Chapter 4

Path Testing

4.1 Basic Idea

Path testing is a kind of structural testing, which is performed by programmers to test code written by themselves. Path testing is applied to small *units* of code, such as a function. Soon it will be clear that effectiveness of path testing decreases as a programmer applies this to larger software aggregates, such as a collection of several functions. When we say test cases for path testing are derived from the source code, we mean test cases are derived from a *program unit*, such as a function or a method, for example, and not from an entire, large program. The fundamental structural concepts in a program unit are

- Control flow,
- Data flow,
- Path.

Control flow refers to the ordering of instruction execution in a program unit. Essentially, it means what is the next instruction to be executed. Data flow refers to the definition and use of variables in a program unit. Structurally, a path is a sequence of statements in a program unit. Semantically, a path is an execution instance of the unit. For a given set of input data, the program unit executes a certain path. For another set of input data, the program unit may execute another path. Control flow and data flow are said to occur along the executed path. The main idea in path testing is to appropriately select a few paths in a program unit, and observe whether or not the selected paths produce the expected outcome. By executing a few paths in a program unit, the programmer tries to conclude whether or not the entire program unit

behaves as expected. In fact, a path is any sequence of statements in a program unit. However, entry-exit paths are naturally selected, because of the difficulty associated with starting and stopping execution at arbitrary points in a program unit.

4.2 Outline of Path Testing

The process of generating test input data for performing path testing is depicted in Figure 4.1 and explained below.

Inputs: The inputs to the process of generating test data and the activities are the *source code* of the progarm unit and a set of *path selection criteria*. In the following, two examples of path selection criteria are given.

Example 1:	Select paths such that every statement is executed at least once.
Example 2:	Select paths such that every `if` statement once evaluates to *true* and once to *false*.

Generation of a Control Flow Graph (CFG): In the first step, a control flow graph is generated from the given program unit. A CFG is a detailed graphical representation of a program unit. The idea behind drawing a CFG is to clearly visualize all the paths in a program unit. The process of drawing a CFG from a program unit will be explained in the following section.

Selection of Paths: In this step, paths are selected from the CFG to satisfy the test selection criteria. Paths are selected by looking at the structure of the CFG.

Generation of Test Input Data: A path can be executed if and only if a certain instance of inputs to the program unit causes all the condition statements along the path to evaluate to *true*. Thus, it is essential to identify certain values of the inputs to the program unit from a given path for that path to execute.

Feasibility Test of a Path: A path is said to be *feasible* if and only if there exists an instance of the inputs to the progarm unit, which cause the condition statements along the path to evaluate to *true*. If there is no such instance of the inputs, then we call the path to be *infeasible*. The idea behind checking the feasibility of a selected path is to meet the path selection criteria. If certain paths are

found to be infeasible, then new paths are chosen to meet the selection criteria, which explains the loop in Figure 4.1.

