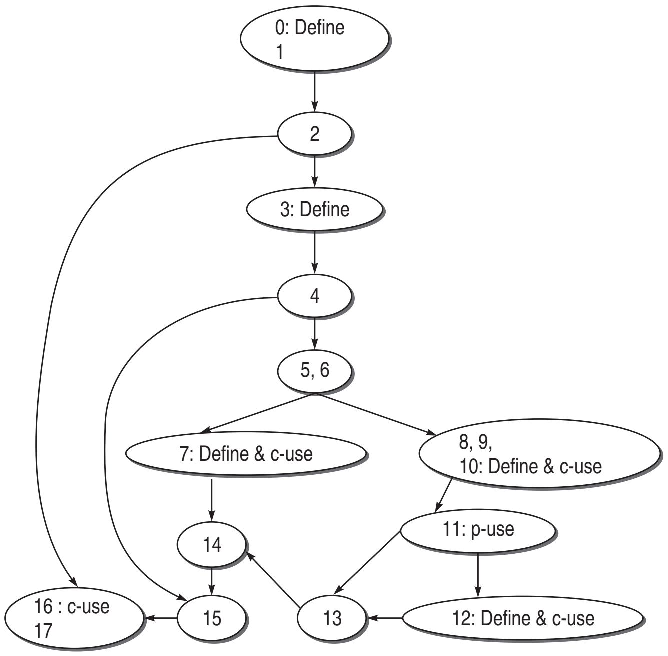


Figure 5.13 Data flow graph for 'payment'

Figure 5.14 shows the data flow graph for the variable 'work'.

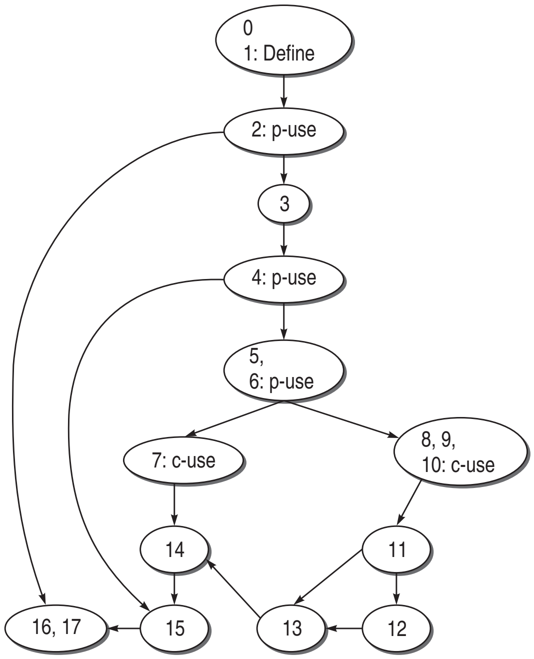


Figure 5.14 Data flow graph for variable 'work'

Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

Data flow testing paths for each variable are shown in Table 5.3.

Table 5.3 Data flow testing paths

Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses (APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6

All c-uses (ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11 10-11-12 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16 0-1-2-3-4-5-6-8-9-10-11	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10 1-2-3-4-5-6
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11 0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All Definitions (AD)	0-1-2-16 3-4-5-6-7 7-14-15-16 10-11 12-13-14-15-16	1-2

5.6.6 Ordering of Data Flow Testing Strategies

While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies. Figure 5.15 depicts the relative strength of the data flow strategies. In this figure, the relative strength of testing strategies reduces along the direction of the arrow. It means that all-du-paths (ADPU) is the strongest criterion for selecting the test cases.

