RBCS
TIME TESTED.
TESTING IMPROVED.
www.RBCS-US.com

# Structural Testing Techniques

*The following is an excerpt from Chapter 2 of the new edition of Advanced Software Testing: Volume 3, by Jamie Mitchell and Rex Black. Jamie is the primary author of the material in this chapter.*

Structure-based testing uses the internal structure of the system as a test basis for deriving dynamic test cases. In other words, we are going to use information about how the system is designed and built to derive our tests.

The question that should come to mind is why. We have all kinds of specification-based (black-box) testing methods to choose from. Why do we need more? We don't have time or resources to spare for extra testing, do we?

Well, consider a world-class, outstanding system test team using all black-box and experience-based techniques. Suppose they go through all of their testing, using decision tables, state-based tests, boundary analysis, and equivalence classes. They do exploratory and attack-based testing and error guessing and use checklist-based methods. After all that, have they done enough testing? Perhaps for some. But research has shown, that even with all of that testing, and all of that effort, they may have missed a few things.

There is a really good possibility that as much as 70 percent of all of the code that makes up the system might never have been executed once! Not once!

How can that be? Well, a good system is going to have a lot of code that is only there to handle the unusual, exceptional conditions that may occur. The happy path is often fairly straightforward to build—and test. And, if every user were an expert, and no one ever made mistakes, and everyone followed the happy path without deviation, we would not need to worry so much about testing the rest. If systems did not sometimes go down, and networks sometimes fail, and databases get busy and stuff didn't happen...

But, unfortunately, many people are novices at using software, and even experts forget things. And people do make mistakes and multi-strike the keys and look away at the wrong time. And virtually no one follows only the happy path without stepping off it occasionally. Stuff happens. And, the software must be written so that when weird stuff happens, it does not roll over and die.

To handle these less likely conditions, developers design systems and write code to survive the bad stuff. That makes most systems convoluted and complex. Levels of complexity are placed on top of levels of complexity; the resulting

 

system is usually hard to test well. We have to be able to look inside so we can test all of that complexity.

In addition, black-box testing is predicated on having models that expose the behaviors and list all requirements. Unfortunately, no matter how complete, not all behaviors and requirements are going to be visible to the testers. Requirements are often changed on the fly, features added, changed, or removed. Functionality often requires the developers to build "helper" functionality to be able to deliver features. Internal data flows that have asynchronous timing triggers often occur between hidden devices, invisible to black-box testers. Finally, malicious code—since it was probably not planned for in the specifications—will not be detected using black-box techniques. We must look under the covers to find it.

Much of white-box testing is involved with coverage, making sure we have tested everything we need to based on the context of project needs. Using white-box testing on top of black-box testing allows us to measure the coverage we got and add more testing when needed to make sure we have tested all of the important complexity we should. In this chapter, we are going to discuss how to do design, create, and execute white-box testing.

## Control Flow Testing Theory

Our first step into structural testing will be to discuss a technique called control flow testing. Control flow testing is done through control flow graphs, which provide a way to abstract a code module to better understand what it does. Control flow graphs give us a visual representation of the structure of the code. The algorithm for all control flow testing consists of converting a section of the code into a control graph and then analyzing the possible paths through the graph. There are a variety of techniques that we can apply to decide just how thoroughly we want to test the code. Then we can create test cases to test to that chosen level.

If there are different levels of control flow testing we can choose, we need to come up with a differentiator that helps us decide which level of coverage to choose. How much should we test? Possible answers range from no testing at all (a complete laissez-faire approach) to total exhaustive testing, hitting every possible path through the software. The end points are actually a little silly; no one is going to (intentionally) build a system and send it out without running it at least once. At the other end of the spectrum, exhaustive testing would require a near infinite amount of time and resources. In the middle, however, there is a wide range of coverage that is possible.

We will look at different reasons for testing to some of these different levels of coverage later in this chapter. In this section, we just want to discuss what these levels of control flow coverage are named.

At the low end of control flow testing, we have *statement* coverage. Common industry synonyms for *statement* are *instruction* and *code*. They all mean the same thing: have we exercised, at one time or another, every single line of executable code in the system? That would give us 100 percent statement coverage. It is possible to test less than that; people not doing white-box testing do it all the time, they just don't measure it. Remember, thorough black-box testing without doing any white-box testing may total less than 30 percent statement coverage, a number often discussed at conferences urging more white-box testing.

To calculate the total statement coverage actually achieved, divide the total number of executable statements into the number of statements that have actually been exercised by your testing. For example, if there are 1,000 total executable statements and you have executed 500 of them in your testing, you have achieved 50 percent statement coverage. This same calculation may be done for each different level of coverage we discuss in this chapter.

The next step up in control flow testing is called *decision* (or *branch*) coverage. This is determined by the total number of all of the outcomes of all of the decisions in the code that we have exercised. We will honor the ISTQB decision to treat *branch* testing and *decision* testing as synonyms. There are very slight differences between decision testing and branch testing, but those differences are insignificant at the level at which we will examine them.[1]

Then we have *condition* coverage, where we ensure that we evaluate each atomic condition that makes up a decision at least once, and *multiple condition* coverage, where we test all possible combinations of outcomes for individual atomic conditions inside all decisions.

We will add a level of coverage called *loop* coverage, not discussed by ISTQB but we think interesting anyway.

If this sounds complex, well, it is a bit. In the upcoming pages, we will explain what all of these terms and all the concepts mean. It is not as confusing as it might be—we just need to take it one step at a time and all will be clear. Each successive technique essentially adds more coverage than could be achieved by the previous technique.

---

[1] The United States Federal Aviation Administration makes a distinction between branch coverage and decision coverage with branch coverage deemed weaker. If you are interested in this distinction, see the FAA report *Software Verification Tools Assessment Study* (June 2007).

## Building Control Flow Graphs

Before we can discuss control flow testing, we must define how to create a control flow graph. In the next few paragraphs, we will discuss the individual pieces that make up control flow graphs.

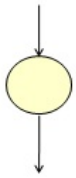

**Figure 1: The process block**

In **Figure 1**, we see the process block. Graphically, it consists of a node (bubble or circle) with one path leading to it and one path leading from it. Essentially, this represents a chunk of code that executes sequentially: that is, no decisions are made inside of it. The flow of execution reaches the process block, executes through that block of code in exactly the same way each time, and then exits, going elsewhere.

This concept is essential to understanding control flow testing. Decisions are the most significant part of the control flow concept; individual lines of code where no decisions are made do not affect the control flow and thus can be effectively ignored. The process block has *no* decisions made inside it. Whether the process block has one line or a million lines of code, we only need one test to execute it completely. The first line of code executes, the second executes, the third...right up to the millionth line of code. Excepting the possibility of an error (e.g., differing data may cause a divide by zero), there is no deviation no matter how many different test cases are run. Entry is at the top, exit is at the bottom, and every line of code executes every time.

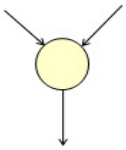

**Figure 2: The junction point**

The second structure we need to discuss is called a junction point, seen in **Figure 2**. This structure may have any number of different paths leading into the process block, with only one path leading out. No matter how many different paths we have throughout a module, eventually they must converge—or exit the module.

Again, no decisions are made in this block; all roads lead to it with only one road out.