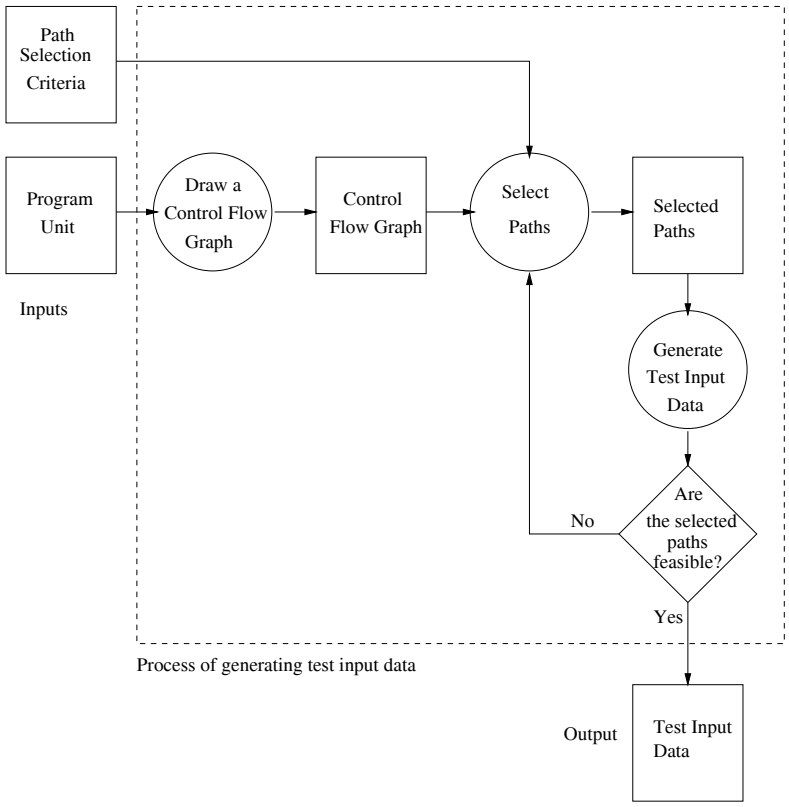

Figure 4.1: The process of generating test input data for path testing.

In the following sections, all the steps of the process of generating test input data from a program unit will be explained.

4.3 Control Flow Graph

A control flow graph is a graphical representation of a program unit. Three symbols are used to construct a control flow graph, as shown in Figure 4.2. The rectangle represents a sequential computation. A maximal sequential computation can be represented either by a single rectangle or by many rectangles, each corresponding to one statement in the program.

We label each computation and decision box with a unique integer value. The two branches of a decision box are labelled with **T** and **F** to represent the *true* and *false* evaluations of the condition inside the box, respectively. We will not label a merge node, because we can clearly identify paths in a control flow graph even without explicitly considering the merge nodes. Moreover, not mentioning the merge nodes in a path will make a path description shorter.

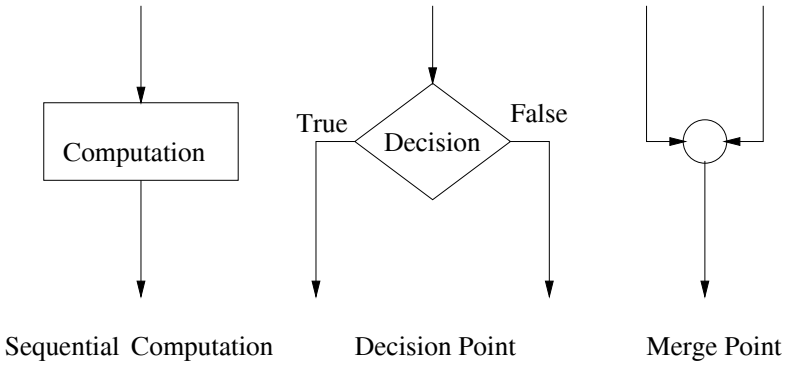

Figure 4.2: Symbols in a control flow graph.

To illustrate the process of drawing a control flow graph, let us consider the *openfiles()* function shown in Figure 4.3. This function has three statements: an assignment statement `int i = 0;`, a conditional statement `if()`, and a `return(i)` statement. The reader may note that irrespective of the evaluation of the condition in the `if()`, the function performs the same action, which is `null`. In Figure 4.4, we show a very high-level representation of control flow in *openfiles()* with three nodes numbered 1, 2, and 3. This control flow graph shows just two paths in *openfiles()*.

A closer examination of the condition part of the `if()` statement reveals that there are not only Boolean and relational operators in the condition part, but also assignment statements. Some of their examples are given below:

assignment statements:	`fptr1 = fopen("file1", "r")` and `i++`
relational operator:	`fptr1 != NULL`
Boolean operators:	`&&` and `||`.

