The Path Not Taken: Maximizing the ROI of Increased Decision Coverage

Laura Bright

Laura_bright@mcafee.com

Abstract

Measuring code coverage is a popular way to ensure that software is being adequately tested and gives software teams confidence in the quality of their product. In particular, decision coverage of both automated and manual tests is a good way to ensure that as many paths as possible are covered in testing and the team is not missing possible defects. Management likes to report high decision coverage numbers as an indicator of product quality, and many QA teams set a goal of reaching a target percentage as a way of demonstrating that the code has been thoroughly tested.

But what does high decision coverage mean in terms of overall product quality? Clearly it is good to cover as many decisions as reasonably possible. A good set of functional test cases will cover the "low-hanging fruit", but may leave many uncovered decisions. Some of these uncovered decisions may be difficult or impossible to cover under normal testing conditions, but many can be covered by adding automated functional tests, unit tests, and manual tests. Hard to reach "corner cases" can be covered by setting up special testing environments. Some of these new tests may cover important use cases while others may provide little benefit. The question this paper aims to answer is, how do you determine where to invest your efforts to increase decision coverage?

This paper presents strategies to increase your decision coverage by targeting areas of the code with the highest ROI. Rather than focus on reaching a target percentage, we show how you can set realistic decision coverage goals that will improve your overall software quality. We present specific examples of how this approach has benefitted our team, including:

- Identifying automated functional test cases that were overlooked by QA
- Removing dead or obsolete code
- Finding product defects that cause some code to not get executed
- Identifying manual test cases to cover parts of code that are difficult to test through automation
- Determining areas of the code that would benefit from increased unit testing

Through these efforts, our team has increased our percentage decision coverage, but more importantly, has improved overall quality by finding defects that otherwise may have been missed.

Biography

Laura Bright is a Software Development Engineer in Test (SDET) at McAfee, with a focus on automated testing. She has spent the past five years developing automation tools and automated test suites for McAfee products. More recently, she has served as a Quality Champion at McAfee and works across teams to share best practices to improve software quality. Previously she has presented "Catch Your Own Bugs: Including all Engineers in the Automation Cycle" at PNSQC 2012.

1 Introduction

Measuring code coverage is a popular way to ensure that software is being adequately tested and gives software teams confidence in the quality of their product. Management likes to report high decision coverage numbers as an indicator of product quality, and many QA teams set a goal of reaching a target percentage as a way of demonstrating that the code has been thoroughly tested.

But what does high coverage mean in terms of overall product quality? Clearly it is good to cover as many code paths as reasonably possible, but how do you know when it is beneficial to add new test cases? A good set of functional test cases will cover the "low-hanging fruit", but may leave many uncovered decisions. Some of these uncovered decisions may be difficult or impossible to cover under normal testing conditions, but many can be covered by adding automated functional tests, unit tests, and manual tests. Hard to reach "corner cases" can be covered by setting up special testing environments. Some of these new tests may cover important use cases while others may provide little benefit.
If your coverage numbers are above your target, does that indicate high quality? If your numbers are below the target, is that a cause for concern? In some cases, the answer to both of these questions is “yes”, but focusing on just one number may oversimplify the problem and cause teams to miss the big picture.

This paper aims to reframe the question of when and how to increase code coverage in terms of product quality. Instead of asking “What should we do to improve our code coverage numbers?”, the question we ask is “What areas of the code need increased coverage to improve the quality of our software?” The goal is not just to increase coverage but to cover overlooked test cases and other reasonable scenarios. We present a case study showing our team’s experience analyzing our code coverage data to identify which areas of the code needed additional testing to meet this goal. We present examples of uncovered areas of the code that required additional tests, as well as areas where we chose not to write tests because these tests were unlikely to improve product quality. The goal is to use code coverage tools and data to improve your testing strategy, rather than focus on tasks with a low return on investment (ROI) just for the sake of increasing your numbers.

Section 2 presents a brief overview of types of code coverage and terminology used in this paper, and discusses related work. Section 3 provides an overview of the team and project structure and describes how we got started on our detailed code coverage analysis and targeted condition/decision coverage improvements. Section 4 presents specific examples encountered while reviewing the coverage data and discusses how to identify areas of the code that should be targeted for improved coverage. Section 5 presents results, and Section 6 concludes.

2 Background and Terminology

There are many different types of code coverage including line coverage, function coverage, branch or decision coverage, and condition coverage. See reference [Bullseye] for more information.