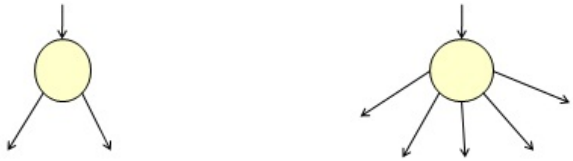

**Figure 3: Two decision points**

A decision point is very important; indeed, it's key to the concept of control flow. A decision point is represented as a node with one input and two or more possible outputs. Its name describes the action inside the node: a decision as to which way to exit is made and control flow continues out that path while ignoring all of the other possible choices. In **Figure 3**, we see two decision points: one with two outputs, one with five outputs.

How is the choice of output path made? Each programming language has a number of different ways of making decisions. In each case, a logical decision, based on comparing specific values, is made. We will discuss these later; for now, it is sufficient to say a decision is made and control flow continues one way and not others.

Note that these decision points force us to have multiple tests, at least one test for each way to make the decision differently, changing the way we traverse the code. In a very real way, it is the decisions that a computer can make that make it interesting, useful, and complex to test.

The next step is to combine these three relatively simple structures into useful control flow graphs.

 

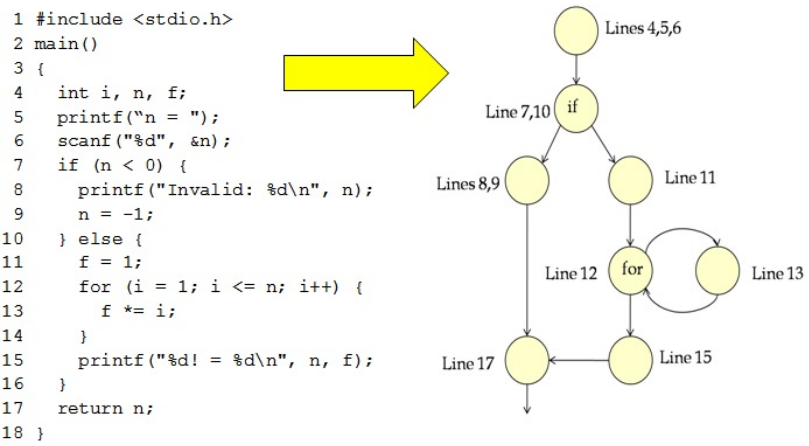


**Figure 4: A simple control flow graph**

In **Figure 4**, we have a small module of code. Written in C, this module consists of a routine to calculate a factorial. Just to refresh your memory, a factorial is the product of all positive integers less than or equal to n and is designated mathematically as n!. 0! is a special case that is explicitly defined to be equal to 1. The following are the first three factorials:

1! = 1

2! = 1 * 2 ==> 2

3! = 1 * 2 * 3 ==> 6 etc.

To create a control flow diagram from this code, we do the following:

1. The top process block contains up to and including line 6. Note there are no decisions, so all lines go into the same process block. By combining multiple lines of code where there is no decision made into a single process block, we simplify the graph. Note that we could conceivably have drawn a separate process block for each line of code.
2. At line 7 there is a reserved word in the C language: *if*. This denotes that the code is going to make a decision. Note that the decision can go one of two ways, but not both. If the decision resolves to TRUE, the left branch is taken and lines 8 and 9 are executed and then the thread jumps to line 17.
3. On the other hand, if the decision at line 7 resolves to FALSE, the thread jumps to line 10 to execute the *else* clause.
4. Line 11 sets a value and then falls through to line 12 where another decision is made. This line has the reserved word *for*, which means we may or may not loop.
5. At line 12, a decision is made in the *for* loop body using the second phrase in the statement: *(i <= n)*. If this evaluates to TRUE, the loop will fire, causing the code to go to line 13 where it calculates the value of *f* and then goes right back to the loop at line 12.

6. If the for loop evaluates to FALSE, the thread falls through to line 15 and thence to line 17.
7. Once the thread gets to line 17, the function ends at line 18.

To review the control flow graph: there are six process blocks, two decision blocks (lines 7 and 12), and two junction points (lines 12 and 17).

## Statement Coverage

Now that we have discussed control flow graphing, let's take a look at the first level of coverage we mentioned earlier, statement coverage (also called instruction or code coverage). The concept is relatively easy to understand: executable statements are the basis for test design selection. To achieve statement coverage, we pick test data that forces the thread of execution to go through each line of code that the system contains.

The first edition of this book was challenged by some who did not believe that statement coverage is a control flow design technique. We have decided that for logical consistency, and for the understanding of those who are just learning this subject, we will discuss statement coverage under control flow testing whether or not it strictly belongs here.

To calculate the current level of statement coverage that we have attained, divide the number of code statements that have been executed during the testing by the number of code statements in the entire module/system; if the quotient is equal to 1, we have 100 percent statement coverage.

The bug hypothesis is pretty much as expected; bugs can lurk in code that has not been executed.

Statement-level coverage is considered the least effective of all control flow techniques. In order to reach this modest level of coverage, a tester must come up with enough test cases to force every line to execute at least once. While this does not sound too onerous, it must be done purposefully. One good practice for an advanced technical test analyst is to make sure the developers they work with are familiar with the concepts we are discussing here. Achieving (at least) statement coverage can (and should) be done while unit testing.

There is certainly no guarantee that even college-educated developers learned how important this coverage level is. Jamie had an undergraduate computer science major and earned a master's degree in computer science from reputable colleges and yet was never exposed to any of this information until after graduation when he started reading test books and attending conferences. Rex can attest that the concepts of white-box coverage—indeed, test coverage in general—were not discussed when he got his degree in computer science and engineering at UCLA.

The Institute of Electrical and Electronics Engineers (IEEE), in the unit test standard ANSI 87B (1987), stated that statement coverage was the minimum level of coverage that should be acceptable. Boris Beizer, in his seminal work, *Software Testing Techniques* (ITC Press 1990), has a slightly more inflammatory take on it: "...testing less than this for new software is unconscionable and should be criminalized."

Also from that book, Beizer lays out some rules of common sense:

1. Not testing a piece of code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.
2. The high-probability paths are always thoroughly tested if only to demonstrate that the system works properly. If you have to leave some code untested at the unit level, it is more rational to leave the normal, high-probability paths untested, because someone else is sure to exercise them during integration testing or system testing.
3. Logic errors and fuzzy thinking are inversely proportional to the probability of the path's execution.
4. The subjective probability of executing a path as seen by the routine's designer and its objective execution probability are far apart. Only analysis can reveal the probability of a path, and most programmers' intuition with regard to path probabilities is miserable.
5. The subjective evaluation of the importance of a code segment as judged by its programmer is biased by aesthetic sense, ego, and familiarity. Elegant code might be heavily tested to demonstrate its elegance or to defend the concept, whereas straightforward code might be given cursory testing because "How could anything go wrong with that?"

Have we convinced you that this is important? Let's look at how to do it.