Executions of the assignment statements in the condition part of the `if` statement depend on the component conditions. For example, consider the following component condition in the `if` part:

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
   /*
      This function tries to open files "file1", "file2", and "file3"
      for read access, and returns the number of files successfully
      opened. The file pointers of the opened files are put in the
      global variables.
   */
     int i = 0;
     if(
         ((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0)) ||
         ((( fptr2 = fopen("file2", "r")) != NULL) && (i++) && (0)) ||
         ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
     );
     return(i);
}
```

Figure 4.3: A function to open three files.

`((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0))`.

The above condition is executed as follows:

- Execute the assignment statement `fptr1 = fopen("file1", "r")`.
- Execute the relational operation `fptr1 != NULL`.
- If the above relational operator evaluates to *false*, skip the evaluation of the subsequent condition components `(i++) && (0)`.
- If the relational operator evaluates to *true*, then first `(i)` is evaluated to *true* or *false*. Irrespective of the outcome of this evaluation, the next statement executed is `(i++)`.
- If `(i)` has evaluated to *true*, then the following condition `(0)` is evaluated. Otherwise, evaluation of `(0)` is skipped.

In Figure 4.5, we show a detailed control flow graph for the *openfiles()* function.

4.4 Paths in a control flow graph

For the convenience of discussion, we assume that a control flow graph has exactly one *entry* node and exactly one *exit* node. Each node is tagged with a unique integer value. Also, the two branches of a decision node are labelled with *True*(T) or *False*(F) depending on the control flow. We are interested in identifying entry-exit paths in a control flow graph.

A path is represented as a sequence of computation and decision nodes from the entry node to the exit node. While including a decision node in a path, we also specify whether control exits the node via its *True* or *False* branch.

In Table 4.1, we show a few paths from the control flow graph of Figure 4.7. The reader may note that we have arbitrarily chosen these paths without applying any path selection criterion. In **Path 3**, we have unfolded the loop just once, whereas **Path 4** unfolds the same loop twice, and these are two distinct paths.

Path 1	**1-2-3(F)-10(T)-12-13**
Path 2	**1-2-3(F)-10(F)-11-13**
Path 3	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**
Path 4:	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**

Table 4.1: Examples of path in the control flow graph of Figure 4.7.

4.5 Path Selection Criteria

The objectives of defining path selection criteria are as follows.

- All program constructs are exercised at least once. The programmer needs to observe the outcome of executing each and every program construct, such as statements, Boolean conditions, and returns, to name a few of them.
- We do not generate test inputs which execute the same path again and again. Executing the same path several times is a waste of resources. However, if each execution of a program path potentially updates the *state* of the system, e.g. the database state, then multiple executions of the same path may not be identical.
- We know what program features we have tested and what we have not tested. For example, we may execute an `if` statement only

once so that it evaluates to *True*. If we do not execute it once again for its *False* evaluation, we are at least aware that we have not observed the program's outcome with a *False* evaluation of the `if`.

Now we explain the following well-known path selection criteria.

- Select *all* paths
- Select paths to achieve complete *statement* coverage
- Select paths to achieve complete *branch* coverage
- Select paths to achieve *predicate* coverage

4.5.1 All-path coverage criterion (P_∞)

If *all* paths are selected, then one can detect all faults, except those due to missing path errors. However, a program may contain a large number of paths, or even an infinite number of paths. The small, loop-free *openfiles()* function shown in Figure 4.3 contains more than 25 paths. Though only eight of all those paths are feasible, one does not know whether or not a path is feasible at the time of selecting those paths. If one selects all possible paths in a program, then we say that the *all-path* selection crietrion has been satisfied. We denote the *all-path* selection criterion by P_∞.

Now we consider the example of *openfiles()* function. *openfiles()* function tries to open the three files "file1", "file2", and "file3." The function returns an integer representing the number of files it has successfully opened. A file is said to be successfully opened with "read" access if the file exists. Therefore, there are three conceptual inputs variables to the function, namely the existence of those three files. The existence of a file is either "Yes" or "No". Thus, the input domain of the function consists of eight combinations of the existence of the three files as shown in Table 4.2.

For each input, that is each row of Table 4.2, of *openfiles()*, we can trace a path in the control flow graph of Figure 4.5. Ideally, we identify test inputs to execute a certain path in a program–and this will be explained in a subsequent part of this chapter. We give three examples of what test input executes what path in Table 4.5.1. In this manner, we can identify eight possible paths in Figure 4.5. Though the all-paths selection criterion is desirable from the viewpoint of detecting faults, it remains as an ideal criterion and may not be achievable in practice.

Existence of "file1"	Existence of "file2"	Existence of "file3"
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

Table 4.2: The input domain of *openfiles()*

.

Input	Path
$< No, No, No >$	**1-2-3(F)-8-9(F)-14-15(F)-19-21**
$< Yes, No, No >$	**1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21**
$< Yes, YesYes >$	**1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14-15(T)-16(T)-18-20-21**

Table 4.3: Inputs and paths in *openfiles()*

.

4.5.2 Statement coverage criterion (C1)

Statement coverage refers to executing individual program statements and observing its outcome. If *all* statements have been executed at least once, we say that 100% statement coverage has been achieved. Complete statement coverage is denoted by **C1**. Complete statement coverage is the *weakest* coverage criterion in program testing. Any test suite that achieves less than **C1** for new software is not acceptable. This is because the programmer must execute the program under various test scenarios to observe the outcome of executing each and every statement in the program.

All program statements are represented in some form in a control flow graph. Referring to the *ReturnAverage()* function and its control flow graph in Figures 4.6 and 4.7, respectively, the four assignment statements

