CHAPTER 21

Control-Flow Testing

In this chapter, I cover some basics related to white-box testing of the control-flows in units. By control flows in units, I mean the sequence of execution of statements. In many cases, one statement follows the previous statement. However, in some cases the sequencing of statements is determined by decisions. Some of these decisions are made in branches like *if* statements and *switch* statements. Some of these decisions are made in loops like *for* statements and *while* statements and function calls.

Control-flow-based white-box testing is the cheapest way to find many types of coding bugs because many bugs are errors in control flows that exist entirely within a line or a few lines of a unit. Why wait until you have a "needle in a haystack" situation, where bugs lurk among millions of lines of code during system test? Early white-box unit testing by programmers can greatly increase the quality of the system and reduce the costs of poor quality later in the project.

If you are a tester, you might not find this section applicable to your daily work. However, I encourage you to skim this chapter and Chapter 23, on data-flow-based white-box testing. A basic familiarity with good unit testing practices can help you have an intelligent conversation with your programmer colleagues and assess the unit test coverage.

If you are a programmer, you'll find this chapter and the next two chapters helpful. The example programs run on either Windows or Linux systems or both, so you can get some hands-on, computer-based experience.

White-box tests are based on how the system works internally. To build them, you can examine the code to find the control flows through each unit.

You can also examine the code and data structures to find the data flows through each unit or between units. You can also look at application program interfaces, class member functions, or class methods to find how a component, library, or object interfaces with the rest of the system. This and the next couple of chapters look at these topics in more detail.

Code Coverage

Code-based or control-flow-based tests are often designed to achieve a particular level of code coverage. There are seven major ways to measure code coverage:

Statement coverage. You have achieved 100 percent statement coverage when you have executed every statement.

Branch (or decision) coverage. You have achieved 100 percent branch coverage when you have taken every branch (or decision) each way. For *if* statements, you'll need to make sure the expression that controls the branching evaluates both true and false. For *switch/case* statements, you'll need to cover each specified case as well as at least one unspecified case or the default case.

Condition coverage. Some branching decisions in programs are made not based on a single condition but on multiple conditions. You have achieved 100 percent condition coverage when you have evaluated behavior with each of these conditions both true and false. For example, if the conditional expression controlling an *if* statement is (A>0) && (B<0), then you need test both (A>0) true and false and (B<0) true and false. This can be done in two tests: true && true and false && false.

Multicondition coverage. When compound conditions apply to branching, you can move beyond condition coverage to multicondition coverage by requiring testing all possible combination of conditions. To continue our example in which the conditional expression controlling an *if* statement is (A>0) && (B<0), then you need to test all four possible combinations: true && true, true && false, false && true, and false && false.

Multicondition decision coverage. For languages like C++ where the subsequent conditions might not be evaluated depending on the preceding conditions, 100% condition coverage doesn't always make sense. In this case, you cover the conditions that can affect the decision made in terms of control flow. To finish our example in which the conditional expression controlling an *if* statement is (A>0) && (B<0), then you can test three combinations: true && true, true && false, and false && true. The extra combination, false && false, is now unneeded because the second condition cannot influence the decision.

Loop coverage. For loop paths, you must also iterate each loop zero, once, and multiple times. Ideally, you'll iterate the loop the maximum possible number of times. Sometimes it's possible to predict the maximum number of times a loop will iterate; sometimes it is not possible.

Path coverage. You have achieved 100 percent path coverage when you have taken all possible control paths. For functions with loops, path coverage is very difficult.

To some extent, you can rank these in order of increasing levels of coverage. If you achieved path coverage, you are guaranteed to have achieved loop, branch, and statement coverage (but not vice versa). If you achieved branch coverage, you are guaranteed to have achieved statement coverage (but not vice versa). If you achieved multicondition coverage, you are guaranteed to have achieved multicondition decision and condition coverage (but not vice versa). Notice that path coverage does not guarantee multicondition coverage, nor does multicondition coverage guarantee path coverage.

Let's look at an example. In Listing 21-1, you see a simple C program that runs on a Windows or Linux system. This program accepts an integer input and prints the factorial of that integer.

```
1.   main()
2.   {
3.      int i, n, f;
4.      printf("n = ");
5.      scanf("%d", &n);
6.      if (n < 0) {
7.           printf("Invalid: %d\n", n);
8.           n = -1;
9.      } else {
10.          f = 1;
11.          for (i = 1; i <= n; i++) {
12.            f *= i;
13.          }
14.          printf("%d! = %d.\n", n, f);
15.     }
16.     return n;
17. }
```

Listing 21-1 A program calculating factorials

What test values for *n* do you need to cover all the statements? Two values for *n* will work: *n* less than 0 and *n* greater than 0. Do those two values get you branch coverage? No, you also need to test *n* equal to zero to cover the "never iterate the loop" branch. Do those three values get you condition coverage? Yes, in this case, as there are no compound conditions in the program. How about loop coverage? Well, not quite. You need to cover *n* equal 1 and *n* equal to whatever the maximum number of iterations would be.

