Editor: **Gerard J. Holzmann**
Nimble Research
gholzmann@acm.org

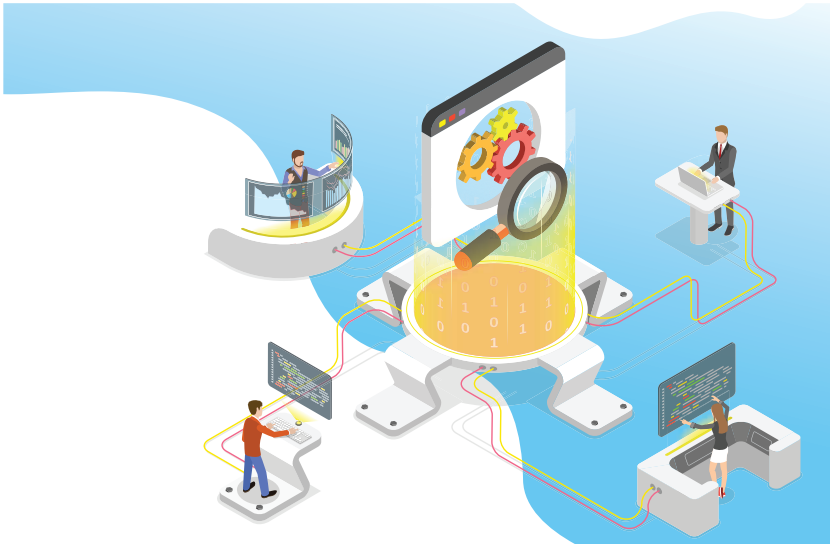
Test Fatigue

Gerard J. Holzmann

AS THE RECENT coronavirus outbreak has made painfully clear, the quantity and quality of our test efforts determine what defect rates we measure. If you don't test, or test poorly, you will discover few defects and may be tempted to draw the wrong conclusions about the quality of whatever it was that you were testing.

Suppose there are two teams developing competing products, let's call them *team Alice* and *team Bob*. If team Alice performs a more rigorous testing of their product than does team Bob, their number of discovered defects will likely be much higher than team Bob's. One may then be tempted to conclude that the quality of team Alice's product is lower than that of team Bob. This could of course be true, but it would be incorrect to conclude that from these numbers.

Clearly, the more rigorous testing you do, the more problems you will find. So, what exactly is a sufficiently rigorous way to test software, especially if that software is safety critical? There are some standard guidelines that most organizations follow, so we may want to look at how good those guidelines are. Can we really trust software products that were tested to the best available standards?



Statement Coverage

The minimum one could require of a test strategy is that it exercises every executable statement in the code. That seems like a relatively mild requirement, but it is not. Consider, for instance, a switch statement where the cases cover all possible values of an enumerated value. Most standards require that every switch statement also contains a default clause to make sure that one does not unintentionally skip some cases. In the example, that default clause will be unreachable. In defensive coding, one also tries to protect against the unthinkable types of errors, just in case some bizarre malfunction or data corruption leads an execution astray in unforeseen ways.

Reaching full statement coverage becomes difficult if we have to make the impossible happen in all these cases. Most organizations therefore do not require 100% statement coverage in product testing, but aim for getting as close to that number as possible. This is, in itself, a little unsettling because it makes the target level

Digital Object Identifier 10.1109/MS.2020.2986107
Date of current version: 18 June 2020

negotiable, which means that it can give way under time pressure. Simple statement coverage also falls short in that it ignores the sensitivity of computations to data values. For that, we have to look at ways to exercise execution paths, and not just statements.

It's All About the Data

For safety critical systems, the currently prevailing standards, such as ISO 26262,[1] EN 50128:2011,[2] and IEC 61508,[3] strongly recommend the use of a test coverage metric known as *modified condition/decision coverage* (*MC/DC*). One standard, DO-178C,[4] which is dominant in the airspace industry, goes a step further and defines its use as *required*.

To meet MC/DC requirements, every condition in a program not only has to evaluate to true and false in separate tests, but every clause in the condition must also independently evaluate to true and false in separate tests. This introduces some of the required sensitivity to data values that influence computations. Of course, it is not too hard to cheat on this metric and ease the test requirements by moving the evaluation of Boolean expressions outside decision points, that is, placing them in statements that are evaluated before each "if" statement that has multiple clauses in the condition.