- **Line coverage** – percentage of lines of code executed
- **Function coverage** – percentage of functions executed.
- **Decision coverage** – percentage of all branches executed in conditional statements (for example, for a single “if” statement, have tests been executed where the statement evaluates to both true and false)
- **Condition coverage** – percentage of conditions that have evaluated to both true and false. This is different from decision coverage because a single “if” statement may consist of multiple conditions.
- **Condition/decision coverage** – This is the union of the condition coverage and decision coverage. It has the advantage of simplicity without the individual limitations of each of these two metrics [Bullseye].

To collect our code coverage data we used the Bullseye code coverage tool [Bullseye] which reports two of these metrics: function coverage and condition/decision coverage. The condition/decision coverage metric in Bullseye reports the percentage of all possible decisions and conditions that have been executed. Note that Bullseye does not report the condition coverage or decision coverage separately so we do not provide the breakdown of these two metrics in our results. We focus on condition/decision coverage rather than function coverage since the condition/decision coverage is a better measure of how well we are testing all possible paths in the code. Condition/decision coverage is also more difficult to increase than function coverage, since there may be many possible branches and conditions that can be executed in a single function. In the remainder of the paper we use the term coverage to refer to the Bullseye condition/decision coverage metric unless otherwise noted.

In the work presented in this paper, we focus on automated functional test coverage since we found these tests to be the most effective way to cover the uncovered code for our project. However, we do include unit test and manual test coverage examples where appropriate, as these types of tests are also valuable for increasing code coverage and improving product quality. We note that these results are partly due to the nature of our project. The project in this paper is complex anti-malware software that integrates many components. While it is possible to write unit tests for individual components, we found that in many cases adding functional tests had a higher ROI. Unit testing is still an important part of our project, and we recognize that it can be a very useful way to increase code coverage of many software projects.

2.1 Previous work

The topic of code coverage has been studied extensively in the literature. [Marick91] provides a good study of achieving different code coverage goals through unit testing, but argues that it is better to achieve as much coverage as possible through black box tests and fill in the gaps with unit tests. The author also notes that some branches are infeasible and discusses the concept of weak mutation coverage, branches that are unlikely to reveal defects. While the specific language, tools, and coverage metrics differ from the examples presented in this paper, the author’s arguments are consistent with the main takeaways of the work we present. [Marick99] notes that code coverage metrics should not be a minimum shipping requirement, but instead should be used as a guide to indicate areas where testing can be improved. The author points out that imposing a minimum requirement may encourage testers to add tests that cover the easiest conditions until they reach the goal, thereby potentially missing more important test cases. Imposing a minimum requirement may also encourage testers to add many low ROI test cases [Marick99]. [Manu2010] presents a case study of a team’s efforts to increase their block coverage from 91% to 100%, starting with “interesting” blocks such as those that cover key customer scenarios.

3 Getting started

This section provides a brief background on our team and project followed by an overview of our efforts to increase code coverage.

3.1 Project overview

We present examples and data gathered while testing the McAfee Endpoint Security suite which consists of multiple components designed to protect systems for small and medium businesses. The project follows a scrum model and consists of multiple scrum teams, each working on a distinct component of the Endpoint Security suite. At the end of each sprint, all teams report condition/decision coverage for both automated functional tests and unit tests as part of a larger set of quality metrics. These metrics are reviewed regularly by all stakeholders to help evaluate overall project quality.

The specific project we focus on in this paper is the Threat Prevention module which includes Anti-Malware protection and memory protection. This module includes both on-access and on-demand scanners to scan for viruses as well as buffer overflow protection and access protection. The code is developed in C++. In most cases, automated functional tests are written by QA engineers and unit tests are written by developers, although there has been increasing overlap in the developer/QA roles.

During feature development the size of the underlying code base and the percentage of conditions and decisions covered fluctuates, but most teams typically reported condition/decision coverage numbers in the range of 30-40%. This range was considered acceptable but not ideal, and a goal for all teams was to raise this number to 50% or greater over the course of the project.

As part of an effort to understand our team's numbers and identify areas for improvement, we held team meetings to review uncovered conditions and decisions in the code and identify which conditions and decisions needed to be covered through additional tests. Since reviewing thousands of lines of code is a huge task, we focused on one component per meeting and limited meeting time to one hour. We started with the largest and most complex components since these were among the most commonly used and were likely to have a higher ROI.

All of the developers and QA engineers on the team were expected to attend, and everyone brought valuable expertise to the meeting. Developers provided insight into when uncovered decisions or functions were expected to be covered to help QA engineers write appropriate test cases. In some cases they determined that the uncovered function or decision was obsolete and could be safely removed from the code. QA engineers determined what additional test cases were required to cover these areas. In some cases QA determined that test cases had been inadvertently overlooked or omitted from the automation suite and needed to be added to weekly automation runs.