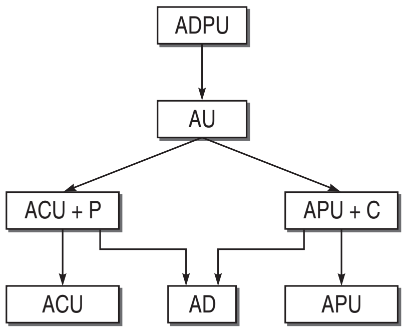


Figure 5.15 Data-flow testing strategies

5.7 Mutation Testing

Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good, otherwise we must focus on the quality of test data. Therefore, mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case. Thus, the main objective is to select efficient test data which have error-detection power. The criterion for this test data is to differentiate the initial program from the mutant. This distinguish-ability between the initial program and its mutant will be based on test results.

5.7.1 Primary Mutants

Let us take an example of a C program to understand primary mutants.

```
...
if (a > b)
    x = x + y;
else
    x = y;
printf("%d", x);
...
```

We can consider the following mutants for the above example:

M1: `x = x - y;`
M2: `x = x / y;`
M3: `x =  x + 1;`
M4: `printf("%d", y);`

When the mutants are single modifications of the initial program using some operators as shown above, they are called *primary mutants.* Mutation operators are dependent on programming languages. Note that each of the mutated statements represents a separate program. The results of the initial program and its mutants are shown below.

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

5.7.2 Secondary Mutants

Let us take another example program as shown below:

```
if (a < b)
    c = a;
```

Now, mutants for this code may be as follows:

```
M1 : if (a <= b-1)
                c = a;
M2: if (a+1 <= b)
                c = a;
M3: if (a == b)
                c = a+1;
```

This class of mutants is called *secondary mutants* when multiple levels of mutation are applied on the initial program. In this case, it is very difficult to identify the initial program from its mutants.

Example 5.12

Consider the program *P* shown below.

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] > a[r]
        r = i;
}
```

The mutants for *P* are:
M1:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[i] > a[r])
        r = i;
}
```

M2:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (i > a[r])
        r = i;
}
```

M3:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] >= a[r])
        r = i;
}
```

