An introduction to code coverage

Potential affects on software bugs
and tools used to increase coverage

Stina Långström
Louise Zetterlund

April 2020

1 Introduction

Whether you are working with machines or computers, one way to improve a product has always been to find flaws and get rid of them. It could be that a part of a lawn-mover easily gets broken or that a system crashes if you try to enter a letter in a digit-only field. Firstly, you want to find these issues before your customers do and either complain or take advantage of the flaws. Secondly, you want to fix these issues, or customers will not continue to buy or use your product.

In computer systems, one way to find these bugs is to write test suites to your program. These test suites could be symbolizing different customer paths through your system, just to see what different choices the customer gets and what inputs they could try. You might want to test as many different inputs as possible, just to make sure there are no bugs in your system.

One way to get a measurement of how much of the code you have tested is to check your so-called code coverage. Code coverage is a measurement that shows in percentage how much of a code-project is executed when a specific test suite is run [1]. Having a high code coverage suggests having a lower risk of containing undetected bugs.

With fewer bugs and less broken code, developers can spend more of their time implementing new features instead of debugging, which improves a team's productivity. Using code coverage as a daily routine can have great benefits for a team, but how great? And is code coverage reliable?

How the code coverage measurements affect code has been discussed a lot among scientists and developers, and there are several different answers. This is what this article will be about, what code coverage is, which metric that is the most accurate, and how to use coverage as a developer.

2 Background

Code coverage metrics are used to evaluate a test suite's quality, meaning its ability to detect faults. The suite's quality is often evaluated by its ability to kill mutants according to Gopinath et al [2], which are artificially seeded potential faults. Suites that kill many mutants are more likely to find real faults according to several studies [3]. While mutation testing is good at evaluating a suit and find faults, it is prohibitively expensive computationally. By this fact, one wants to use coverage to evaluate the test suit instead.

To make sure that any computer program is reliable, scientists have for decades tried to invent methods for systematic software testing. As early as 1963, Joan C. Miller and Clifford J. Maloney discussed the coverage of code [4]. By that time they talked about a logic tree to find out what combinations of input data the program needed to be equipped to handle. Today logic trees are not used for this, but the thought is the same. One wants to make sure that programs can handle all types of inputs.

There exist two types of code coverage today, Data Flow code coverage and Control Flow code coverage. The latter is, in general, more commonly used, mainly because Data Flow code coverages are more expensive to run, which outperforms the fact that it is expected to provide better results than Control Flow [5].

Data Flow code coverage examines how values are associated with variables and how these associations can affect the execution of the program [5]. Each variable is classified either as a definition occurrence (def), where the value is bound to the variable or a use occurrence (use) where the value of the variable is referred. The criteria used to determine the coverage is the def-use pair coverage. This criterion examines whether the use of a variable has been tested properly, meaning that the values are computed and used correctly in the code.

Control Flow Code coverage is used to examine the execution flow of test cases in the code [5]. There exists several metrics to evaluate this [6][2], for example:

- Function coverage - measures how many of the functions that have been called.
- Statement coverage - measures the number of statements in the program that have been executed.
- Branch coverage - measures how many control structures that have been executed, meaning how many branches that were taken.
- Condition coverage - measures how many of the boolean sub-expressions that have been tested for a true and false value

- Path coverage - measures the paths explored in a program's control flow graph
- Line coverage - measures how many lines of the source code that have been tested

Since Data Flow code coverage is more expensive to run than Control Flow coverage, Control Flow is more used. By this fact, the focus of this article will be on Control Flow code coverage.

3 Which metric is the best?

Code coverage is often used by developers as a metric to evaluate the test suit of their project, but since there exist many metrics to use it can be rather challenging to deduce which metric best explains the particular suit. When researching what science says the result ends up in no straightforward answer.

Gligoric et al. [3] experiments showed that branch coverage does the best prediction of a test suit because of its high effectiveness and low overhead, but they also concluded that acyclic intra-procedural path coverage is highly competitive to it. Frankl and Weiss [7] showed however that def-use coverage is more effective than branch coverage for fault detection when they compared the two.

Gopinath et al. [2] concluded that statement coverage is the best predictor for mutation kills when comparing statement, block, branch, and path coverage. Gupta et al., on the other hand, compared block coverage, branch coverage, and condition coverage [8], and concluded that branch coverage was better in killing mutants than block coverage. They also discovered that condition coverage was better than branch coverage when the methods had composite conditional statements.

Wei et al. examined the correlation of branch coverage and fault detection [9]. They concluded that the correlation of the two in their tests was very weak. They used randomly generated test suits and observed that when the branch coverage increased so did the fault detection, but when branch coverage begins to saturate the fault detection continued to increase. By that, they concluded that branch coverage was not that reliable.