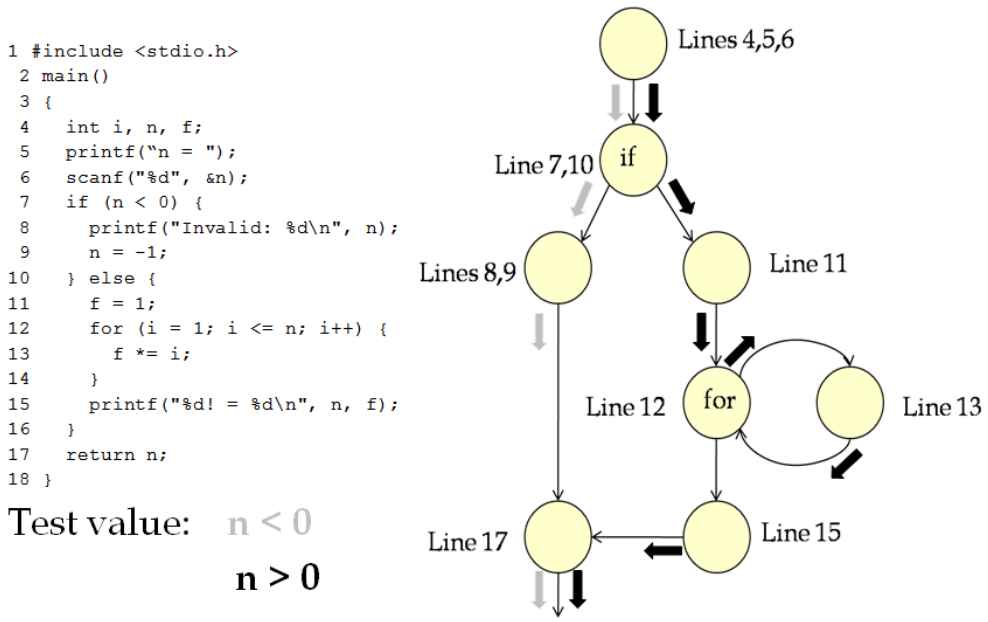


**Figure 5: Statement coverage example**

Looking at the same code and control flow graph we looked at earlier, **Figure 5** shows the execution of our factorial function when an input value less than zero is tested. The gray arrows show the path taken through the code with this test. Note that we have covered some of the lines of code, but not all. Based on the graph, we still have a way to go to reach statement coverage.

Following the black arrows, we see the result of testing with a value greater than 0. At this point, the *if* in line 7 resolves to FALSE and so we go down the untested branch. Because the input value is at least 1, the loop will fire at least once (more times if the input is greater than 1). When *i* increments and becomes larger than *n* (due to the third phrase in the parentheses, *i*++), the loop will stop and the thread of execution will fall through to line 15, line 17, and out.

Note that you do not really need the control flow graph to calculate this coverage; you could simply look at the code and see the same thing. However, for many people, it is much easier to read the graph than the code.

At this point, we have achieved statement coverage. If you are unsure, go back and look at **Figure 5** again. Every node (and line of code) has been covered by at least one arrow. We have run two test cases (n<0, n>0). At this point, we might ask ourselves if we are done testing. If we were to do more testing than we need, it is wasteful of resources and time. While it might be wonderful to always test enough that there are no more bugs possible, it would (even if it were possible) make software so expensive that no one could afford it.

 

Tools are available to determine if your testing has achieved 100 percent statement coverage. Coverage tools were introduced in the ISTQB Foundation level syllabus.

The real question we should ask is, Did we test enough for the context of our project? And the answer, of course, is that it depends on the project. Doing less testing than necessary will increase the risk of really bad things happening to the system in production. Every test group walks a balance beam in making the "do we need more testing?" decision.

Maybe we need to see where more coverage might come from and why we might need it. Possibly we can do a little more testing and get a lot better coverage.

1. z = 0;
2. if (a > b) then
3. z = 12;
4. Rep = 72/z;

Test 1: a = 3, b = 2 → Rep = 6

Test 2: **a = 2, b = 3 → Rep = ?**

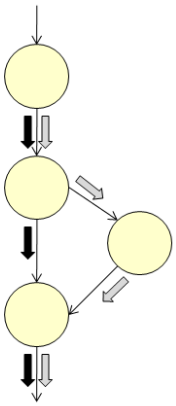

**Figure 6: Where statement coverage fails**

**Figure 6** shows a really simple piece of code and a matching control flow graph. Test case 1 is represented by the gray arrows; the result of running this single test with the given values is full 100 percent statement coverage. Notice on line 2, we have an *if* statement that compares *a* to *b*. Since the inputted values (*a = 3, b = 2*), when plugged into this decision, will evaluate to TRUE, the line *z = 12* will be executed, and the code will fall out of the *if* statement to line 4, where the final line of code will execute and set the variable *Rep* to 6 by dividing 72 by 12.

A few moments ago, we asked if statement coverage was enough testing. Here, we can see a potential problem with stopping at only statement coverage. Notice that, if we pass in the values shown as test 2 (*a = 2, b = 3*), we have a different decision made at the conditional on line 2. In this case, *a* is not greater than *b* (i.e., the decision resolves to FALSE) and line 3 is not executed. No big deal, right. We still fall out of the *if* to line 4. There, the calculation *72/z* is performed, exactly the way we would expect. However, there is a nasty little surprise at this point. Since

*z* was not reset to 12 inside the branch, it remained set to 0. Therefore, expanding the calculation performed on line 4, we have 72 divided by 0.

Oops!

You might remember from elementary school that anything divided by 0 is not defined. Seriously, in our mathematics, it is simply not allowed. So what should the computer do at this point? If there is an exception handler somewhere in the execution stack, it should fire, unwinding all of the calculations that were made since it was set. If there is no exception handler, the system may crash—hard! The processor that our system is running on likely has a "hard stop" built into its micro-code for this purpose. In reality, most operating systems are not going to let the CPU crash with a divide by zero failure—but the OS will likely make the offending application disappear like a bad dream.

But, you might say, we had statement coverage when we tested! This just isn't fair! As we noted earlier, statement coverage by itself is a bare minimum coverage level that is almost guaranteed to miss defects. We need to look at another, higher level of coverage that can catch this particular kind of defect.

## Decision Coverage

The next strongest level of structural coverage is called decision (or branch) coverage.

Rather than looking at individual statements, this level of coverage looks at the decisions themselves. Every decision has the possibility of being resolved as either TRUE or FALSE. No other possibilities. Binary results, TRUE or FALSE. For those who point out that the switch statement can make more than two decisions, well conceptually that appears to be true. The switch statement is a complex set of decisions, often built as a table by the compiler. The generated code, however, is really just a number of binary compares that continue sequentially until either a match is found or the default condition is reached. Each atomic decision in the switch statement is still a comparison between two values that evaluates either TRUE or FALSE.

To get to the 100 percent decision level of coverage, every decision made by the code must be resolved both ways at one time or another, TRUE and FALSE. That means—at minimum—two test cases must be run, one with data that causes the evaluation to resolve TRUE and a separate test case where the data causes the decision to resolve FALSE. If you omit one test or the other, then you do not achieve 100 percent decision coverage.

Our bug hypothesis was proved out in **Figure 6**. An untested branch may leave a land mine in the code that can cause a failure even though every line was executed at least once. The example we went over may seem too simplistic to be

true, but this is exactly what often happens. The failure to set (or reset) a value in the conditional causes a problem later on in the code.

Note that if we execute each decision both ways, giving us decision coverage, it guarantees statement coverage in the same code. Therefore, decision coverage is said to be stronger than statement coverage. Statement coverage, as the weakest coverage, does not guarantee anything beyond each statement being executed at least once.

As before, we could calculate the exact level of decision coverage by dividing the number of decision outcomes tested by the total number of decision outcomes in the module/system. For this book, we are going to speak as if we always want to achieve full—that is 100 percent—decision coverage. In real life, your mileage might vary. There are tools available to measure decision coverage. However, we have found that some of the tools that claim to calculate decision coverage do not work correctly; you should verify the correctness of the tool you are using if it matters to your organization.

Having defined this higher level of coverage, let's dig into it.

The ability to take one rather than the other path in a computer is what really makes a computer powerful. Each computer program likely makes millions of decisions a minute. But exactly what is a decision?

