Predicting Test Suite Effectiveness Using Static Analysis
- Work in Progress -

Paco van Beckhoven[1,2], Ana Oprescu[1], and Magiel Bruntink[2]

[1]University of Amsterdam
[2]Software Improvement Group

Abstract

Software testing is an important part of the software engineering process, widely used in industry. Software testing is partly covered by test suites, comprising unit tests written by developers. As projects grow, the size of the test suites grows along. Monitoring the quality of these test suites is important as they often influence the cost of maintenance. Part of this monitoring process is to measure the effectiveness of test suites in detecting faults. Unfortunately, this is computationally expensive and requires the ability to run the tests, which often have dependencies on other systems or require non-transferable licenses. To mitigate these issues, we investigate whether metrics obtained from static analysis could predict test suite effectiveness, as measured with mutation testing. Preliminary results show that, when size is ignored, there is a correlation between statically estimated code coverage and effectiveness. However, when suites of equal sizes are compared the correlation drops significantly. Our current focus is investigating the reasons of this behaviour.

1 Introduction

Software testing is an important part of the software engineering process. It is widely used in the industry for quality assurance [3] as tests can tackle software bugs early in the development process and also serve for regression purposes. Part of the software testing process is covered by developers writing automated tests such as integration or unit tests. Monitoring the quality of the test code has been shown to provide valuable insight when maintaining high quality assurance standards [2]. Previous research shows that as the size of production code grows, the size of test code grows along [13]. Quality control on test suites is therefore important as the maintenance on tests can be difficult and generate risks if done incorrectly [5]. Typically, such risks are related to the growing size and complexity which consequently lead to incomprehensible tests. An important risk is the occurrence of *test bugs i.e.*, tests that fail although the program is correct (*false positive*) or even worse, tests that do not fail when the program is not working as desired (*false negative*). Especially the latter is a problem when making changes to existing code and breaking changes are not detected by the test suite. This issue can be addressed by measuring the fault detecting capability of a test suite *i.e.*, test effectiveness, often performed using mutation testing. Mutation testing tools generate mutants *i.e.*, faulty versions of the program, and then run the tests to determine if the mutant was detected. However, mutation testing techniques have several drawbacks, such as limited availability across languages and being resource expensive [14].

To tackle these issues, we investigate whether static analysis can be used to predict test effectiveness scores as measured with mutation testing. We formulate the following research question:

Research question 1 : What static analysis based metrics would accurately predict test suite effectiveness?

Our work builds on previous research, *i.e.*, a Test Quality Model (TQM) [2] based on static analysis

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org). 07-09 June 2017, Madrid, Spain.

originally introduced to relay test quality to speed of issue-handling. First, we consider approaches to adapt TQM to predict test suite effectiveness. Next, we consider approaches to improve the quality of the predictions obtained via static code analysis.

Based on TQM we structure our analysis on the following sub-questions:

Research question 1.1 : To what extent is TQM a good model for test suite effectiveness?

Research question 1.2 : How can the TQM be improved to better predict test suite effectiveness?

We obtained an initial assessment of these approaches by evaluating them through a prototype implementation on a limited, baseline set of experiments using a single Java project.

Our findings indicate that the number of assertions is related to test effectiveness when test suite size is ignored. However, when size is accounted for, the correlation is gone.

2 Background and Related Work

Athanasiou *et al.* introduced a TQM [2] based on three characteristics of test code: Completeness, Effectiveness and Maintainability. Each characteristic is scored using a number of static metrics. The model is used to analyse the relation between test quality and the speed of handling issues, as high quality test suites should ease code maintenance [2]. Their findings suggest that test code quality, as measured by their model, is positively related to some aspects of the speed at which issues are handled.

Nagappan introduced the Software Testing and Reliability Early Warning (STREW) static metric suite to provide "an estimate of post-release field quality early in software development phases" [8]. These metrics could be used to extend the set contained by TQM.

