Using Code Coverage to Improve the Reliability of Embedded Software

Whitepaper | V3.0 2020-11
# Table of Contents

1. Introduction ................................................................................................................................. 3
2. Levels of Code Coverage .................................................................................................................. 3
   2.1 Statement Coverage .................................................................................................................... 3
   2.2 Statement Coverage Limitations .................................................................................................... 3
   2.3 Branch Coverage .......................................................................................................................... 3
   2.4 Branch Coverage Limitations ....................................................................................................... 3
3. Understanding MC/DC ....................................................................................................................... 4
4. Coverage from Different Types of Testing ....................................................................................... 4
5. Challenges of Capturing Coverage in an Embedded Environment ................................................... 5
6. Capturing Coverage Data in a Shared Environment ......................................................................... 5
7. Conclusion ....................................................................................................................................... 5
1 Introduction

Code coverage is a metric used to gauge the completeness of software testing by identifying which areas of source code in an application were exercised during a test. It provides a convenient way to ensure that applications are not released with untested code.

Can code coverage be used to improve reliability? The answer is yes. Performing code coverage analysis is the simplest way to ensure quality in a software application.

2 Levels of Code Coverage

Code coverage can be measured in a variety of ways. The most common approach is to measure one or a combination of one or more of the following: statement coverage, branch coverage, and Modified Condition/Decision Coverage (MC/DC). The following sections will describe each of these coverage types in detail.

2.1 Statement Coverage

Statement Coverage is the measure of which executable lines of code have been executed. It does not take into consideration loops or conditional statements, just the statements within an executable line. It should be noted that a “statement” is not the same as a line of code. In C, C++, Java or Ada, typically, a semicolon terminates a statement. In some scenarios, a single statement can span multiple lines of code. While Statement Coverage provides a good measure of what has and hasn't been executed in the program, it does have some limitations.

2.2 Statement Coverage Limitations

Consider the following code fragment in Figure 1.

```
int* p = NULL;
if (condition)
p = &variable;
*p = 123;
```

Figure 1: Statement coverage defect code example

By 'condition' being true, it is possible to achieve 100% statement coverage. However, this test case misses the scenario where 'condition' is false. This program would de-reference a null pointer in that case. While Statement Coverage is a good metric, it is the “entry level” of code coverage. Ideally the ‘condition’ false case is also tested.

2.3 Branch Coverage

Branch Coverage measures whether decision and branch points are tested completely for all possible outcomes. For example, an 'if' statement must take on both “true” and “false” outcomes to be considered covered. If only one of the paths is taken, this is classified as 'partial' coverage.

However, like Statement Coverage, there are some subtleties we need to be conscious of, especially when working with languages that perform “lazy evaluation.” Lazy evaluation is the technique of delaying computation of a piece of code until it is needed.

2.4 Branch Coverage Limitations

A typical scenario where “lazy evaluation” can occur is with complex Boolean expressions like the code in Figure 2.

```
int* p = NULL;
if (condition1 && (condition2 || function1(*p)))
  statement1;
else
```

Figure 2: Branch coverage defect code example

Consider the scenario where 'condition1' is false. Lazy evaluation would not need to evaluate 'condition2' or 'function1(*p)'. This would also result in covering the ‘false’ coverage path for 'if (condition1 && (condition2 || function1(*p)))'.

Now consider the scenario that 'condition1' and 'condition2' are both ‘true’. Again, lazy evaluation would result in 'function1(*p)' not being evaluated. This would also result in the 'true' coverage path being taken for the conditional above. In this case, it is possible to have 100% Branch coverage and still have potential defects in the software.
3 Understanding MC/DC

MC/DC is sort of a “super branch coverage.” It reports on the true and false outcomes of a complex conditional as is done in branch coverage, but it also reports on the true and false outcomes of the sub-condition in a complex conditional.

MC/DC was created at Boeing and is typically required for aviation software for DO-178B Level A certification. It addresses the issue raised by lazy evaluation, by requiring a demonstration that every sub-condition can affect the outcome of the decision independent of the other sub-condition values.

Looking at the example in Fig. 2, we would need to verify ‘condition1’ for its “true” and “false” while holding ‘condition2’ and ‘function1(*p)’ fixed, then we would need to do the same, for ‘condition2’ while holding ‘condition1’ and ‘function1(*p)’ fixed.

Finally, we would do the same for ‘function1(*p)’, while holding ‘condition1’ and ‘condition2’ fixed. The verification of each sub-condition for its ‘true’ and ‘false’ values while holding the other sub-conditions fixed is known as an ‘MC/DC pair’. An MC/DC truth table is typically used to identify pairs. An example of this can be seen in Table 1.