If you have access to a Windows or Linux system, give it a try.

What value for *n* did you use to achieve loop coverage? What happened to the factorial at the maximum value?

Here's what happened on my Windows 2000 system:

```
n = 12
12! = 479001600.
n = 13
13! = 1932053504.
```

However, the real answer for 13! is 6,227,020,800.

Is that a bug? If you understand how computers store integers, you can certainly see how the integer value of *f* would overflow eventually. However, there are other Windows applications, including the bundled calculator that comes with Windows, that can handle larger integers. And a routine for calculating factorials that only calculates the first 12 factorials isn't good for much.

This example illustrates one danger of white-box testing. I did not specify the expected results in advance. The output might appear reasonable, but is it correct? Beware of white-box unit testing without some reliable oracle that can help you determine test pass/fail status.

McCabe Cyclomatic Complexity

Another common kind of control-flow testing derives from work done by Thomas McCabe in the 1970s. McCabe created a metric called Cyclomatic Complexity, which measures program unit complexity in terms of control flows, specifically branching. You can measure Cyclomatic Complexity by drawing a directed graph to represent the unit under test. Nodes or bubbles represent entries, exits, and branching decision points. Edges or arrows represent nonbranching statements.

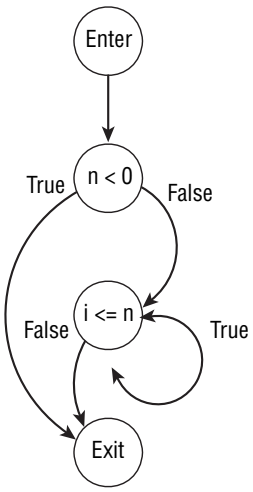
McCabe was trying to understand why programs are so hard to write and maintain. However, Cyclomatic Complexity has some useful testing implications. First, high-complexity modules are inherently buggy and regression prone. Second, the Cyclomatic Complexity number also tells you the number of basis paths through the graph, which is equal to the number of basis tests needed to cover the graph. McCabe basis test coverage is used as a testing-completeness standard for unit testing in some situations, including avionics software regulated by the Federal Aviation Administration.

Let's take a look at an example, using the factorial program again. In Figure 21-1, you can see how I turned the factorial program into a McCabe-style directed graph. I've used arrows to connect chunks of code with the nodes and edges that represent them.

Program

```
main()
{
  int i, n, f;
  printf("n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d.\n", n, f);
  }
   return n;
}
```

Flow Diagram



Cyclomatic Complexity

C = #R + 1 =
2 + 1 = 3

or

C = #E – #N + 2
5 – 4 + 2 = 3

Figure 21-1: Control flow graph for factorial program

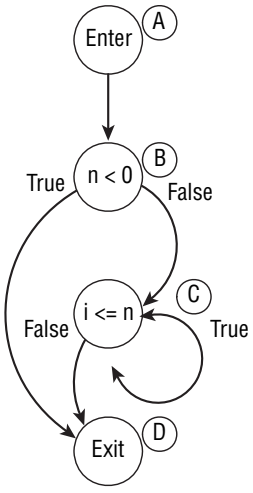
Given such a McCabe-style directed graph, you can calculate the McCabe Cyclomatic Complexity. It's the number of enclosed regions (R) in the graph plus one, or the number of edges (E) minus the number of nodes (N) plus two. For the factorial program, the Cyclomatic Complexity is 3. Since the Cyclomatic Complexity is 3, that means three basis paths, and thus three basis tests.

Figure 21-2 shows the basis paths and tests. I've lettered the nodes in the bubbles and described the basis paths in terms of the sequence of nodes traversed. You can arbitrarily pick values for *n* to cover the basis paths, though I used boundary value analysis.

Program

```
main()
{
  int i, n, f;
  printf("n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d.\n", n, f);
  }
   return n;
}
```

Flow Diagram



Basis Paths

1. ABD
2. ABCD
3. ABCCD

Basis Tests

	Input	Expect
1.	-1	Invalid: -1
2.	0	0! = 1.
3.	1	1! = 1.

Figure 21-2: Basis paths and tests for factorial program

If you compare the basis tests against the tests I needed to achieve branch coverage earlier in this chapter, you can see that McCabe basis tests satisfy branch coverage. This makes the McCabe technique a useful one when branch coverage is a requirement. It's also a quick way to ensure that you've covered the major control flows.[1]

[1] For more on the McCabe technique, see the report written by Thomas McCabe and Arthur Watson, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," available at `www.nist.gov`.