As noted earlier, each predicate eventually must resolve to one of two values, TRUE or FALSE. As seen in our code, this might be a really simple expression: ***if (a > b)*** or ***if (n < 0)***. However, we often need to consider more complex expressions in a decision. In fact, a decision can be arbitrarily complex, as long as it eventually resolves to either TRUE or FALSE. The following is a legal expression that resolves to a single Boolean value:

*(a>b) || (x+y==-1) && ((d) != TRUE)*

This expression has three subexpressions in it, separated by the Boolean operators || and &&. The first is an *OR* operator and the second is an *AND* operator. While the *AND* operator has a slightly higher precedence for evaluation than the *OR*, the rule for evaluation of Boolean operators in the same expression is to treat them as if they have equal precedence and to evaluate them according to their associativity (i.e., left to right). Only parentheses would change the order of evaluation. Therefore, the first subexpression to be evaluated would be the leftmost, (a>b). This will evaluate to either TRUE or FALSE based on the actual runtime values. Next, the subexpression (x+y==-1*)* will evaluate to either TRUE or FALSE based on the runtime values. This result will be ORed to the first result and stored for a moment. Finally, the last subexpression, ((d) != TRUE),

 

will evaluate to either TRUE or FALSE based on the value of *d* and will then be ANDed to the previous result.[2]

While this expression appears to be ugly, the compiler will evaluate the entire predicate—step-by-step—to inform us whether it is a legal expression or not. This actually points out something every developer should do: *write code that is understandable!* Frankly, we would not insult the term *understandable* by claiming this expression is!

There are a variety of different decisions that a programming language can make.

| Either/or Decisions | Loop Decisions |
|---|---|
| `if (expr)`<br>`    {}`<br>`else`<br>`    {}` | `while(expr)`<br>`   {}` |
| `switch (expr) {`<br>`    case const_1: {}`<br>`break;`<br>`    case const_2: {}`<br>`break;`<br>`    case const_3: {}`<br>`break;`<br>`    case const_4: {}`<br>`break;`<br>`    default {}`<br>`}` | `do`<br>`   {}`<br>`while (expr)`<br>`for (expr_1; expr_2; expr_3)`<br>`   {}` |

**Table 1: C language decision constructs**

Here, in Table 1, you can see a short list of the decision constructs that the programming language C uses. Other languages (including C++, C#, and Java) use similar constructs. The commonality between all of these (and all of the other

[2] This may not be strictly true if the compiler optimizes the code and/or short-circuits it. We will discuss those topics later in the chapter. For this discussion, let's assume that it is true.

decision constructs in all imperative programming languages) is that each makes a decision that can go only two ways: TRUE or FALSE.

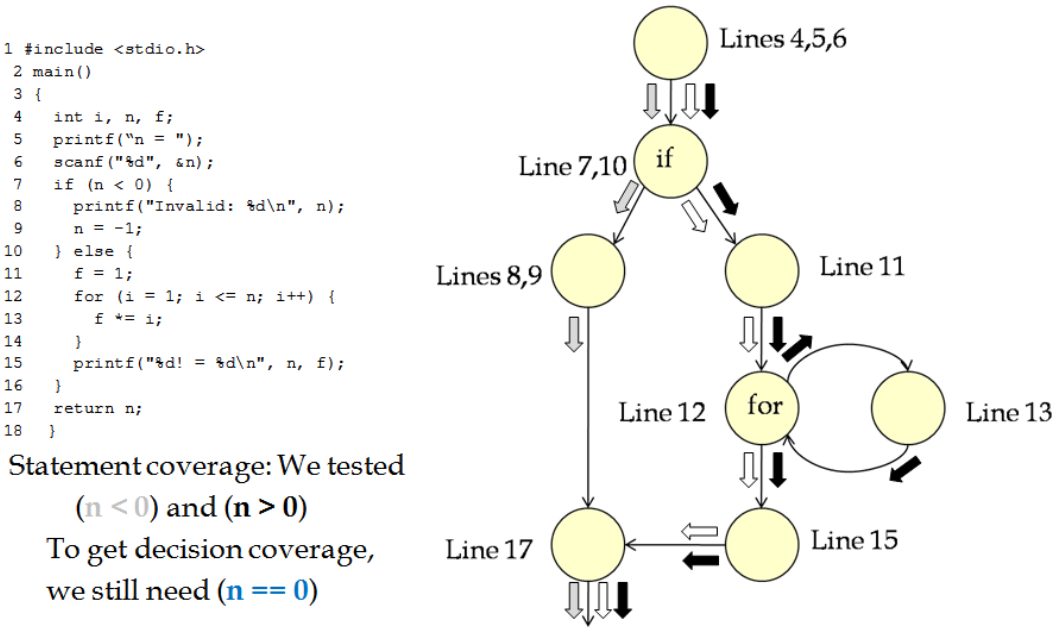


**Figure 7: Decision coverage example**

Looking back at our original example, with just two test cases we did not achieve decision coverage, even though we did attain statement coverage. We did not *not* execute the *for* loop during the test. Remember, a *for* loop evaluates an expression and decides whether to loop or not to loop based on the result.

In order to get decision coverage, we need to test with the value 0 inputted, as shown in **Figure 7**. When 0 is entered, the first decision evaluates to FALSE, so we take the *else* path. At line 12, the predicate (is 1 less than or equal to 0) evaluates to FALSE, so the loop is not taken. Recall that earlier, we had tested with a value greater than 0, which did cause the loop to execute. Now we have achieved decision coverage; it took three test cases.

At the risk of being pedantic, when the test was *(n > 0)*, causing the loop to fire, it did eventually run down and not execute the *for* loop. When discussing a loop, there will always be a time that the loop ends and does fall out (or else we have an infinite loop, which brings other problems). That time after the final loop does not count as not taking the loop. We need a test that never takes the loop to achieve decision coverage.

That brings us back to the same old question. Having tested to the decision level of coverage, have we done enough testing?

 

Consider the loop itself. We executed it zero times when we inputted 0. We executed it an indeterminate number of times when we inputted a value greater than zero. Is that sufficient testing?

Well, not surprisingly, there is another level of coverage called loop coverage. We will look at that next.

## Loop Coverage

While not discussed in the ISTQB Advanced syllabus, loop testing should be considered an important control flow, structural test technique. If we want to completely test a loop, we would need to test it zero times (i.e., did not loop), one time, two times, three times, all the way up to *n* times where it hits the maximum it will ever loop. Bring your lunch; it might be an infinite wait!

Like other exhaustive testing techniques, full loop testing does not provide a reasonable amount of coverage. Different theories have been offered that try to prescribe how much testing we should give a loop. The basic minimum tends to be two tests, zero times through the loop and one time through the loop (giving us decision coverage). Others add a third test, multiple times through the loop, although they do not specify how many times. We prefer a stronger standard; we suggest that you try to test the loop zero and one time and then test the maximum number of times it is expected to cycle (if you know how many times that is likely to be). In a few moments, we will discuss Boris Bezier's standard, which is even more stringent.

We should be clear about loop testing. Some loops could execute an infinite number of times; each time through the loop creates one or more extra paths that could be tested. In the Foundation syllabus, a basic principle of testing was stated: "Exhaustive testing is impossible." Loops, and the possible infinite variations they can lead to, are the main reason exhaustive testing is impossible. Since we cannot test all of the ways loops will execute, we want to be pragmatic here, coming up with a level of coverage that gives us the most information in the least amount of tests. Our bug hypothesis is pretty clear; failing to test the loop (especially when it does not loop even once) may cause bugs to be shipped to production.