<table>
<thead>
<tr>
<th>Row</th>
<th>Ca</th>
<th>Cb</th>
<th>Cc</th>
<th>Rslt</th>
<th>Pa</th>
<th>Pb</th>
<th>Pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>*1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Modified Condition / Decision Coverage Truth Table

Actual Expression is: ( condition1 && ( condition2 || function1(*p) ) )
Condition “a” (Ca) is: condition1
Condition “b” (Cb) is: condition2
Condition “c” (Cc) is: function1(*p)
Simplified Expression is: ( a && ( b || c ) )

Pa => no pair was satisfied
1/5 2/6 3/7
Pb => no pair was satisfied
2/4
Pc => no pair was satisfied
3/4

4 Coverage from Different Types of Testing

Software testing comes in many flavors. For simplicity, we will use three terms in this paper:

- System / Functional Testing: Testing the fully integrated application
- Integration Testing: Testing integrated sub-systems
- Unit Testing: Testing a few individual files or classes

Every project does some amount of system testing where the source code is stimulated with some of the same actions that the end users will do. One of the most common causes of applications being fielded with bugs is that unexpected, and therefore untested, combinations of inputs are encountered by the application when in the field.

Not as many projects do integration testing and even fewer do unit testing. If you have done integration or unit testing, you are probably painfully aware of the amount of test code that has to be generated to isolate a single file or group of files from the rest of the application.

At the most stringent levels of unit and integration test, it is not uncommon for the amount of test code written to be larger than the amount of application code being tested. As a result, these levels of testing are generally applied to mission and safety critical applications in regulated markets such as: aviation, medical devices, railway, process control, and soon automotive. Many of the applications written for these industries contain embedded software.
The structural testing process for regulated industries often revolves around testing the high and low-level requirements and analyzing the code coverage that results from this “requirements-based” testing. On many projects, high-level or functional requirements are tested first. Code Coverage can be used to capture and report on the amount of coverage achieved.

Unfortunately, it is almost impossible to get 100% code coverage during system/functional testing. More commonly, you will achieve 60%-70% code coverage during this type of testing. The remaining 30-40% code coverage is achieved using unit and integration testing techniques.

Unit testing involves using test code in the form of drivers and stubs to isolate particular functions in the application and stimulating those functions with test cases. These “low-level” requirement-based tests provide much greater control over the code being tested. They are used to augment the previously executed system tests and allow you to get to 100% coverage. For this reason, it is very convenient to be able to share coverage data from different types of testing.

5 Challenges of Capturing Coverage in an Embedded Environment

As the old saying goes, “You can’t get something for nothing.” There’s always a price to be paid. In the case of code coverage, the price to be paid is the addition of instrumentation to the source files to be tested. Instrumentation is the additional source code added to an application to allow the collection of coverage data as tests are executed.

The overhead associated with instrumentation translates directly into increased source file and program size, and indirectly into increased execution time. Being able to forecast the impact of instrumenting source code for coverage can be of critical importance. This is especially true when testing in a real-time embedded environment.

It is not possible to forecast the precise impact that instrumentation would have on a particular set of application files. No algorithm exists for this purpose, and none is possible. Too many variables are involved, and every application is unique in its complexities. It is possible, however, to derive a set of estimates from a representative example.

6 Capturing Coverage Data in a Shared Environment

One of the primary issues with managing code coverage in an embedded target environment is the absence of reserve memory to accommodate the additional instrumentation code.

Estimates done at Vector on different samples of code indicate that the source code can grow up to an additional 10% (for a state-of-the-art coverage tool) for the various types of coverage mentioned. For most 32-bit targets, this is not an issue. For 8 and 16-bit targets with limited memory, this most certainly can be an issue.

Different code instrumentation techniques are employed to reduce the size of the executable. Several different data capture techniques are also used depending on the memory available. In-memory caching systems are used to monitor which sections of code have been reached. This is essential technology in keeping the amount of RAM used in the instrumented executable to a minimum.

7 Conclusion

Do not mistake Code Coverage for “Tested” or “Bug Free.” To have 100% code coverage does not mean the application is reliable. It only means that the tests that were executed stimulated all of the code for the desired level of coverage.

Code coverage is not a final activity, but rather a continual activity toward a reliable system. The earlier it is employed the better. It provides scope to add tests, improve tests, remove redundant tests, and cover more code, thereby ensuring higher quality and more reliable systems. While there are some hurdles to overcome using code coverage tools in an embedded environment, recent tool advances make the use of such tools practical in almost all instances.
Get More Information

Visit our website for:
> News
> Products
> Demo software
> Support
> Training classes
> Addresses

www.vector.com