4 Tools to help with code coverage

Two of the different types of tools to help with code coverage are tools that measure the code coverage and tools that help with creating tests to increase the code coverage in the program [10]. The tools are often focusing on one or a few programming languages. This report will look at some different programming languages and some different test coverage tools, both those that generate their own test suites and those that measure the code coverage.

4.1 Java

There are quite many tools when wanting to increase code coverage in Java, both measurement tools and generating tools.

One of the early code coverage tools is EMMA [11]. This is an open-source project created by Vlad Roubtsov [10] with features for measuring and reporting code coverage for Java projects. EMMA is using flags in the code to see if it has been executed or not [10]. The tool was created in 2001 [11] and the last stable release took place in 2005, so it is not currently under active development [10] and therefore does not support newer versions of Java. Several tools have been developed as replacements for EMMA, one of them being JaCoCo (short for Java Code Coverage) [10]. The tool was created in 2009 with the same features as EMMA and is still maintained, so it supports active Java versions. JaCoCo also supports Maven plug-ins which makes it possible to create reports in Maven builds [12].

Another tool for Java is Parasoft Jtest, a program providing tools for analyzing code, identifying coverage gaps, and suggesting JUnit recommendations to increase the code coverage [13].

One of the tools that generate JUnit tests to increase code coverage is EvoSuite [14], but it does not measure the coverage. On their webpage, they are suggesting some other tools to execute their generated tests with, to get the measurements. One of their recommendations is JaCoCo.

4.2 Python

There are not as many code coverage tools for Python as there are for Java, which is logical since Java is more frequently used than Python [15]. One of the most popular code coverage tools is Coverage.py. It is an analyzing tool that measures the code coverage in a program. The way it works is that Coverage.py records each file and line number that is executed in the tests by using a trace function [16]. The same function is used when analyzing branch coverage, where each line is being paired up with the previous, so it is possible to see branches [16].

4.3 C/C++

As mentioned above, some tools support more than one language, and Parasoft is one of those tools. Parasoft has tools for not only Java but also C, C++, and .NET. The tool for C and C++ is combined and, similar as the Java-version, offers features for code coverage and runtime analysis. However, it does not seem to provide the same test generating feature as the Java-version, only modules to "simplify unit test creation" [17].

4.4 Fuzzing

Fuzzing is primarily a software testing technique but can be used to improve the code coverage of a program. Fuzzing is not a language-specific tool, but more of an approach to generate semi-random data into a program, in other words - generate test cases [18]. Many fuzzers need initial test cases to fuzz a program, which means that fuzzing is more of a complement to already existing test suites. The main benefit of a fuzzer is that it is very exhaustive, and is thereby more likely to find other inputs than a human, hence enhancing the code coverage. There exist several tools for fuzzing, two commonly used are American Fuzzy Loop (AFL) [19] and Radamsa [20]. Both of them require initial tests to be successful and generated new inputs based on them.

5 Discussion

In the aspect of the research of code coverage, the scientists who have written articles on the subject do not seem to agree on the best approach. Some say that branch coverage is best at detecting faults, others argue that branch coverage and fault detection are not that correlated and some claim that statement coverage is better than branch coverage. To make a conclusion out of this, one can say that it depends on the program. Code coverage information does help developers extend their tests and by that find faults, but it all depends on how the tests are written. If the test is only written to enhance the coverage, they are likely not good tests and will probably not detect any bugs. However, if the tests are written to detect bugs, then code coverage can be a great tool for helping a developer do so. With that being said, high code coverage does not generally induce high quality of the test suite.

To get as high code coverage as possible all code should be tested, but is that necessary? A frequently discussed question on the internet is whether to test trivial code or not. Trivial code could, for example, be getters and setters, a very simple code where you will discover issues quite fast.

Carlos Schults writes at TechBeacon [21] that he does not think one should test trivial code. Schults compares the opinions of Robert C. "Uncle Bob" Martin (American software engineer and author/instructor) and Mark Seemann (Danish programmer and author), where Martin does not think one should test trivial code. Martin himself does not test, for example, getters, setters, member variables, GUIs, and code written by third parties [21]. Seemann, on the other hand, thinks otherwise. He agrees with not testing GUI code, but he argues that even if getters and setters are trivial from start, it does not mean that they will stay trivial if the code changes [22].

In the case of code coverage, it would mean that not testing trivial code would decrease the coverage percentage, and by testing trivial code the result would

be increased code coverage without actually increasing the quality of the tests. Code coverage does not count the complexity of the code when doing any analysis, which means that 10 % of getters and setters are still 10 %, even if they are trivial. Should a tester strive for the highest test coverage, no matter how trivial the code covered is?

Noticeable in this article is that there exist a lot of tools to help with code coverage, both in measurement but also in the creation of tests. Many of the tools are language-specific, and by that one needs to use different tools depending on the programming language. By this fact, it can be harder to get started with coverage tools. It could also mean that such a tool is more adaptive to the code than a tool that is not language-specific, and by that, it can perhaps create more complex tests.