```
i   = 0;
ti  = 0;
tv  = 0;
sum = 0;
```

have been represented by node **2**. The `while` statement has been represented as a loop, where the condition for loop control

```
(ti < AS && value[i] != -999)
```

has been represented by nodes **3** and **4**. Thus, covering a statement in a program means visiting one or more nodes representing the statement. To be precise, covering a statement means selecting a **feasible** entry-exit path that includes those nodes. Since a single entry-exit path includes many nodes, we need to select just a few paths to cover all the nodes of a control flow graph.

Therefore, the basic problem is to select a few feasible paths to cover all the nodes of a control flow graph in order to achieve the complete statement coverage criterion **C1**. While selecting paths, we follow these rules:

- Select short paths.
- Select paths of increasingly longer length. Unfold a loop several times, if there is a need.
- Select arbitrarily long, "complex" paths.

One can select the two paths shown in Figure 4.4 to achieve complete statement coverage.

SCPath 1	**1-2-3(F)-10(F)-11-13**
SCPath 2	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**

Table 4.4: Paths for statement coverage of the flow graph of Figure 4.7.

4.5.3 Branch coverage criterion (C2)

Syntactically, a branch is an outgoing edge from a node. All the rectangle nodes have at most one outgoing branch (edge)–the exit node of a control flow graph does not have an outgoing branch. All the diamond nodes have two outgoing branches or edges. Covering a branch means selecting

a path that includes the branch. Complete branch coverage, denoted by **C2**, means selecting a number of paths such that every branch is included in at least one path. Statement coverage guarantees that all the outgoing branches from the rectangle nodes are covered, but there is no guarantee that all the outgoing branches from the diamond nodes will be covered.

In a preceeding discussion, we showed that one can select two paths, **SCPath 1** and **SCPath 2** in Table 4.4, to achieve complete statement coverage. These two paths cover *all* the nodes (statements) and most of the branches of the control flow graph shown in Figure 4.7. The branches which are not covered by these two paths have been highlighted by bold dotted lines in Figure 4.8. These uncovered branches correspond to the three independent conditions