To keep the meeting within the one-hour time limit, whenever we found an uncovered function or code block that required additional testing or investigation, the meeting moderator would make a note of the code segment and action required and assign follow-up tasks after the meeting. Sample follow up tasks included:

1. Determine if function X is still called anywhere in the code
2. Write test cases and automation scripts to cover condition Y
3. Remove a block of obsolete code

In the next section we provide some more in-depth examples of the types of uncovered decisions found in these meetings. We present both examples of code branches that we added additional tests to cover, as well as branches where we determined that covering them would do little to improve overall product quality.

4 Examples

In this section we present pseudocode examples that are representative of the types of uncovered code we found in our review sessions. In general, the uncovered conditions and decisions that have the highest ROI to add coverage for are the ones that are expected to get covered in practice, are straightforward to test either manually or through an automated functional or unit test, and that could potentially expose a previously undetected defect. Note that this does not mean we expect to find defects if we cover the condition, but rather that if there is a defect it would be undetected if we do not test the condition. In the code segments we use the following notation for the decisions, similar to what is provided by the Bullseye tool.

- ➔ Decision or function is not covered at all
- ➔ T – path only taken when decision evaluates to true
- ➔ F – path only taken when decision evaluates to false
- ➔ t – the individual condition within the decision statement has evaluated to true but not to false
- ➔ f – the individual condition in the decision statement has evaluated to false but not true
- TF – Both true and false decisions have been evaluated
- tf – the individual condition has evaluated to both true and false

4.1 Uncovered functions

The first example we found in our review were uncovered functions, i.e., functions that were never executed during the test run. Since high function coverage is generally easier to achieve than high condition/decision coverage, these examples offer some “low-hanging fruit”, and determining how to handle these functions is a good first step towards improving code coverage.

We found uncovered functions that fell into several categories:

1. **Additional test cases needed to cover the function**. Some of the uncovered functions were types of functions developers could provide guidance for what product functionality needed to be tested to cover the function. In these cases it is usually straightforward to add manual or automated tests for product features that were not commonly used and may have been overlooked by QA.

2. **Wrapper functions**. Some libraries contained multiple APIs for the same functionality, for example wrapper functions that set default values for optional parameters. Generally covering these uncovered APIs was considered lower priority as long as the underlying functionality was being adequately tested, although if the interface was confirmed to be obsolete it was removed.

3. **Dead code**. Some uncovered functions were obsolete APIs or functionality that had been removed from the product. Once developers confirmed that these functions were not ever expected to be executed, they were removed from the code base, which helped increase overall code coverage numbers and more importantly makes the code easier to understand and maintain.

4. **Functions that were expected to be covered**. In some cases, a function was not covered even though we believed it should have been covered by an existing test case. Such cases could either indicate a product defect, or at the very least further investigation to understand why they were not covered and what needed to be changed to cover them.

4.2 Dead code

In addition to unused functions, our review sessions identified code that was unreachable in practice. While much of this code was technically reachable through unit tests, it would never be covered by an end user. For example, some parts of the code included features that were not scoped for the current release, either because they were obsolete features dropped from the product or they were “placeholders” for features that were planned for a future release. In both cases, there was little benefit to testing this code for the current release since it covered features that were not supported. We estimate that at least 1-2% of the code in some components was unreachable. See Section 5 for details.

Obsolete code blocks can be safely deleted from the code, while code that is intended for a future release can be commented out. As we show in Section 5, removing and/or commenting out this code significantly improved our team’s decision coverage metrics and ensured that our coverage numbers were meaningful when measuring overall quality. Further, it cleaned up the code and allowed the team to focus on testing supported code that would be covered by an end user.

4.3 Boundary Tests

Many of the uncovered decisions we encountered were boundary cases that were verifying that variable values were within the acceptable range. Consider the following example in Figure 1:

```
 ➔ T  if
 ➔ t (x >= MIN_VALUE &&
 ➔ t                x <= MAX_VALUE) {
          //do something
   }
 ➔ else {
          // handle the invalid value
   }
```

Figure 1: Boundary Test Example

In Figure 1 the two conditions x >= MIN_VALUE and x <= MAX_VALUE have both evaluated to true but not false, so we have not tested handling invalid values. Such test cases are usually straightforward to automate through black box tests if the values are configurable by the end user, or through unit tests otherwise. Adding these test cases increases decision coverage and verifies that invalid values or errors are properly handled.