In the next few paragraphs, we will discuss how to get loop coverage in our factorial example. Here in **Figure 8**, following the gray arrows, we satisfy the first leg of loop coverage, zero iterations. Entering a zero, as we discussed earlier, allows the loop to be skipped without causing it to loop. In line 12, the *for* loop condition evaluates as *(1 is less than or equal to 0),* or FALSE. This causes us to drop out of the loop without executing it.

 

```
1 #include <stdio.h>
 2 main()
 3 {
 4   int i, n, f;
 5   printf("n = ");
 6   scanf("%d", &n);
 7   if (n < 0) {
 8     printf("Invalid: %d\n", n);
 9     n = -1;
10   } else {
11     f = 1;
12     for (i = 1; i <= n; i++) {
13       f *= i;
14     }
15     printf("%d! = %d\n", n, f);
16   }
17   return n;
18 }
```

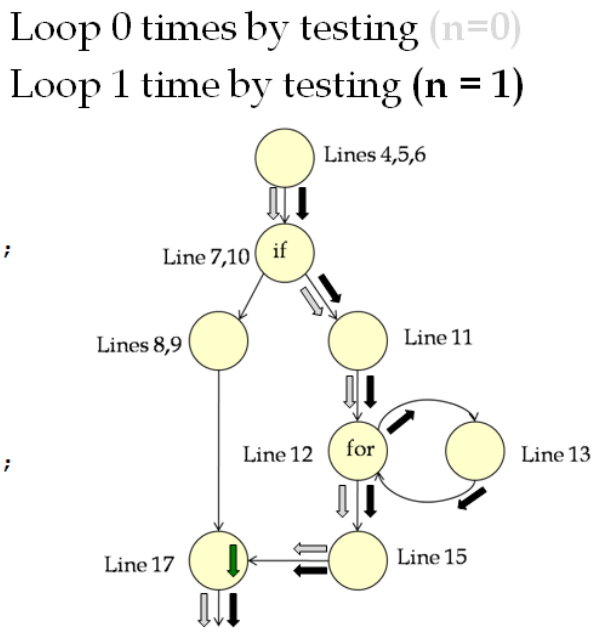


**Figure 8: Loop coverage 0 and 1 times through**

By entering a 1 (following the black arrows), we get the loop to execute just once. Upon entering line 12, the condition evaluates to *(1 is less than or equal to 1,* or TRUE. The loop executes and *i* is incremented. After the loop (line 13) is executed, the condition is evaluated again, *(2 is less than or equal to 1),* or FALSE. This causes us to drop out of the loop.

Zero and one time through the loop are relatively easy to come up with. How about the maximum times through? Each loop is likely to have a different way of figuring that out, and for some loops, it will be impossible to determine.

In this code, we have a monotonically increasing value which makes it easy to calculate the greatest number of loops possible. The maximum size of the data type used in the collector variable, *f*, will allow us to calculate the maximum number of iterations. We need to compare the size of the calculated factorial against the maximum integer size (the actual data type would have to be confirmed in the code).

In Table 2, we show a table with factorial values.

 

| n | n! |
|---|---|
| **0** | 1 |
| **1** | 1 |
| **2** | 2 |
| **3** | 6 |
| **4** | 24 |
| **5** | 120 |
| **6** | 720 |
| **8** | 40,320 |
| **9** | 362,880 |
| **10** | 3,628,800 |
| **11** | 39,916,800 |
| **12** | 479,001,600 |
| **13** | 6,227,020,800 |
| **14** | 87,178,291,200 |
| **15** | 1,307,674,368,000 |

**Table 2: Factorial values**

Assuming a signed 32-bit integer being used to hold the calculation, the maximum value that can be stored is 2,147,483,647. An input of 12 should give us a value of 479,001,600. An input value of 13 would cause an overflow (6,227,020,800). If the programmer used an unsigned 32-bit integer with a maximum size of 4,294,967,295, notice that the same number of iterations would be allowed; an input of 13 would still cause an overflow.

In **Figure 9**: Loop coverage max times, we would go ahead and test the input of 12 and check the expected output of the function.

 

```
1 #include <stdio.h>
 2 main()
 3 {
 4   int i, n, f;
 5   printf("n = ");
 6   scanf("%d", &n);
 7   if (n < 0) {
 8     printf("Invalid: %d\n", n);
 9     n = -1;
10   } else {
11     f = 1;
12     for (i = 1; i <= n; i++) {
13       f *= i;
14     }
15     printf("%d! = %d\n", n, f);
16   }
17   return n;
18 }
```

Loop 12 times by testing **(n==12)**

Loop 13 times and it overflows

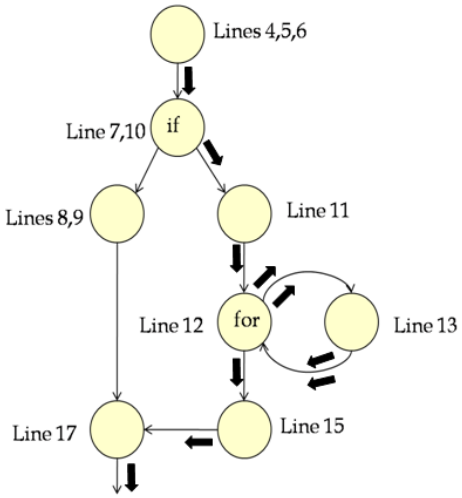


**Figure 9: Loop coverage max times through**

It should be noted that our rule for loop coverage does not deal comprehensively with negative testing. Remember that negative testing is checking invalid values to make sure the failure they cause is graceful, giving us a meaningful error message and allowing us to recover from the error. We probably want to test the value 13 to make sure the overflow is handled gracefully. This is consistent with the concept of boundary value testing discussed in the ISTQB Foundation level syllabus.

Boris Beizer, in his book *Software Testing Techniques*, had suggestions for how to test loops extensively. Note that he is essentially covering the three point boundary values of the loop variable with a few extra tests thrown in.

1. If possible, test a value that is one less than the expected minimum value the loop can take. For example, if we expect to loop with the control variable going from 0 to 100, try -1 and see what happens.
2. Try the minimum number of iterations—usually zero iterations. Occasionally, there will be a positive number as the minimum.
3. Try one more than the minimum number.
4. Try to loop once (this test case may be redundant and should be omitted if it is).
5. Try to loop twice (this might also be redundant).
6. A typical value. Beizer always believes in testing what he often calls a nominal value. Note that the number of loops, from one to max, is actually an equivalence set. The nominal value is often an extra test that we tend to leave out.

 

7. One less than the maximum value.
8. The maximum number of values.
9. One more than the maximum value.

The most important differentiation between Beizer's guidelines and our loop coverage described earlier is that he advocates negative testing of loops. Frankly, it is hard to argue against this thoroughness. Time and resources, of course, are always factors that we must consider when testing. We would argue that his guidelines are useful and should be considered when testing mission-critical or safety-critical software.

Finally, one of the banes of testing loops is when they are nested inside each other. We will end this topic with Beizer's advice for reducing the number of tests when dealing with nested loops.

1. Starting at the innermost loop, set all outer loops to minimum iteration setting.
2. Test the boundary values for the innermost loop as discussed earlier.
3. If you have done the outermost loop already, go to step 5.
4. Continue outward, one loop at a time until you have tested all loops.
5. Test the boundaries of all loops simultaneously. That is, set all to 0 loops, 1 loop, maximum loops, 1 more than maximum loops.