M4:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[r] > a[r])
        r = i;
}
```

Let us consider the following test data selection:

	a[1]	a[2]	a[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

We apply these test data to mutants, M1, M2, M3, and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 and M4
TD3	1	1	1	1	1	none

We need to look at the efficiency of the proposed test data. It can be seen that M1 and M3 are not killed by the test data selection. Therefore, the test data is incomplete. Therefore, we need to add a new test data, TD4 = {2,2,1}, then this test data kills M3.

5.7.3 Mutation Testing Process

The mutation testing process is discussed below:

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories.
    - One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.
    - Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called *mutation-adequate*.

SUMMARY

White-box testing is a dynamic testing category under which the software is tested using the structure or logic of the program under test. This technique is closer to a developer who needs to test his module while developing it. The developer cannot design and code the module in one go and therefore he is supposed to verify his logic using various test case design methods.

White-box testing is necessary, as it is one of the steps to verify the correctness of the program and consequently in enhancing the quality of the program. All the methods of white-box testing have been discussed in this chapter, taking sufficient number of examples such that a new method can be learnt easily and the reader is ready to design the test cases using a particular method.

Let us review the important concepts described in this chapter:

- White-box testing is the technique wherein it is required to understand the structure and logic of the program to test the software.
- White-box testing is largely done by the developer of the software.
- White-box testing covers the logic of the program. The forms of logic coverage are: statement coverage, decision/branch coverage, condition coverage, decision-condition coverage, and multiple condition.
- Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.
- The basis set is derived by calculating the cyclomatic complexity number. Having the knowledge of this number, independent paths are derived from the flow graph of the program. Corresponding to independent paths, test cases are prepared which form the basis set.
- An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.
- Cyclomatic complexity is the number which tells us about the complexity of the design of a module. It should be less than 10, otherwise the module should be redesigned.
- Cyclomatic complexity of a module can be calculated by the following methods:
    - $V(G) = e - n + 2P$

      where e is number of edges, n is the number of nodes in the graph, and P is the number of components in the whole graph
    - $V(G) = d + P$

      where d is the number of decision nodes in the graph.
    - $V(G)$ = number of regions in the graph
    - Using graph matrices
- Graph matrix, a data structure, can assist in developing a tool for automation of path tracing.

- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph.
- A matrix defined with link weights is called a connection matrix.
- The connection matrix is used to find the cyclomatic complexity number of the flow graph.
- Graph matrices are also used to produce a set of all paths between all nodes. An *m*th power of the matrix represents all path segments *m* links long.
- Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.
- Data flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- Definition node in a data flow graph is the node wherein a variable is assigned a value for the very first time in the program.
- Usage node in a data flow graph is the node wherein the variable has already been used in some statement of the program.
- A du-path with respect to a variable *v* is a path between the definition node and the usage node of that variable.
- A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.
- du-paths which are not definition-clear paths are important from testing viewpoint, as these are potential problematic spots for testing persons. du-paths which are definition-clear are easy to test in comparison to those which are not dc-paths.
- The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in the code to trace the bug. So the du-paths which are not definition-clear need more attention of the tester.
- During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains a fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called mutants of the original program and a mutant is said to be killed when a test case causes it to fail.
- When the mutants are single modifications of the initial program using some operators, they are called primary mutants.
- When multiple levels of mutation are applied on the initial program, they are called secondary mutants.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. White-box testing is _______ to black-box testing.
   (a) mutually exclusive
   (b) complementary
   (c) not related

2. The effectiveness of path testing rapidly _______ as the size of software under test.
   (a) decreases
   (b) increases
   (c) does not change
   (d) none of the above
3. A node with more than one arrow leaving it is called a _______
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
4. A node with more than one arrow entering it is called a _______
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
5. Areas bounded by edges and nodes are called _______
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
6. The length of a path is measured by the number of _______
   (a) instructions
   (b) junction nodes
   (c) decision nodes
   (d) links
7. An independent path is any path through the graph that introduces at least _______ new set of processing statements or new conditions.
   (a) 4
   (b) 3
   (c) 1
   (d) 2
8. The number of independent paths is given by _______.
   (a) $V(G) = e - n + 1$
   (b) $V(G) = 2e - n + 1$
   (c) $V(G) = e - n + 2$
   (d) none of the above
9. According to Mill's Theorem, _______
   (a) $V(G) = d + 2P$

(b) $V(G) = d + P$
(c) $V(G) = 2d + P$
(d) None of the above

10. In data flow anomalies, dd is a _______
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
11. In data flow anomalies, du is a _______
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
12. In data flow anomalies, ku is a _______
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
13. In single-character data anomalies, ~d is _______
    (a) potential bug
    (b) normal situation
    (c) none of the above
15. In single-character data anomalies, ~k is _______
    (a) potential bug
    (b) normal situation
    (c) none of the above
16. _______ is the strongest criterion for selecting test cases.
    (a) AD
    (b) APU
    (c) AU
    (d) ADPU

Review Questions

1. What is the need of white-box testing?
2. What are the different criteria for logic coverage?
3. What is basis path testing?

4. Distinguish between decision node and junction node?
5. What is an independent path?
6. What is the significance of cyclomatic complexity?
7. How do you calculate the number of decision nodes for switch-case?
8. How do you calculate the cyclomatic complexity number of the program having many connected components?
9. Consider the program.

```
#include <stdio.h>
main()
{
        int a,b, c,d;
        clrscr();
        printf("enter the two variables a,b");
        scanf("%d %d",&a,&b);
        printf("enter the option 1:Addition,
               2:subtraction,3:multiplication,4:division");
        scanf("%d",&c);
        switch(c)
        {
                case 1:d = a+b;
                printf("Addition of two no.=%d", d);
                break;
                case 2:d = a-b;
                printf("Subtraction of two no.=%d", d);
                break;
                case 3:d = a*b;
                printf("Multiplication of two no.=%d", d);
                break;
                case 4:d = a/b;
                printf("division of two no.=%d",d);
                break;
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