```
value[i] != -999,
value[i] >= MIN,
value[i] <= MAX
```

evaluating to *False*. The meaning of these uncovered branches is that as a programmer we have not observed the outcome of program execution as a result of the conditions evaluating to *False*. Thus, complete branch coverage means selecting enough number of paths such that every condition evaluates to *True* at least once and to *False*. at least once.

To cover the branches highlighted by the bold dotted lines in Figure 4.8, we need to select more paths. A set of paths for complete branch coverage is shown in Table 4.5.

BCPath 1	**1-2-3(F)-10(F)-11-13**
BCPath 2	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**
BCPath 3	**1-2-3(T)-4(F)-10(F)-11-13**
BCPath 4	**1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13**
BCPath 5	**1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13**

Table 4.5: Paths for branch coverage of the flow graph of Figure 4.7.

4.5.4 Predicate coverage criterion (C3)

To explain the concept of predicate coverage, first we refer to the partial control flow graph of Figure 4.9(a). *OB1*, *OB2*, *OB3*, and *OB* are four Boolean variables. First, the program computes the values of the individual variables *OB1*, *OB2* and *OB3*–details of their computation are

irrelevant to our discussion and have been omitted. Next, *OB* is computed as shown in the control flow graph. The control flow graph checks the value of *OB* and executes either *OBlock1* or *OBlock2* depending on whether *OB* evaluates to *True* or *False*, respectively.