Recent research evaluated test suites with a focus on coverage-based metrics [5, 11]. Although code coverage is often used as the main test adequacy criterion, the correlation between high coverage and defect detection is not always present [11]. Their method for systematic assessment and enhancement of test suites, Test Suite Assessment and Improvement Method (TAIME), extends simple code coverage with specialization and partition metrics. These metrics could be used to extend the TQM when coverage is measured statically.

Static code analysis and test quality research [12, 10] could be input for our test effectiveness prediction method.

2.1 Measuring Test Effectiveness

An important aspect of test quality is the effectiveness of a test suite, often measured as the number of faulty versions of a System Under Test (SUT) that can be detected by the test suite. However, as real faults are not known in advance, mutation testing is applied as a proxy measurement. Just *et al.* showed statistically significant evidence that mutant detection is correlated with real fault detection [7].

A strong correlation between coverage and test suite effectiveness was found. However, it is argued that this correlation was also due to test size. (Test size is defined by the number of tests in a suite.) The ongoing debate on the strength of this relation was fed by Inozemtseva *et al.*'s find that if test size is accounted for, the correlation between coverage and effectiveness is generally weak (< 0.7) [6]. Accounting for size means that only test suites of equal sizes are compared. Namin *et al.* showed that both coverage and size influence effectiveness independently[9].

3 Problem Analysis

Dynamic analysis of large projects, such as dynamic code coverage or mutation testing, is typically expensive.

Moreover, mutation analysis has several disadvantages: it is not available for all languages, it is resource expensive, it often requires compilation of source code, it requires running the tests which often depend on other systems. Mutation testing cannot be applied when an external analysis of the code is performed as tests often depend on an environment. All these issues are compounded when performing software evolution analysis on large-scale legacy or open source projects.

4 Research Method

First, we evaluate the metrics of the TQM to investigate whether its metrics are relevant to test effectiveness. Next, we run the mutation and static analysis and compare them. We then investigate metrics from other research.

We implement a tool as a vehicle to answer the main research question. It reads source files of a specific project and calculates the TQM metrics scores using static source code analysis. Based on these scores, the tool predicts the effectiveness of the test suite. The tool could also help study the evolution of test code quality.

4.1 RQ 1: What static analysis based metrics would accurately predict test suite effectiveness?

Research on test effectiveness has shown that size and coverage influence the effectiveness [6, 9] others found a negative relation between assertion and fault density [4]. We limit our initial set of metrics to those related to size, coverage or assertions.

The TQM [2] contains metrics related to each of these categories. Additionally, the model is solely based on static analysis making it a suitable starting point for our research.

4.2 RQ 1.1: To what extent is the TQM able to predict test suite effectiveness?

We hypothesize that test effectiveness can be predicted statically, by employing metrics from the TQM [2].

Initially, they held the TQM against a benchmark of systems, normalizing the results such that different systems could be compared to each other. However, we argue that it is more relevant to first explore the raw metrics to see which relate to effectiveness.

We do not use the entire set of metrics as the original model was intended for measuring the entire test code base. Metrics such as code duplication make less sense when taking random sets of tests which are scattered throughout the test code base.

The baseline effectiveness of a test suite is measured using mutation testing. To answer this question, we design an experiment based on work by Inozemtseva *et al.* [6]. Mutants are generated using the default set of mutators [1] provided by PIT, a mutation testing system for Java.

The first problem that needs to be addressed is that of equivalent mutants: mutants that do not change the output program. Suppose a for loop that exits if $i == 10$, where i is incremented by 1 per iteration, a mutant that changes the exit condition to $i >= 10$ will not be detected. Manually removing equivalent mutants is time-consuming and generally undecidable [9].

A commonplace solution is removing all the mutants that survived when the complete test suite was executed [6, 9]. The resulting mutants were detected by at least one of the tests for a specific project. The disadvantage of this approach is that many non-equivalent mutants may also be removed in the process. To compensate for that, from the set of tests available for a project, 1000 subsets of equal size are randomly selected. For each of these subsets the quality and effectiveness will be measured. This allows us to determine if subsets of higher quality are more effective than subsets of lower quality. By generating test suites of fixed sizes, we can account for the size of a test suite. This is beneficial as size is often the leading factor for test effectiveness *e.g.*, larger suites contain more tests and therefore will also cover more code and contain more assertions.