Beizer points out that practicality may not allow hitting each one of these values at the same time for step 5. As guidelines go, however, these will ensure pretty good testing of looping structures.

## Condition Coverage

So far we have looked at statement coverage (have we exercised every line of code?), decision coverage (have we exercised each decision both ways?), and loop coverage (have we exercised the loop enough?).

As Ron Popeil, the purveyor of many a fancy gadget in all-night infomercials used to say, "But wait! There's more!"

While we have exercised the decision both ways, we have yet to discuss how the decision is made. Earlier, when we discussed decisions, you saw that they could be relatively simple, such as

$A > B$

or arbitrarily complex–such as

*(a>b) || (x+y==-1) && (d) != TRUE*

If we input data to force the entire condition to TRUE, then we go one way; force it FALSE and we go the other way. At this point, we have achieved decision coverage. But is that enough testing?

Could there be bugs hiding in the evaluation of the condition itself? The answer, of course, is a resounding yes! And, if we know there might be bugs there, we might want to test more.

Our next level of coverage is called condition coverage. The basic concept is that, when a decision is made by a complex expression that eventually evaluates to TRUE or FALSE, we want to make sure each atomic condition is tested both ways, TRUE and FALSE.

An atomic condition is defined as "the simplest form of code that can result in a TRUE or FALSE outcome."[3]

Our bug hypothesis is that defects may lurk in untested atomic conditions, even though the full decision has been tested both ways. As always, test data must be selected to ensure that each atomic condition actually be forced TRUE and FALSE at one time or another. Clearly, the more complex a decision expression is, the more test cases we will need to execute to achieve this level of coverage.

Once again, we could evaluate this coverage for values less than 100 percent, by dividing the number of Boolean operand values executed by the total number of Boolean operand values there are. But we won't. Condition coverage, for this book, will only be discussed as 100 percent coverage.

Note an interesting corollary to this coverage. Decision and condition coverage will always be exactly the same when all decisions are made by simple atomic expressions. For example, for the conditional, *if (a == 1) {}*, condition coverage will be identical to decision coverage.

Here, we'll look at some examples of complex expressions, all of which evaluate to TRUE or FALSE.

*x*

The first, shown above, has a single Boolean variable, *x*, which might evaluate to TRUE or FALSE. Note that in some languages, it does not even have to be a Boolean variable. For example, if this were the C language, it could be an integer, because any non-zero value constitutes a TRUE value and zero is deemed to be FALSE. The important thing to notice is that it is an atomic condition and it is also the entire expression.

*D && F*

The second, shown above, has two atomic conditions, *D* and *F*, which are combined together by the AND operator to determine the value of the entire expression.

*(A || B) && (C == D)*

---

[3] Definition from *The Software Test Engineer's Handbook* by Graham Bath and Judy McKay (Rocky Nook, 2014).

The third is a little tricky. In this case, *A* and *B* are both atomic conditions, which are combined together by the OR operator to calculate a value for the subexpression (*A* || *B*). Because *A* and *B* are both atomic conditions, the subexpression (*A* || *B*) cannot be an atomic condition. However, (*C* == *D*) is an atomic condition because it cannot be broken down any further. That makes a total of three atomic conditions: *A*, *B*, (*C* == *D*).

*(a>b)||(x+y==-1)&&((d)!=TRUE)*

In the last complex expression, shown above, there are again three atomic conditions. The first, *(a>b)*, is an atomic condition that cannot be broken down further. The second, *(x+y == -1)*, is an atomic condition following the same rule. In the last subexpression, *(d!=TRUE)* is the atomic condition.

Just for the record, if we were to see the last expression in actual code during a code review, we would jump all over the expression with both feet. Unfortunately, it is an example of the way some people program.

In each of the preceding examples, we would need to come up with sufficient test cases that each of these atomic conditions was tested where they evaluated both TRUE and FALSE. Should we test to that extent, we would achieve 100 percent condition coverage. That means the following:

- *x*, *D*, *F*, *A*, and *B* would each need a test case where it evaluated to TRUE and one where it evaluated to FALSE.
- *(C==D)* needs two test cases.
- *(a>b)* needs two test cases.
- *(x+y==-1)* needs two test cases.
- *((d)!=TRUE)* needs two test cases.

Surely that must be enough testing. Maybe, maybe not. Consider the following pseudo code:

```
if (A && B) then
     {Do something}
else
     {Do something else}
```

In order to achieve condition coverage, we need to ensure that each atomic condition goes both TRUE and FALSE in at least one test case each.

Test 1*: A* == FALSE, *B* == TRUE resolves to FALSE

Test 2*: A* == TRUE, *B* == FALSE resolves to FALSE

Assume that our first test case has the values inputted to make *A* equal to FALSE and *B* equal to TRUE. That makes the entire expression evaluate to FALSE, so we execute the *else* clause. For our second test, we reverse that so *A* is set to TRUE

 

and *B* is set to FALSE. That evaluates to FALSE so we again execute the *else* clause.

Do we now have condition coverage? *A* was set both TRUE and FALSE, as was *B*. Sure, we have achieved condition coverage. But ask yourself, do we have decision coverage? The answer is no! At no time, in either test case, did we force the predicate to resolve to TRUE. Condition coverage does not automatically guarantee decision coverage.

That is an important point. We made the distinction that decision coverage was *stronger* than statement coverage because 100 percent decision coverage implied 100 percent statement coverage (but not vice versa). In the 2011 ISTQB Foundation level syllabus, section 4.4.3 has the following sentence: "There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and …"

As we showed in the previous example, condition coverage is *not* stronger than decision coverage because 100 percent condition coverage does not imply 100 percent decision coverage. That makes condition coverage—by itself—not very interesting for structural testing. However, it is still an important test design technique; we just need to extend the technique a bit to make it stronger.

We will look at three more levels of coverage that will make condition coverage stronger—at the cost of more testing, of course.

## Decision Condition Coverage

The first of these we will call decision condition coverage. This level of coverage is just a combination of decision and condition coverage (to solve the shortcoming of condition only coverage pointed out in the preceding section). The concept is that we need to achieve condition level coverage where each atomic condition is tested both ways, TRUE and FALSE, *and* we also need to make sure that we achieve decision coverage by ensuring that the overall predicate be tested both ways, TRUE and FALSE.

To test to this level of coverage, we ensure that we have condition coverage, and then make sure we evaluate the full decision both ways. The bug hypothesis should be clear from the preceding discussion; not testing for decision coverage even if we have condition coverage may allow bugs to remain in the code.

Full decision condition coverage guarantees condition, decision, and statement coverage and thereby is stronger than all three.

Going back to the previous example, where we have condition coverage but not decision coverage for the predicate (A && B), we already had two test cases as follows:

*A* is set to FALSE and *B* is set to TRUE.

*A* is set to TRUE and *B* is set to FALSE.

We can add a third test case:

Both *A* and *B* are set to TRUE.

We now force the predicate to evaluate to TRUE, so we now have decision coverage also. An alternative would be to select our original test cases a bit more wisely, as follows:

*A* and *B* are both set to TRUE.

*A* and *B* are both set to FALSE.

In this case, we can achieve full decision condition coverage with only two test cases.

Whew! Finally, with all of these techniques, we have done enough testing. Right?

Well maybe.

## Modified Condition/Decision Coverage (MC/DC)

There is another, stronger level of coverage that we must discuss. This one is called modified condition/decision coverage (usually abbreviated to MC/DC or sometimes MCDC).