Cyclomatics

A different metric, introduced in the late 1970s by Thomas McCabe,[5] aims to measure the number of paths through the control-flow graph of a function. To achieve sufficient path coverage of a function, we should now run at least as many tests as the cyclomatic complexity number indicates. This metric is typically defined by the formula E–N + 1, where N is the number of nodes, and E is the number of edges of the control-flow graph.

Not surprisingly, it is also easy to cheat on this metric and dramatically lower the measured cyclomatic complexity counts without changing the functionality of a function. Consider, for instance, the switch statement shown in Figure 1(a).

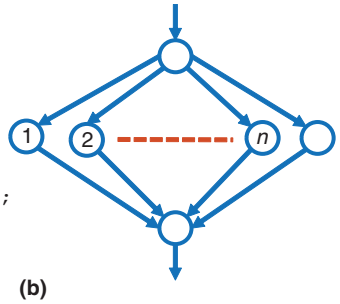
The cyclomatic complexity contributed by the switch statement is $n + 1$. Yet, we can write this same fragment of code with a data-driven approach, using a lookup table with function pointers, as shown in Figure 1(c). Now the fragment of code has the minimum cyclomatic complexity of *one*. If n is 10, using cyclomatic complexity metric, we would need to run at least *11* tests of the fragment in Figure 1(a), but only *one* for the fragment in Figure 1(a), even though they perform the same computation.

There are proposals, based on examples like this, to modify the complexity metric by not counting switch statements at all. It is not hard to imagine that this will provide an incentive to some developers to rewrite all if-then-else statements as switch statements as well, and artificially lower the cyclomatic complexity numbers, and thus the perceived test burden.

Other proposals have tried to move the definition of cyclomatic complexity numbers closer to an MC/DC metric by increasing the number by one for every Boolean operator that is used in conditional tests. The extended metric produces higher numbers, although it loses some of the intuition behind the definition as a pure graph property. Are any of these metrics sufficient to achieve adequate test coverage?

```
switch (i) {
case 1: fct_1(); break;
case 2: fct_2(); break;
…
case n: fct_n(); break;
default: assert(false); break;
}
```

(a)



(b)

```
void (* table[])(void) = { fct_1, …, fct_n};
assert(i > 0 && i <= n);
table[i]();
```

(c)

FIGURE 1. (a) A switch statement with $n + 1$ cases and (b) the corresponding control-flow graph with $n + 3$ nodes and $2(n + 1) + 2$ edges, giving a cyclomatic complexity of $n + 1$, equal to the number of execution paths. Code that performs the same function with a lookup table is shown in (c), giving a cyclomatic complexity of just one (assuming that the assertion is implemented as a function call).

Some Gotchas

Something that is easily lost in the debate about useful test metrics is that the true measure of test quality is not coverage, but how well it helps us determine if all design requirements are

met. If we have a test suite that accomplishes this fully, yet leaves portions of the code uncovered, this either means that there is redundant code in the application or that the design requirements are incomplete. Both issues would need addressing before we would just blindly add test cases that accomplish nothing but to reach uncovered parts of the code. But there are other problems as well.

Figure 2 shows a small fragment of code written in C, with two conditional tests in a row and the corresponding control-flow graph. The first condition allows for global pointer variable p to be assigned the address of integer parameter named *x*, and the second condition allows that same pointer variable to be dereferenced and assigned the value of a second integer parameter named *y*. We'll leave aside here the wisdom of using pointers to function parameters or manipulating their values in this way, but just focus on the structure of the control-flow graph.

Two tests will suffice to get 100% statement coverage in this case, and because the conditions are very simple, they also suffice to achieve 100% MC/DC coverage. The first test can be to call fct(0,0), and the second test fct(1,1). These two tests exercise two of the four possible paths through the graph. The remaining two paths can be reached by calling fct(1,0) and fct(0,1). The first three of these tests reveal no problems with the way this function is written. The last test though, fct(0,1), will lead to a crash. So in this case, the MC/DC-compliant test suite covered just 50% of the paths in the control-flow graph and fails to reveal a serious bug.

What if we did not have just two conditional tests in a row but 10 or 100? If we otherwise don't change the structure of the graph, just two separate tests could still produce 100% MC/DC compliance, but the number of paths would be 1,024 (2^{10}) in the first case and a staggering 1.27×10^{30} (2^{100}) in the second case. This means that the odds of finding a bug, if it exists in just one of those paths, is 0.1% in the first case, and just about zero in the second case. That doesn't sound too good, so let's look at a different example.