4.4 Manual tests

Our review also identified parts of the code that should be covered manually, and gave us the opportunity to evaluate the coverage of our manual test suite. For example, our product needs to detect viruses and malware on removable drives, and provides the option to pause tasks on laptops while running in battery mode. Such test cases cannot be easily automated. Since these test cases are not part of the automation suite, they are not executed as often and may be more prone to defects. Reviewing our coverage data allowed us to easily identify features and scenarios that are best covered by manual tests and ensure that the manual test suite is adequately covering the product.

4.5 Error handling

In our experience reviewing uncovered conditions/decisions, there were many error and null pointer checks. Some of these checks may be valid error conditions that could occur in practice, but others are checking for errors that will only occur if there is a bug in the code. In this section we discuss the difference between these two cases. In the former case, adding a test case to cover an uncovered error check may be beneficial, but in the latter case the code will only be covered if a bug exists and there is less benefit to explicitly trying to cover it.

It is good coding style to verify that a pointer is not null before dereferencing, and to verify that a function call succeeded before continuing. All of these checks are important to include in the code, especially during development and testing phases where they may help catch critical defects. However, once the code has been debugged and everything is working as expected, most of these conditions should not get covered in practice. How do you to determine when it is beneficial to write a test case to cover an error condition? Consider the following two examples in Figure 2 and Figure 3:

```
void importantFunction() {

         int errorCode = DoSomethingImportant();

    ➔ T if (errorCode == SUCCESS) {
            // <continue execution here>
         }
    ➔    else {
            // this should not happen
            Log_Debug("DoSomethingImportant() failed");
         }
         return;
}
```

Figure 2: Handling an error that is not expected to occur in practice

It is considered good coding practice to verify error codes returned by function calls, and many static analysis tools will indicate a potential defect if the return value of a function is not checked.

In the example in Figure 2, the function DoSomethingImportant is expected to return success, but the caller of the function validates the returned error code as a formality and to make the code clean and readable. If the call to DoSomethingImportant fails it indicates a bug in that function that needs to be fixed, but there is less benefit to explicitly writing a unit test to cover the uncovered condition. (We note that there is benefit to verifying that the debug log is working correctly, but this particular example covers a failure that is not expected to occur in practice). Adding a specific test case to cover this decision would improve the coverage metric but would be unlikely to help find new defects. Note that all of the decisions within the DoSomethingImportant function should still be covered adequately to ensure that this function is working as expected.

```
void anotherFunction() {

         int errorCode = DoSomethingElse();

    ➔ T if (errorCode == SUCCESS) {
           // <continue execution here>
         }
    ➔    else {
           Log_Event("a serious error occurred");
           ErrorRecoveryFunction();
         }
}
```

Figure 3: Handling an error that may occur in practice

In the second example in Figure 3, there is explicit error handling code in ErrorRecoveryFunction() to handle a rare catastrophic failure that could potentially occur in practice. The error check is not just a formality for good coding style, but is actually testing a valid error condition. In this case, there would be high value in adding a test case to make sure ErrorRecoveryFunction() is working.

4.6 Null Pointer Checks

Another class of error handling decisions where there is often little benefit to covering both True and False is null pointer checks.

It is good coding practice to verify that a pointer is not null before dereferencing it, and doing so improves the overall quality of the code and makes the product more robust. However, as in the case of error handling, there is often little value in explicitly adding a test case to cover both the True and False evaluations of these conditions unless they are failures that might be expected occur in practice. Consider the following examples:

```
          int * integer_array;
          integer_array = new int[MAX_ARRAY_SIZE];

    ➔  T if (integer_array) {
              // <continue execution here>
          }
```

```
          Object * myTestObject = new Object();

    ➔  T if (myTestObject) {
                // <continue execution here>
          }
```

Figure 4: Verifying memory allocation

In the examples in Figure 4, the memory allocation is expected to succeed in almost all cases unless the system is out of memory. While testing a system with insufficient memory is an important test scenario, explicitly covering every null pointer check in the code is less likely to have a high ROI. A related scenario is a “clean-up” function that deletes an object or array if it exists (Figure 5):

```
void cleanUp (Object * obj) {

    ➔  T if (obj) {
              delete obj;
          }
}
```

Figure 5: Verifying an object exists before deletion

As in the previous examples in Figure 4, the null pointer check in Figure 5 is included to make the code more robust and correctly handle cases where cleanup is called before initialization of the object. However, the case where the object is null may not be expected to occur in practice and explicitly testing this condition is a low priority.

Many functions return a reference to an object, and similar rules apply in cases where this reference is not normally null. If a function returns a reference to an array or object and the value is never expected to be null unless there is a catastrophic system error, there is little value in adding a test case to cover the null pointer condition.