We need to design just two test cases to achieve both statement coverage and branch coverage. To achieve this, we select inputs such that the four Boolean conditions in Figure 4.9(a) evaluate to values shown in Table 4.6. The reader may note that we have shown just one way of forcing *OB* to *True*. As a programmer if we select inputs so that these two cases hold, then we are not observing the effect of the computations taking place in nodes **2** and **3**. There may be faults in the computation parts of nodes **2** and **3** such that *OB2* and *OB3* always evaluate to *False*.

Therefore, there is a need to design test cases such that a path is executed under all possible conditions. Referring to Figure 4.9(a), the *False* branch of node **5** is executed under exactly one condition, namely when *OB1* = *False*, *OB2* = *False*, and *OB3* = *False*, whereas the *True* branch executes under *seven* conditions.

If all possible combinations of truth values of the conditions affecting a selected path have been explored under some tests, then we say that *predicate coverage* has been achieved. Therefore, the path taking the *True* branch of node **5** in Figure 4.9(a) must be executed for all *seven* possible combinations of truth values of *OB1*, *OB2*, and *OB3* which result in *OB* = *True*.

A similar situation holds for the partial control flow graph shown in Figure 4.9(b), where *AB1*, *AB2*, *AB3*, and *AB* are Boolean variables.

Cases	*OB1*	*OB2*	*OB3*	*OB*
Case 1	*T*	*F*	*F*	*T*
Case 2	*F*	*F*	*F*	*F*

Table 4.6: Two cases for complete statement and branch coverage of the control flow graph of Figure 4.9(a).

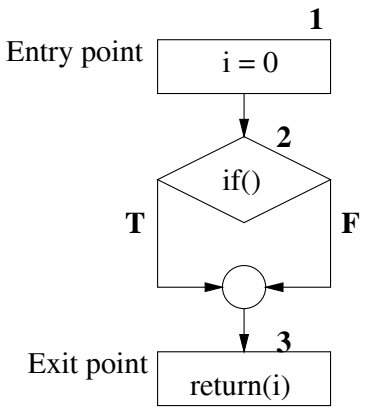

Figure 4.4: A high-level control flow graph representation of *openfiles()*.

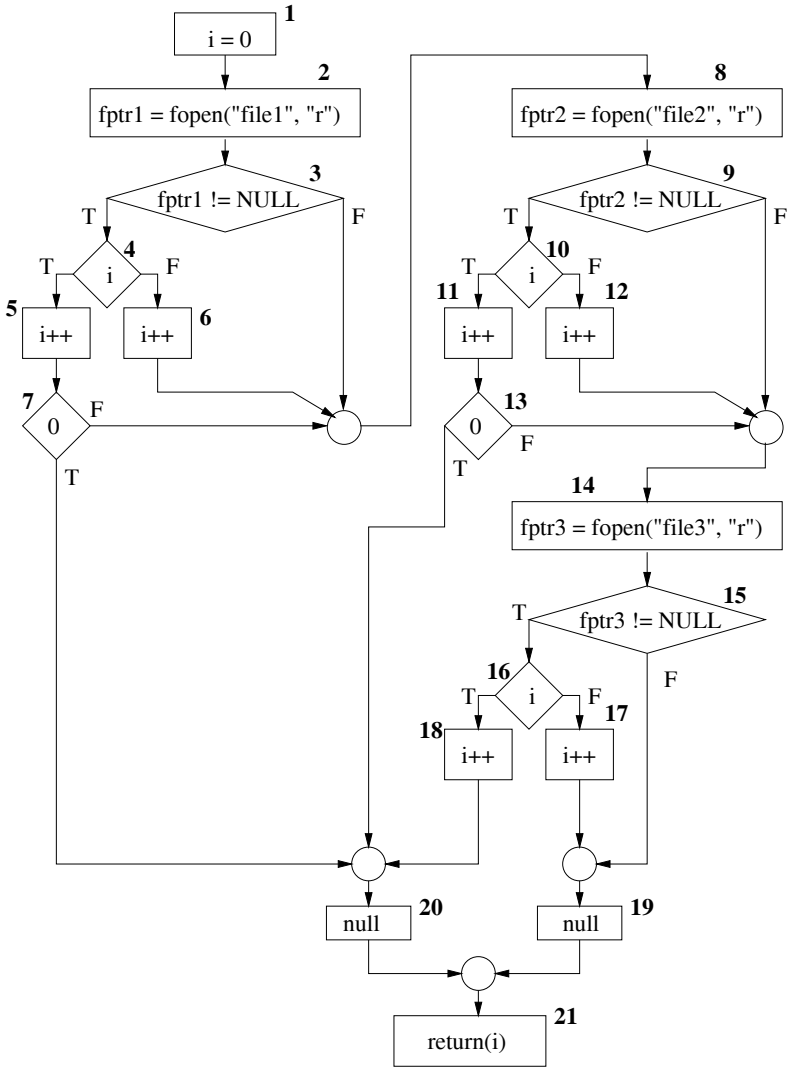

Figure 4.5: A detailed control flow graph representation of *openfiles()*.