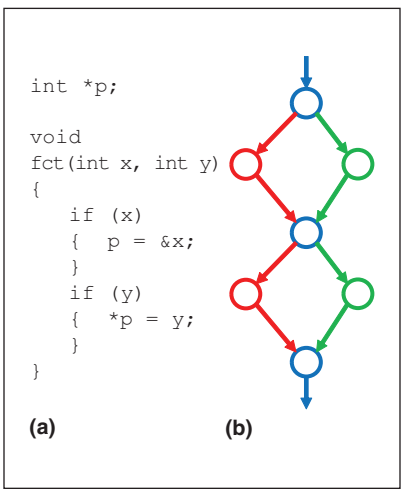


FIGURE 2. (a) A code fragment with two consecutive conditional tests and (b) its corresponding control-flow graph. With seven nodes and 10 edges, the cyclomatic complexity is four.

Figure 3 shows another small fragment of code, this time containing a simple for-loop, which is used to assign values to the elements of a locally declared array, and the corresponding control-flow graph. How many paths are there that we can take through this graph? It depends, of course, on the value of x + y. All negative values of the sum will short circuit the loop, but all positive values (in say, 64-bit integer arithmetic) will produce a different number of iterations of the loop, and thus a different execution path. We can achieve 100% MC/DC coverage with just a single test though, for instance, by executing just fct(1,1). Needless to say, this one test will fail to reveal the potential out-of-bounds array-indexing error that is lurking in this code.

Both of these first two examples are still reasonably simple, but things go further downhill fast if we also introduce a small amount of concurrency into the code. I've often used the following example in tutorials and courses I've given, because it shows the nature of the problem so well.

Consider three threads of execution, with three statements in each. That's a total of only nine statements,

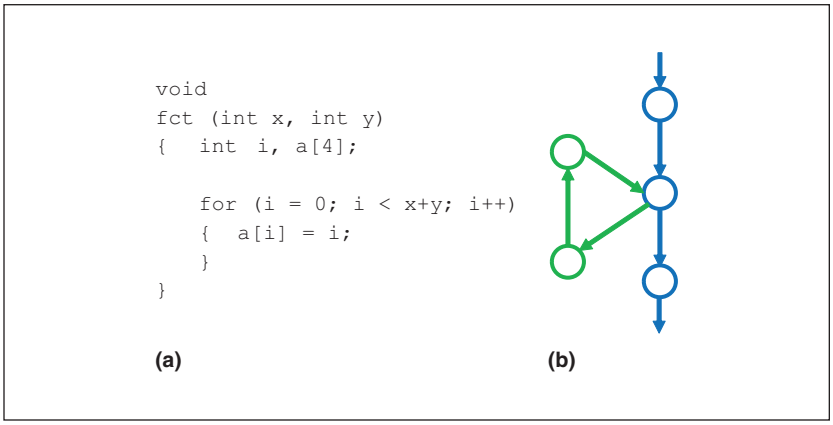


FIGURE 3. (a) A code fragment with a loop and (b) its corresponding control-flow graph. With five nodes and seven edges, the cyclomatic complexity is three.

without any conditional tests or loops—just straight-line code, as displayed in Figure 4.

How many possible execution paths are there for this system? If we assume arbitrary interleaving, using a process scheduler that is fully unconstrained, you can visualize the executions with a Rubik's cube, where every path from one corner of the cube to the opposite corner, traveling along the edges of the 27 smaller cube segments, is a possible execution path. Say the steps of thread_1 move in the *x*-axis alongside the cube edges, the steps of thread_2 move along the *y*-axis, and the steps of thread_3 along the *z*-axis.

If you do the calculation, you'll see that there are 1,680 such paths through the cube. But, any single one such path will produce 100% MC/DC coverage. If only one of these interleaving paths would lead to a crash (there are more), that would mean that the odds of uncovering it with that single test would be 1 in 1,680, or less than 0.06%. Note that this is still a very small system. The code that runs the Mars Curiosity Rover built at NASA's Jet Propulsion Laboratory, for example, has roughly 2.8 million lines of code executing in approximately 120 different parallel tasks. Although there are constraints in this case on task interleaving, the number of possible executions is astonishingly large. So, is there something else we can do that is not more burdensome than MC/DC compliance testing already is, that can perform better?

Fuzzing

A popular alternative method is fuzz testing. The basic approach is simple: randomize the inputs to the software under test. It is likely to shake out bugs, especially where input values fall outside the range that a developer expects. Generally, the best approach is to bias the input selections to likely vulnerable spots, for instance, near boundary values, but we can still look at how well a purely random set of tests performs.