We note that the majority of the uncovered conditions we encountered during our review meetings were either return code checks evaluating to error, or pointer checks evaluating to null. This is one reason we determined that trying to achieve near 100% decision coverage would have low ROI for our project.

5 Results and Takeaways

This section presents some data from the detailed coverage review meetings. As mentioned in Section 3, we reviewed one component at a time to make the workload manageable, and also to allow us to focus on the most critical parts of the product first. This table presents some key metrics from these review meetings:

	Percent condition/ decision coverage before review	Percent condition/ decision coverage after review	Test cases added	Unreachable conditions/ decisions removed	Defects filed	Additional conditions/ decisions covered
Component 1	46	53	25	121	2	114
Component 2	48	50	20	~20	2	~25

Table 1: Results

For component 1, we found approximately 25 new functional test cases to automate and add to our nightly automation suite. Most of these test cases were boundary conditions or uncovered functions for less commonly used product features. We also noted a handful of manual test cases that needed to be executed. During this review we also identified large blocks of code for features that were no longer supported in the product as well as for a feature that was planned for a future release but is not supported in the current release. These blocks included 121 uncovered conditions/decisions. This code was commented out to allow us to focus on higher-priority decisions that were still uncovered.
After removing the uncovered decisions and adding the 25 new test cases to our test suite, coverage for Component 1 increased from 46% to 53%. Most of this increase was due to the dead code removal, but the 25 functional test cases contributed to the increase and also uncovered at least 2 product defects. While 53% is a lower number than many organization aim to achieve, our team determined that most of the remaining uncovered decisions were error checks and null pointer checks that had a low ROI. After this review our team had greater confidence that Component 1 had been adequately tested, and we had increased the coverage above 50% which was considered a realistic and achievable goal for our project.

We later conducted a similar review for Component 2. Component 2 had fewer obsolete/placeholder blocks, but we still identified several functions that were no longer needed. We also identified 20 valid test cases that covered important product functionality and found 2 new defects. We note that during the period this review took place, the code for this component underwent a refactor to reduce unnecessary complexity, which makes it difficult to report the exact numbers for removal of unreachable code and additional conditions/decisions covered.

After the refactoring was completed, our overall condition/decision coverage of the component increased from 50% to 56%. While the refactoring was done independently of the team's code coverage efforts, an important benefit of this change was eliminating unnecessary conditions and decisions from the code base and helping the team more easily identify the highest priority uncovered conditions/decisions going forward. We plan to continue these efforts across all components of the project.

6 Conclusions

Code coverage is an important quality metric that has received considerable attention in the software community. While every software project aims to achieve high code coverage, often less attention is given to how high coverage translates to high quality and how software teams can best use their available resources to improve coverage.

The results presented in this paper present some key findings for our team and product, but your mileage may vary. For example, your code base may have fewer error checks and null pointer checks than the code presented in these examples, so your team may realistically expect to achieve coverage much greater than 50%. Similarly, you may find that most of the null pointer checks or error checks in your code are valid error conditions that need to be covered. The examples presented in this paper should not be used as hard rules, rather they should serve as examples of how you can interpret your own code coverage data to identify the test cases that have the highest ROI for your project. While this paper focused on examples using C++ code and the Bullseye code coverage tool, similar principles should apply to other programming languages and tools.

When reviewing uncovered conditions/decisions in your code coverage data, you should consider the following questions:

- Is this uncovered condition a valid test case?
- Is it straightforward to cover this condition by a manual test, or by an automated functional or unit test?
- Is there a good chance this condition will be covered in practice by an end user?
- Is it a high risk if we don't test this condition?

If you answer yes to all these questions, there is a high ROI to add a test to cover the condition. If your test suite has covered all the conditions and decisions that are likely to occur in practice and the team agrees there is a low ROI to covering the remaining uncovered paths, you can state with greater confidence that you have met your code coverage goals.

References

[Bullseye] Bullseye Testing Technology "Code Coverage Analysis." http://www.bullseye.com/coverage.html (accessed July 9, 2014).

[Marick91] Marick, Brian. "Experience with the Cost of Different Coverage Goals for Testing" *Pacific Northwest Software Quality Conference, 1991. Available at http://www.exampler.com/testing-com/writings/experience.pdf*

[Marick99] Marick, Brian. "How to Misuse Code Coverage" *Testing Computer Software, 1999. Available at http://www.exampler.com/testing-com/writings/coverage.pdf*

[Manu2010] Manu, C., Amalo, D., Nagpal, P., Tan, R. "The Last 9% Uncovered Blocks of Code – A Case Study" *Pacific Northwest Software Quality Conference, 2010.*