10. Consider the program to find the greatest number:

```
#include <stdio.h>
main()
{
        float x,y,z;
        clrscr();
        printf("enter the three variables x,y,z");
        scanf("%f %f %f",&x,&y,&z);
        if(x > y)
        {
            if(x > z)
                printf("x is greatest");
            else
                printf("z is greatest");
        }
        else
        {
            if(y > z)
                printf("y is greatest");
            else
                printf("z is greatest");
        }
    getch();
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

11. Consider the following program which multiplies two matrices:

```
#include <stdio.h>
main()
{
int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE], i, j, k, rowl,
colml, row2, colm2;

printf("Enter the order of first matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row1, colm1);
printf("Enter the order of second matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row2, colm2);
if(colm1==row2)
```

```
        {
        printf("Enter first matrix");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colml; j++)
                scanf("%d", &a[i][j]);
                }
        printf("Enter second matrix");
        for(i=0; i<row2; i++)
                {
                for(j=0; j<colm2; j++)
                scanf("%d", &b[i][j]);
                }
        printf("Multiplication of two matrices is");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colm1; j++)
                        {
                        c[i] [j] = 0;
                        for(k=0; k<row2; k++)
                        c[i][j]+ = a[i][k] + b[k][j];
                        printf("%6d", c[i][j]);
                        }
                }
        }
        else
        {
                printf("Matrix multiplication is not possible");
        }
}
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all four methods.

(c) List all independent paths.

(d) Design all test cases from the independent paths.

(e) Derive all du-paths and dc-paths using data flow testing.

12. Consider the following program for finding the prime numbers, their sum, and count:

```
main()
{
        int num, flag, sum, count;
        int CheckPrime(int n);
        sum = count = 0;
        printf("Prime number between 1 and 100 are");
        for(num=1; num<=50; num++)
        {
                flag = CheckPrime(num);
                if(flag)
                {
                        printf("%d", num);
                        sum+ = num;
                        count++;
                }
        }
        printf("Sum of primes %d", count);
}
int CheckPrime(int n)
{
        int srt, d;
        srt = sqrt(n);
        d = 2;
        while(d <= srt)
        {
                If(n%d == 0)
                break;
                d++;
        }
        if(d > srt)
                return(1);
        else
                return(0);
}
```

(a) Draw the DD graph for the program.
(b) This program consists of main() and one module. Calculate the individual cyclomatic complexity number for both and collectively for the whole program. Show that individual cyclomatic complexity of main() and CheckPrime() and cyclomatic complexity of whole program is equal.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