For this experiment, I took a randomly generated graph using a program created by mathematician Richard Johnsonbaugh. I generated a graph with 1,000 nodes and 2,000 edges and an average fanout for each node of seven successors. A total of 781 of the nodes are reachable from a preselected start node. How many of those reachable nodes can we find with a series of random walks? For the experiment, we limited the maximal length of a test run to a fixed 5,000 steps to avoid getting bogged down in cycles.

We can take the graph to represent not just a control-flow graph but a full program execution graph, with explicit data values, so that each path through this graph is representative of a true execution. The same state in a control-flow graph could then appear in many places in the execution graph, when it is reached for different data values. Given that, the 1,000 node graph is only a tiny example of what we can expect to see for a full program execution of a real software application.

Figure 5 shows the number of visited nodes if we perform between 10 and 100,000 random test runs in this graph. The solid line shows the percentage of the visited nodes that are unique, that is, after we remove duplicates when the same nodes are visited repeatedly in different tests.

The effective coverage that is realized increases quickly for the first few tests, but then flattens and more slowly reaches an asymptote, making it harder and harder to increase coverage further. Even after 100,000 runs of up to 5,000 steps each, we never reach all 781 nodes. The time needed for these tests grows, of course, linearly with the number of tests performed, which,

```
int    x,  y,  r;
int   *p, *q, *z;
int  **a;

thread_1() // initialize
{
     p = &x;
     q = &y;
     z = &r;
}
```

(a)

```
thread_2() // swap *p and *q
{
     r = *p;
    *p = *q;
    *q = r;
}
```

(b)

```
thread_3() // access z via a and p
{
      a = &p;
     *a = z;
    **a = 12;
}
```

(c)

FIGURE 4. (a)–(c) The three parallel threads of execution, with three statements in each, without conditionals or loops. Because there are no conditional paths, the cyclomatic complexity of each function is one.

even for this small graph, can quickly become excessively large (we stopped the tests after 26 h of runtime). The main reason for the inefficiency of this test suite is the amount of duplicate work that is done, with tests performing the same executions over and over. So, is there no hope?

Graph Algorithms

What if we used a plain depth-first search algorithm from the same start node in the graph and see how long it takes to visit each node? As you might expect, it takes just one single "test" to do this, and this run visits all 781 unique nodes, for 100% coverage, in a fraction of a second.

How can we make use of the large difference between the performance of a depth-first search algorithm compared with the relatively low coverage that can be obtained with randomized tests or test suites that are compliant with an MC/DC metric?

The key here is that the depth-first search algorithm can remember nodes that have been visited before and can backtrack efficiently to a previous point in the search to explore alternatives for moving forward. To enable backtracking, we should be able to either save complete search states on a stack or to recreate a previous state by undoing the last action performed. That is simple for small graphs but can be expensive for execution graphs of a realistic size, where a single-state description could require the storage of hundreds of kilobytes or more. There are tools such as logic model checkers that can optimize this process, but they are not always easy to use.

The Mars Rover

The following numbers can show what is possible though. I was involved in the testing of the flash file system software for the Curiosity Rover that landed on the surface of Mars in August 2012, and that today still continues its exploration of the Martian surface. The flash file system code is roughly 6,000 lines of code, so it's not particularly large, but it is quite complex. Standard testing of the code was a required part of the development process, with the usual goal of getting as close as possible to 100% statement coverage. There was no requirement to also maximize MC/DC coverage or to consider the cyclomatic complexity of functions at this time.

Jointly, the approximately 100 unit tests that were defined for this code reached 35,796 unique system states. Across six modules from this code, the median value of the statement coverage that was realized in these tests was a respectable 98%, meeting the formal test requirements.

We also instrumented the same code to do a randomized search. After approximately 5 h, the randomized search had reached 398 Million system states, exploring roughly 50,000 execution paths. That's already quite a bit better than the standard test suite. We then repeated the test by instrumenting the code for a depth-first search using the Spin[6] model checker as the search engine. After running this test for another 5 h, the search had reached 745 million distinct system states, while exploring approximately 50 million distinct execution paths. The numbers are summarized in Table 1. So, in this application, the more rigorous tests explored four orders-of-magnitude-more states and execution paths than with standard test methods, bringing a comparable increase in rigor and in the number of problems discovered in these tests.