This level of coverage is considered stronger than those we have covered (so far) because we add other factors to what we were already testing in decision condition coverage. Like decision condition coverage, MC/DC requires that each atomic condition be tested both ways and that decision coverage must be satisfied. It then adds two more constraints:

1. For each atomic condition, there is at least one test case where the entire predicate evaluates to FALSE only because that specific atomic condition evaluated to FALSE.
2. For each atomic condition, there is at least one test case where the entire predicate evaluates to TRUE only because that specific atomic condition evaluated to TRUE.

There has been some discussion about the correct definition for MC/DC coverage. For example, the standard DO-178C[4] (discussed below) has a slightly looser definition, as follows:

---

[4] The ISTQB syllabus refers to standard DO-178B. However, this standard was replaced by DO-178C in January 2012. Therefore, because a few of the definitions are slightly different, we have moved to that standard.

Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect a decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

The definition we are using appears to be slightly more restrictive and so we will use it; this will ensure that when we are using it for safety- and mission-critical testing, it will cover all requirements for MC/DC coverage. By using our version of the definition rather than that in the syllabus or DO-178C, we supply coverage that's at least as good, and it simplifies the way we can create our tests, using the neutral-value design technique.

Note that, in order to use our definition, we need to deal with the unary negation operator. Assume we have the following predicate:

*A && (!B)*

According to rule 1, if *A* is TRUE, the predicate must evaluate to TRUE. Also, if *B* is TRUE, then the predicate must evaluate to TRUE. However, if *B* is TRUE, then *(!B)* evaluates to FALSE and the predicate evaluates to FALSE. The only way we can use our "neutral value" design method (as seen below) is to deem that the atomic condition is the variable and the unary operator together. That is, *(!B)* is deemed to be the atomic condition that, when it evaluates TRUE (i.e., *B* is FALSE), fulfills the MC/DC definition by forcing the predicate TRUE. This definition fits within the glossary definition.

For either definition, when the specified atomic condition is toggled from one state to its opposite, while keeping all other atomic conditions fixed, the entire predicate will also toggle to its opposite value. Interestingly enough, this tends to be a lot less testing than it sounds like. In fact, in most cases, this level of coverage can usually be achieved in (N + 1) test cases, where (N) is the total number of atomic conditions in the predicate.

Many references use the term *MC/DC* as a synonym for exhaustive testing. However, this level of testing does not rise to the level of exhaustive testing, which would include far more than just predicate testing (running every loop every possible number of times, for example). MC/DC does, however, exercise each complex predicate in a number of ways that should result in finding certain types of subtle defects.

MC/DC coverage may not be useful to all testers. It will likely be most useful to those who are testing mission- or safety-critical software. According to the most prevalent legend, it was originally created at Boeing for use when certain

programming languages were to be used in safety-critical software development. It is the required level of coverage under the standard FAA DO-178C[5] when the software is judged to fall into the Level A category (where there are likely to be catastrophic consequences in case of failure).

Avionics systems—which are covered by DO-178C—are a perfect example of why MC/DC coverage is so important. At one time, some software professionals believed that safety-critical software should be tested to a multiple condition level of coverage. That would require that every single possible combination of atomic conditions be tested—requiring essentially $2^n$ different test cases. See **Error! Reference source not found.** in section 2.5 to compare the advantage MC/DC coverage has over multiple condition coverage.

There are several different algorithms that have been advanced to define the specific test cases needed to achieve MC/DC-level coverage. Perhaps the simplest to use is the test design technique using neutral values.[6] This method works quite well with our definition of MC/DC coverage.

A neutral value for an atomic condition is defined as one that has no effect on the eventual outcome evaluation of the expression. The operator used in joining two atomic conditions determines exactly what value will be neutral. For example, suppose we are evaluating two atomic conditions, *A* and *B*. Keeping it simple, suppose we have two possible operators, AND and OR:

*A* AND *B*

*A* OR *B*

If *A* was the atomic condition we were interested in evaluating in our MC/DC test design, we would like the output result to be TRUE when *A* is TRUE and FALSE when *A* is FALSE. What value of *B* can we use that will enable that set of values? This is the value we call a neutral as it has no effect on the evaluation.

If the operator is an AND, then the neutral value will be TRUE. That is, if we set *B* to the neutral value, we control the output completely by toggling *A*.

*(A AND B)* → (TRUE AND ***TRUE*** == TRUE), (FALSE AND ***TRUE*** == FALSE)

On the other hand, if the operator is OR, the neutral value must evaluate to FALSE.

*(A OR B)* → (TRUE OR ***FALSE*** == TRUE), (FALSE OR ***FALSE*** == FALSE)

---

5 DO-178C: Software Considerations in Airborne Systems and Equipment Certification

6 This technique—and our definition—is adapted from *TMap Next – for Result-Driven Testing* by Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. (UTN Publishers, 2006).

Likewise, if we are interested in evaluating the value *B*, the value of *A* would be set to neutral in the same way. By using this concept of neutral substitutes, we can build a table for coming up with sufficient test values to achieve MC/DC coverage.

Let us consider a non-trivial predicate and then determine which test cases we would need to exercise to achieve full MC/DC coverage.

*(A OR B) AND (C AND D)*

The first step is to build a table with three columns and as many rows as there are atomic conditions plus one (the extra row is used as a header). That means, in this case, we start with three columns and five rows, as shown in Table 3.

| (A OR B) AND (C AND D) | T | F |
|---|---|---|
| **A** | T -- -- -- | F -- -- -- |
| **B** | -- T -- -- | -- F -- -- |
| **C** | -- -- T -- | -- -- F -- |
| **D** | -- -- -- T | -- -- -- F |

**Table 3: Non-trivial predicate (1)**

The second column will be used for when the predicate evaluates to TRUE and the third column will be used for when the predicate evaluates to FALSE. Each of the rows represents a specific atomic value as shown. Finally, each cell shown at the junction of an output and an atomic condition is set up so that the neutral values are set as dashes. In the second column (representing a TRUE output) and the first row (representing the atomic condition A), the contents represent a value:

TRUE Neutral Neutral Neutral

If the values substituted for the neutrals are indeed neutral, then cell (2,2) should contain values that will evaluate to TRUE, and cell (2,3) should contain values that evaluate to FALSE.

The next step (as seen in **Table 4**) is to substitute the correct values for the neutrals in each cell, basing them on the operators that are actually used in the predicate. This can be confusing because of the parentheses. Look carefully at the explanation given for each cell to achieve the possible combinations of inputs that can be used.

| (A OR B) AND (C AND D) | T | F |
|---|---|---|
| **A** | T F T T | F F T T |
| **B** | F T T T | F F T T |
| **C** | T F T T | T T F T |
| **D** | T F T T | T T T F |

**Table 4: Non-trivial predicate (2)**

- Cell (2,2): (**TRUE** OR B) AND (C AND D)
  - B is ORed to A, therefore its neutral value is FALSE.
  - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
  - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (2,3): (**FALSE** OR B) AND (C AND D)
  - B is ORed to A, therefore its neutral value is FALSE.
  - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
  - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (3,2): (A OR **TRUE**) AND (C AND D)
  - A is ORed to B, therefore its neutral value is FALSE.
  - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
  - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (3,3): (A OR **FALSE**) AND (C AND D)
  - A is ORed to B, therefore its neutral value is FALSE.
  - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
  - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (4,2): (A OR B) AND (**TRUE** AND D)
  - D is ANDed to C, therefore its neutral value is TRUE.
  - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
  - (A OR B) can achieve a value of TRUE in three different ways. It turns out not to matter which of those you use.
    - (F OR T)
    - (T OR F)
    - (T OR T)
- Cell (4,3): (A OR B) AND (**FALSE** AND D)
  - D is ANDed to C, therefore its neutral value is TRUE.
  - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.