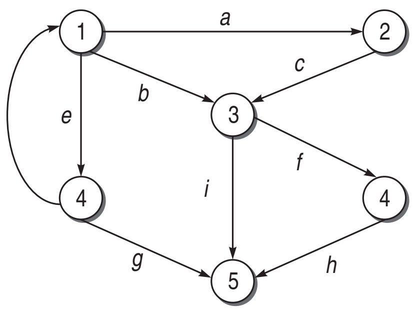
13. Consider the following program for calculating the grade of a student:

```
main()
{
     char grade;
     int s1, s2, s3, s4, total;
     float average;
     printf("Enter the marks of 4 subjects");
     scanf("%d  %d  %d  %d", &s1,&s2,&s3,&s4);
```

```
        if(s1 < 40 || s2 < 40 || s3 < 40 || s4 < 40)
              printf("Student has failed");
        else
        {
              total = s1 + s2 + s3 + s4;
              average = total/4.0;
              if(average > 80.0)
                    grade = 'A';
              else if(average >= 70.0)
                    grade = 'B';
              else if(average >= 60.0)
                    grade = 'C';
              else if(average >= 50.0)
                    grade = 'D';
              else if(average >= 45.0)
                    grade = 'E';
              else
                    grade = 'F';
              switch(grade)
              {
                    case 'A':  printf("Student has got A grade");
                    case 'B':  printf("Student has got B grade");
                    case 'C':  printf("Student has got C grade");
                    case 'D':  printf("Student has got D grade");
                    case 'E':  printf("Student has got E grade");
                    case 'F':  printf("Student has got F grade");
              }
        }
  }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.

14. Consider the following graph:



(a) Represent this graph in the form of a graph matrix.
(b) Represent this graph in the form of a connection matrix.
(c) Find 2-link paths and 3-link paths for each node.

15. ‘Nested loops are problematic areas for testers.’ Comment on this.
16. Give examples of each type of data anomaly.
17. Consider Question 9 and perform static data flow analysis by finding out the define-use-kill patterns for all the variables in the source code.
18. Consider Question 9 and draw data flow graph for each variable used in the program and derive data flow testing paths with all the strategies.
19. What is the difference between primary and secondary mutants?
20. Consider Example 5.10. Find out the possible mutants and check how many of them are killed by a set of test data. Add new test data if required.

Chapter

6

Static Testing

OBJECTIVES

After reading this chapter, you should be able to understand:

- Static testing is complementary to dynamic testing
- Static testing also improves the software quality
- Some bugs can be detected only through static testing
- Three types of static testing: Inspection, Walkthroughs, and Reviews
- Inspections are the most widely used technique for static testing which is a formal process to detect the bugs early
- Benefits and effectiveness of inspection process
- Variants of inspection process
- Walkthrough is a less formal and less rigorous method as compared to inspection process
- Review is a higher level technique as compared to inspection or walkthrough, as it also includes management representatives

We have discussed dynamic testing techniques that execute the software being built with a number of test cases. However, it suffers from the following drawbacks:

- Dynamic testing uncovers the bugs at a later stage of SDLC and hence is costly to debug.
- Dynamic testing is expensive and time-consuming, as it needs to create, run, validate, and maintain test cases.
- The efficiency of code coverage decreases with the increase in size of the system.
- Dynamic testing provides information about bugs. However, debugging is not always easy. It is difficult and time-consuming to trace a failure from a test case back to its root cause.
- Dynamic testing cannot detect all the potential bugs.

While dynamic testing is an important aspect of any quality assurance program, it is not a universal remedy. Thus, it alone cannot guarantee defect-free product, nor can it ensure a sufficiently high level of software quality.

In response to the above discussion, static testing is a complimentary technique to dynamic testing technique to acquire higher quality software. Static testing techniques do not execute the software and they do not require the bulk of test cases. This type of testing is also known as non-computer based testing or human testing. All the bugs cannot be caught alone by the dynamic testing technique; static testing reveals the errors which are not shown by dynamic testing. Static testing can be applied for most of the verification activities discussed earlier. Since verification activities are required at every stage of SDLC till coding, static testing also can be applied at all these phases.

Static testing techniques do not demonstrate that the software is operational or that one function of software is working; rather they check the software product at each SDLC stage for conformance with the required specifications or standards. Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.

Static testing has proved to be a cost-effective technique of error detection. An empirical comparison between static and dynamic testing [26, 27], proves the effectiveness of static testing. Further, Fagan [28] reported that more than 60% of the errors in a program can be detected using static testing. Another advantage of static testing is that it provides the exact location of a bug, whereas dynamic testing provides no indication of the exact source code location of the bug. In other words, we can say that static testing finds the in-process errors before they become bugs.

Static testing techniques help to produce a better product. Given below are some of the benefits of adopting a static testing approach:

- As defects are found and fixed, the quality of the product increases.
- A more technically correct base is available for each new phase of development.
- The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.
- The effectiveness of the dynamic test activity is increased and less time needs to be devoted for testing the product.
- Immediate evaluation and feedback to the author from his/her peers which will bring about improvements in the quality of future products.

The objectives of static testing can be summarized as follows:

- To identify errors in any phase of SDLC as early as possible
- To verify that the components of software are in conformance with its requirements
- To provide information for project monitoring
- To improve the software quality and increase productivity

Types of Static Testing

Static testing can be categorized into the following types:

- Software inspections
- Walkthroughs
- Technical reviews