Given the effort that is required to set up this type of model-driven test, this level of rigor is typically only feasible for a subset of the truly critical modules within a larger application. The full Mars Rover software counted roughly 2.8 million lines of code, of

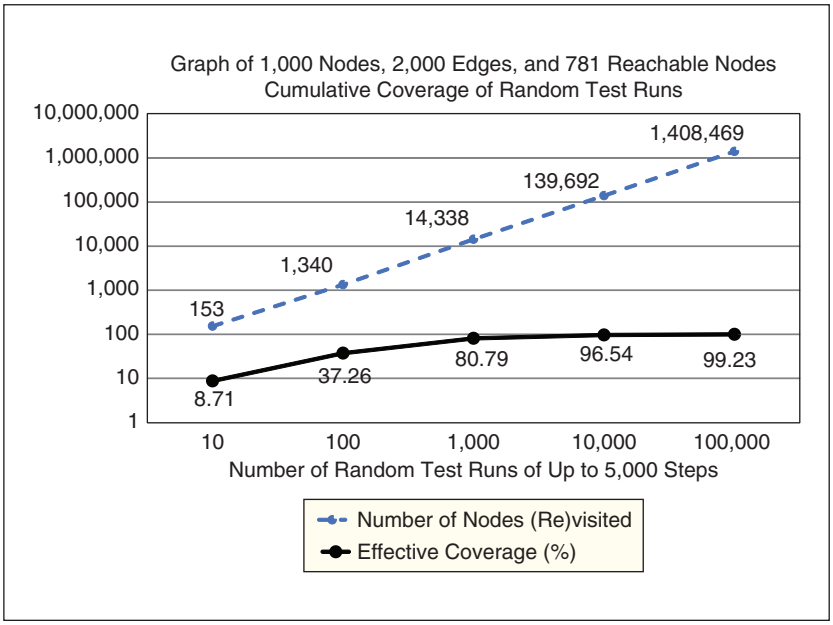


FIGURE 5. The cumulative coverage of random test runs in a random graph of 1,000 nodes and 2,000 edges, with 781 nodes reachable.

Table 1. A comparison of coverage for three different test methods of the Mars Curiosity Rover flash file system software.

	Number of unique states reached	Number of execution paths
Standard unit test with 98% statement coverage	35,796	100
Randomized fuzz test	398 million	50,000
Depth-first search instrumented test	745 million	50 million

ABOUT THE AUTHOR

GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.

which the 6,000 lines for the flash file system was only a small part.

Static Testing

Test rigor is, of course, not an all-or-nothing issue. What if you do not have the resources to explore the type of rigorous code exercise that I described previously? Fortunately, there are still some very good choices, and for the Mars Rover software, we used them all.

The most direct method for increasing the level of rigor of a software test effort is currently to use tools that work by performing symbolic executions of the code while testing for potential anomalies. A single symbolic execution uses ranges of possible data values, capturing the possible effect of large numbers of concrete executions, although with less precision. One can even use this type of framework to reason backwards and answer questions like "for which input data values can a given statement execution result in an error, such as a nil-pointer deference, an out-of-bounds array-indexing error, or the evaluation of an uninitialized variable?"

The tools that can do this type of analysis are static source code analyzers, which have quickly gained in popularity. If you can afford it, it is recommended to use more than just one state-of-the-art static source code analyzer. For the Mars Rover flight software, for example, we used five different source code analyzers. There is surprisingly little overlap in the output of the tools: most tools currently on the market are developed with a particular strength and theoretical foundation, and they excel at the corresponding type of analyses. The combined results of all the tools were an integral part of the code reviews that we used on the Rover software.[7]

The best part of static code analysis is perhaps that the checks that are performed are automated and can be run repeatedly, from the moment coding starts, on every new module check-in and on every integration build. This addresses another common feature of standard testing: test fatigue. After all, "exhaustive" testing often means that testing continues until either the test team, or the time available to them, is exhausted.

References

1. *Road Vehicles—Functional Safety—Part 2: Management of Functional Safety*, ISO Standard 26262-2, 2018.
2. *Railway Applications—Communication, Signalling and Processing Systems—Software for Railway Control and Protection Systems*, EN Standard 50128, 2011.
3. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, IEC Standard 61508, 2010.
4. *Software Considerations in Airborne Systems and Equipment Certification*, RTCA Standard DO-178C, 2012.
5. T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976. doi:10.1109/TSE.1976.233837. [Online]. Available: https://ieeexplore.ieee.org/document/1702388
6. *Spin*. [Online]. Available: http://spinroot.com.
7. G. J. Holzmann, "Mars code," *Commun. ACM*, vol. 57, no. 2, pp. 64–73, 2014. doi: 10.1145/2560217.2560218. [Online]. Available: https://dl.acm.org/doi/10.1145/2560217.2560218