 

    - (A OR B) can achieve a value of TRUE in three different ways. It turns out not to matter which of those you use.
        - (F OR T)
        - (T OR F)
        - (T OR T)
- Cell (5,2): (A OR B) AND (C AND **TRUE**)
    - C is ANDed to D, therefore its neutral value is TRUE.
    - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
    - (A OR B) can achieve a value of TRUE in three different ways. It turns out not to matter which of those you use.
        - (F OR T)
        - (T OR F)
        - (T OR T)
- Cell (5,3): (A OR B) AND (C AND **FALSE**)
    - C is ANDed to D, therefore its neutral value is TRUE.
    - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
    - (A OR B) can achieve a value of TRUE in three different ways. It turns out not to matter which of those you use.
        - (F OR T)
        - (T OR F)
        - (T OR T)

Okay, that is a mouthful. We have made it look really complex, but it is not. In truth, when you do this yourself, it will take a very short time (assuming you do not have to write it all out in text form for a book).

There is one final step you must take, and that is to remove redundant test cases (shown in **Table 5**). This is where it might get a little complex. When substituting values for (A OR B) in the final two rows, note that you have a choice whether to use (T OR T), (T OR F), or (F OR T). All three expressions evaluate to TRUE, which is the required neutral value.

Our first try was to use (T OR T) in all four cells where we could make that substitution. When we started to remove redundant test cases, we found that we could remove only two, leaving six test cases that we must run to get MC/DC coverage. Since the theory of MC/DC says we will often end up with (N + 1), or five test cases, we wondered if we could try alternate values, which might allow us to remove the sixth test case and get down to the theoretical number of five. By substituting the logical equivalent of (T OR F) in cells (4,2) and (5,2), we were able to remove three redundant test cases, as can be seen in Table 5.

| (A OR B) AND (C AND D) | T | F |
|---|---|---|
| **A** | T F T T | F F T T |
| **B** | F T T T | ~~F F T T~~ |
| **C** | ~~T F T T~~ | T T F T |
| **D** | ~~T F T T~~ | T T T F |

**Table 5: Test cases for non-trivial predicate**

This leaves us five (N + 1) tests, as follows:

Test 1: A = TRUE, B = FALSE, C = TRUE, D = TRUE

Test 2: A = FALSE, B = TRUE, C = TRUE, D = TRUE

Test 3: A = FALSE, B = FALSE, C = TRUE, D = TRUE

Test 4: A = TRUE, B = TRUE, C = FALSE, D = TRUE

Test 5: A = TRUE, B = TRUE, C = TRUE, D = FALSE

## Complicating Issues: Short-Circuiting

These two issues affect MC/DC. We are trying to test a complex predicate with multiple test cases. However, the actual number of test cases needed may be complicated by two different issues: short-circuiting and multiple occurrences of an individual atomic condition.

First, we will look at short-circuiting. Well you might ask, what does that term mean?

Some programming languages are defined in such a way that Boolean expressions may be resolved by the compiler without evaluating every subexpression, depending on the Boolean operator that is used. The idea is to reduce execution time by skipping some nonessential calculations.

| Value of A | Value of B | A\|\|B |
|---|---|---|
| **T** | T | T |
| **T** | F | T |
| **F** | T | T |
| **F** | F | F |

**Table 6: Truth table for A || B**

 

Consider what happens when a runtime system, that has been built using a short-circuiting compiler, resolves the Boolean expression (*A || B*) (see **Table 6**). If *A* by itself evaluates to TRUE, then it does not matter what the value of *B* is. At execution time, when the runtime system sees that *A* is TRUE, it does not even bother to evaluate *B*. This saves execution time.

C++ and Java are examples of languages whose compilers may exhibit this behavior. Some flavors of C compilers short-circuit expressions. Pascal does not provide for short-circuiting, but Delphi (object-oriented Pascal) has a compiler option to allow short-circuiting if the programmer wants it.

Note that both the OR and the AND short-circuit in different ways. Consider the expression *(A || B)*. When the Boolean operator is an OR (||), it will short-circuit when the first term is a TRUE because the second term does not matter. If the Boolean operator was an AND (&&), it would short circuit when the first term was FALSE because no value of the second term would prevent the expression from evaluating as FALSE.

Short-circuiting may cause a serious defect condition when it is used. If an atomic condition is supplied by the output of a called function and it is short-circuited out of evaluation, then any side effects that might have occurred through its execution are lost. Oops! For example, assume that the actual predicate, instead of

*(A || B)*

is

*(A || funcB())*

Further suppose that a side effect of ***funcB()*** is to initialize a data structure that is to be used later (not necessarily a good programming custom). In the code, the developer could easily assume that the data structure has already been initialized since they knew that the predicate containing the function had been used in a conditional that was expected to have already executed. When they to use the non-initialized data structure, it causes a failure at runtime. To quote Robert Heinlein, for testers, "there ain't no such thing as a free lunch."

On the other hand, short-circuiting may be used to help avoid serious defects. Consider a pointer p that may or may not be NULL. Short-circuiting allows this common code to work:

```
if ((p != NULL) && (*p>0))
```

If short-circuiting was not used and p actually evaluated to NULL, then this code would crash. Jamie has always been wary of such code, but it is widely used. Because the short-circuiting is going to be dependent on the compiler writer, Jamie would avoid this dependency by writing this code:

```
if (p) {
     if (*p > 0) {
     }
}
```

If the B expression is never evaluated, the question then becomes, Can we still achieve MC/DC coverage? The answer seems to be maybe. Our research shows that a number of organizations have weighed in on this subject in different ways; unfortunately, it is beyond the scope of this book to cover all of the different opinions. If your project is subject to regulatory statutes, and the development group is using a language or compiler that short-circuits, our advice is to research exactly how that regulatory body wants you to deal with that short-circuiting.

## Complicating Issues: Coupling

The second issue to be discussed is when there are multiple occurrences of a condition in an expression. Consider the following pseudo-code predicate:

A || (!A && B)

In this example, A and !A are said to be coupled. They cannot be varied independently as the MC/DC coverage rule says they must. So how does a test analyst actually need to test this predicate and still achieve the required coverage? As with short-circuiting, there are (at least) two approaches to this problem.

One approach is called unique cause MC/DC. In this approach, the term condition in the definition of MC/DC is deemed to mean uncoupled condition and the question becomes moot.

The other approach is called masking MC/DC, and it permits more than one atomic condition to vary at once but requires analyzing the logic of the predicate on a case-by-case basis to ensure that only one atomic condition of interest influences the decision. Once again, our suggestion is to follow whatever rules are imposed upon you by the regulatory body that can prevent the release of your system.

Wow! This is really down in the weeds. Should you test to the MC/DC level? Well, if your project needed to follow the FAA DO-178C standard, and the particular software was of Level A criticality, the easy answer is yes, you would need to test to this level. Level A criticality means that, if the software were to fail, catastrophic results would likely ensue. If you did not test to this level, you would not be able to sell your software or incorporate it into any module for the

project. We will discuss this standard and others that might affect your level coverage later in the chapter.

As always, context matters. The amount of testing should be commensurate with the amount of risk.