```
public static double ReturnAverage(int value[],
                        int AS, int MIN, int MAX){
  /*
   Function: ReturnAverage  Computes the  average
   of all  those  numbers in the  input array  in
   the  positive  range  [MIN, MAX]. The  maximum
   size  of the array is AS. But, the  array size
   could be smaller than AS in which case the end
   of input is represented by -999.
  */

     int i, ti, tv, sum;
     double av;
     i = 0; ti = 0; tv = 0; sum = 0;
     while (ti < AS && value[i] != -999) {
        ti++;
        if (value[i] >= MIN && value[i] <= MAX) {
           tv++;
           sum = sum + value[i];
        }
        i++;
     }
     if (tv > 0)
        av = (double)sum/tv;
     else
        av = (double) -999;
     return (av);
}
```

Figure 4.6: A function to compute the average of selected integeres in an array.

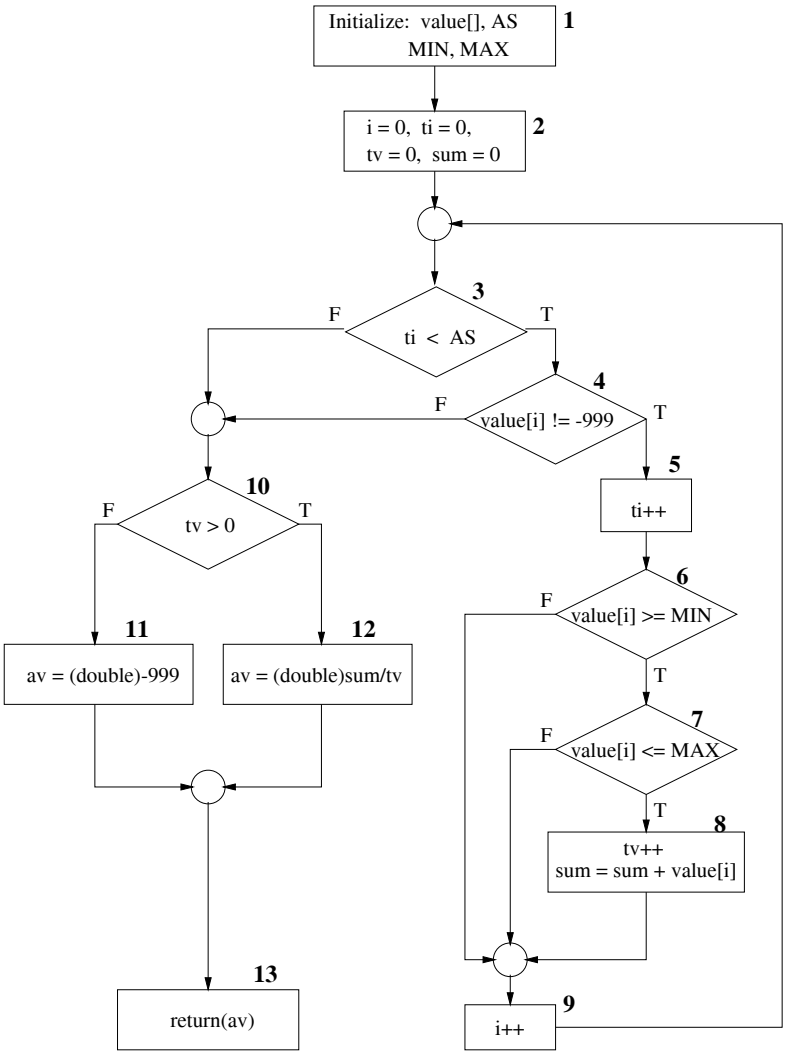

Figure 4.7: A control flow graph representation of *ReturnAverage()*.

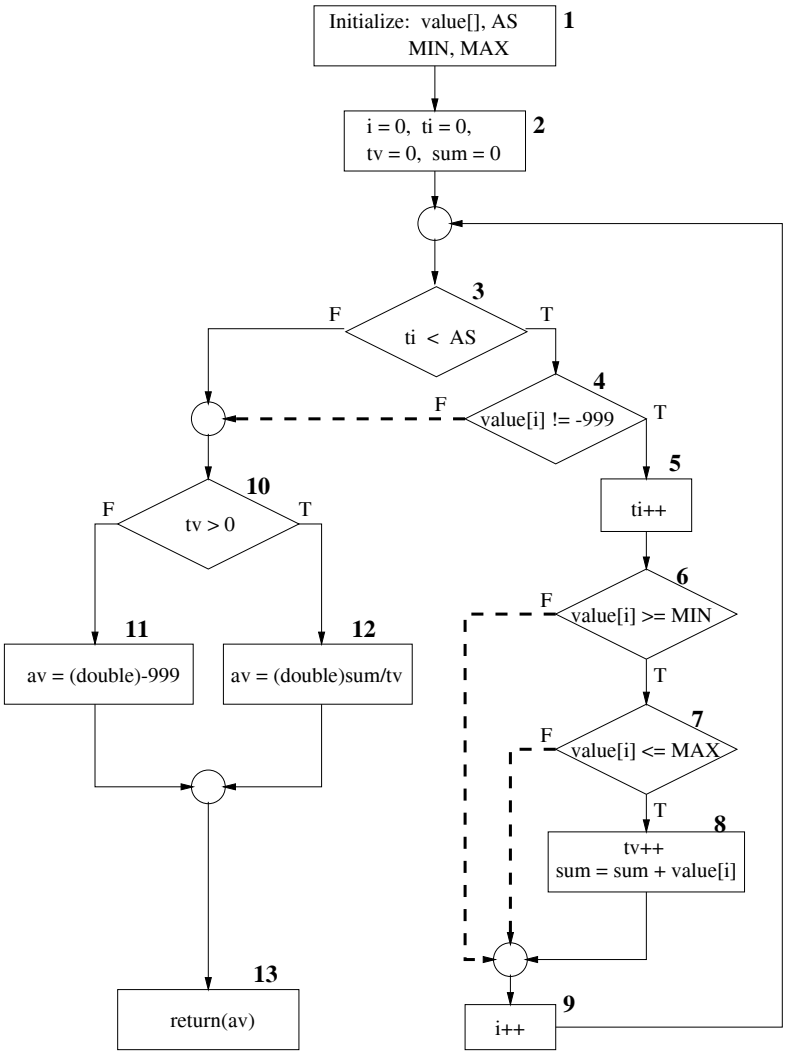

Figure 4.8: Branches not covered by statement coverage in *ReturnAverage()*.

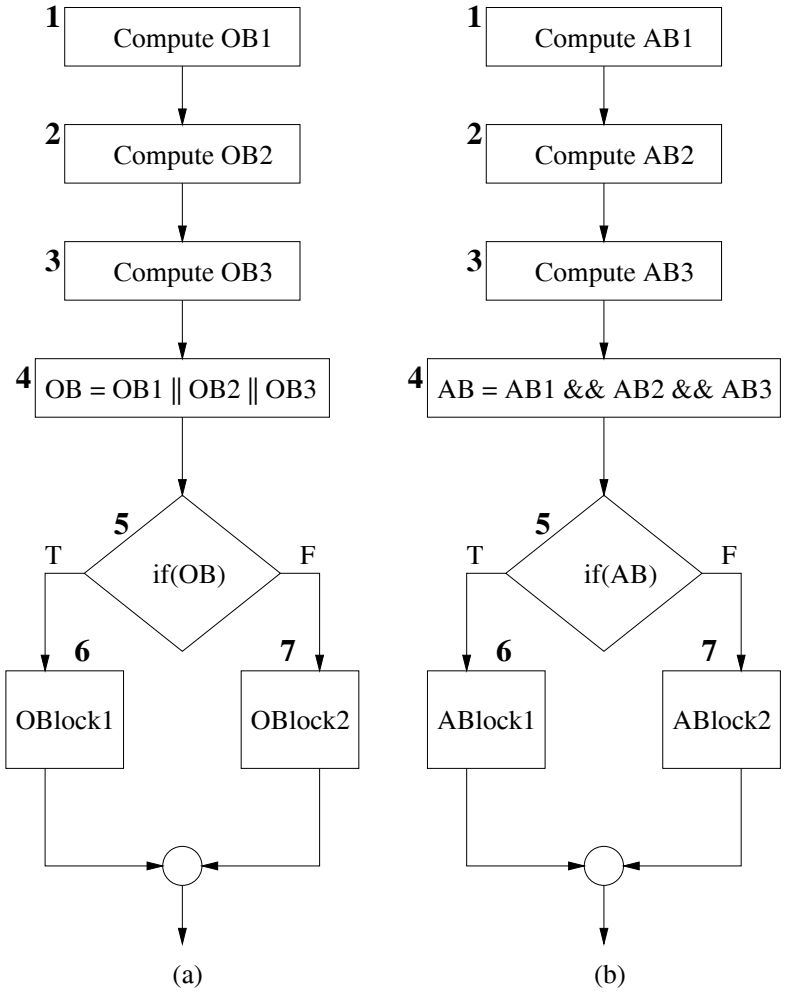

Figure 4.9: Partial control flow graph with OR operation (a) and AND operation (b).