6 Conclusion

Code coverage is a great way of finding untested parts of the source code of a program, but that does not mean that a high code coverage promises no bugs in the code. It is very much up to the developers themselves how and if they want to use code coverage in their programs. The most important part is that they are testing their program, not what percentage they have, or if they are covering every branch of a function.

It is possible to get a higher code coverage with the help from some tools, and it is easier if one is using a specific programming language since the tools are often language-specific. Fuzzing can on the other hand be used on all programs, but can also be relatively shallow and is often dependent on initial tests. Use of unit tests, like Parasoft and EvoSuite, makes it easy to find out if code return unwanted answers, while the use of fuzzing might find out that there exists a bug that crashed the program. What tool that is best for a project depends on a lot of what the developer wants to achieve.

Code coverage is a very useful tool, but important to note is that a high code coverage does not mean a reliable program.

References

[1] Wikipedia contributors. *Code coverage — Wikipedia, The Free Encyclopedia*. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=939135811` (visited on 04/27/2020).

[2] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code Coverage for Suite Evaluation by Developers". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 72–82. ISBN: 9781450327565.

DOI: 10.1145/2568225.2568278. URL: https://doi.org/10.1145/2568225.2568278.

[3] Milos Gligoric et al. "Comparing Non-Adequate Test Suites Using Coverage Criteria". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, pp. 302–313. ISBN: 9781450321594. DOI: 10.1145/2483760.2483769. URL: https://doi.org/10.1145/2483760.2483769.

[4] Joan C. Miller and Clifford J. Maloney. "Systematic Mistake Analysis of Digital Computer Programs". In: *Commun. ACM* 6.2 (Feb. 1963), pp. 58–63. ISSN: 0001-0782. DOI: 10.1145/366246.366248. URL: https://doi-org.focus.lib.kth.se/10.1145/366246.366248.

[5] H. Hemmati. "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 151–156.

[6] Sten Pittet. *An introduction to code coverage*. 2020. URL: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage (visited on 04/19/2020).

[7] P. G. Frankl and S. N. Weiss. "An experimental comparison of the effectiveness of branch testing and data flow testing". In: *IEEE Transactions on Software Engineering* 19.8 (1993), pp. 774–787.

[8] A Gupta and P Jalote. "An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing". In: *Int J Softw Tools Technol Transf 10*. 2008, pp. 145–160. DOI: 10.1007/s10009-007-0059-5. URL: https://doi-org/10.1007/s10009-007-0059-5.

[9] Y Wei, B Meyer, and M Oriol. "Is Branch Coverage a Good Measure of Testing Effectiveness?" In: *Lecture Notes in Computer Science, vol 7007*. Berlin, Heidelberg: Springer, 2012, pp. 194–212.

[10] Wikipedia contributors. *Java code coverage tools — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/wiki/Java_code_coverage_tools# (visited on 04/25/2020).

[11] Vlad Roubtsov. *EMMA*. 2006. URL: http://emma.sourceforge.net/index.html (visited on 04/25/2020).

[12] Marc R. Hoffmann. *JaCoCo Java Code Coverage Library*. 2020. URL: https://www.jacoco.org/jacoco/ (visited on 04/25/2020).

[13] Parasoft. *Parasoft Jtest*. 2020. URL: https://www.parasoft.com/products/jtest (visited on 04/25/2020).

[14] EvoSuite. *EvoSuite - Automatic Test Suite Generation for Java*. 2018. URL: http://www.evosuite.org (visited on 04/25/2020).

[15] W3Techs. *Comparison of the usage statistics of Java vs. Python for websites*. 2020. URL: https://w3techs.com/technologies/comparison/pl-java,pl-python (visited on 04/25/2020).

[16] Ned Batchelder. *Coverage.py documentation*. 2020. URL: https://coverage.readthedocs.io/en/coverage-5.1/index.html (visited on 04/25/2020).

[17] Parasoft. *Parasoft Ctest*. 2020. URL: https://www.parasoft.com/products/ctest (visited on 04/25/2020).

[18] OWASP Open Web Application Security Project. *Fuzzing*. 2020. URL: https://owasp.org/www-community/Fuzzing (visited on 04/26/2020).

[19] Google Michal Zalewski. 2020. URL: https://github.com/google/AFL (visited on 04/26/2020).

[20] Oulu University Secure Programming Group. 2019. URL: https://gitlab.com/akihe/radamsa (visited on 04/26/2020).

[21] Carlos Schults. *Should you test trivial code?* 2018. URL: https://techbeacon.com/app-dev-testing/should-you-test-trivial-code (visited on 04/25/2020).

[22] Mark Seemann. *Test trivial code*. 2013. URL: https://blog.ploeh.dk/2013/03/08/test-trivial-code/ (visited on 04/25/2020).