Next, we analyse if the TQM can be used as a predictor for effectiveness. First, we use a limited set of projects from previous research [6]. Depending on the outcome, we generalize and test the model on projects of different programming languages and sizes.

4.3 RQ 1.2 How can the TQM be improved to better predict test suite effectiveness?

To answer this question, we analyse the correlation of static analysis-based metrics [8, 5, 12, 10] with test effectiveness. The experiments for each set of metrics uses the same set-up as RQ1.1. If the metrics do well in measuring the test suite effectiveness *i.e.*, there is a correlation between the two, we will perform another experiment to measure their predictive value.

5 Evaluation Setup

To make sure that we are not just measuring the influence of the size of a test suite on our results, we want to compare test suites of similar sizes. For each test suite, we include at least the following metrics using static source code analysis:

Method coverage Calculated using static call graph analysis [1].

Assertion count Number of times an assertion is done. This will be combined with Lines Of Test Code (TLOC) to calculate assertion density and with the complexity of the production code to calculate the ratio between assertions and complexity.

We count the assertions for each test using static call graph analysis and record the number of calls to an assert method. The assertion count for a test suite is the sum of the number of assertions of the individual tests.

Directness The percentage of covered methods called directly from the unit test scope.

Unit Size Cumulative TLOC of tests. If two test methods call the same utility method the TLOC is only included once.

We define the scope of a unit test as all the methods invoked by the test within the test package including *e.g.*, utility functions and set-up/tear-down methods as used with JUnit.

The dynamically measured data include code coverage (method, line, branch and complexity) using JaCoCo [2] and raw and normalized effectiveness scores us-

[1] `http://pitest.org/quickstart/mutators/`

[2] `http://www.jacoco.org/`

ing PIT. The normalized effectiveness only takes into account the mutants that were reachable by the given test suite[6].

5.1 Projects

Currently, our experiment has run on two Java open-source projects: JFreeChart [3] and JodaTime [4], partially replicating a previous study [6]. Selected projects had in the order of hundreds of thousands Lines of Code (LOC) and thousands test methods. JFreeChart's test suite covers 60% of the methods whereas JodaTime has 90% method coverage.

5.2 Composing Test Suites

We compose test suites of fixed sizes by randomly selecting subsets from the tests available for a project [6]. We create suites with 3, 10, 30, 100, ... tests, this pattern is equal to that of other research [6] allowing us to compare results.

5.3 The base line

We use the correlation between code coverage and mutation scores as a baseline for predicting test suite effectiveness. As the focus is on static code analysis, the static code coverage metric [1] is our first approach.

5.4 Evaluation tool

The analysis tool measures both the static metrics and the dynamic metrics for each individual test. The tool's input are the source code, the sizes of the test suites to create and the number of test suites of each size. We then construct test suites by randomly selecting subsets of required size. The output of the analysis tool is a dataset containing the scores on the dynamic and static metrics for each test suite.

In addition to the actual metric scores, the data required for combining tests is also stored *e.g.*, which mutants and methods each test covers. This allows us to combine test results to simulate suites results *e.g.*, for a test suite with 5 tests we combine the coverage results and detect mutants to get the scores for the suite with 5 tests.

6 Preliminary Results

The experiment reports the scores of the static metrics and the dynamic mutation scores for each composed test suite of JodaTime and JFreeChart. Preliminary results show the static coverage metric has the strongest relation with the raw effectiveness scores. Figures 1 and 2 plot the static method coverage scores against the normalized and raw effectiveness scores for the JFreeChart project.

Using the Pearson coefficient [5] we notice there is a correlation between normalized effectiveness scores and static coverage, when ignoring size, of 0.89 for JodaTime and 0.55 for JFreeChart ($p < 0.0005$). When accounting for size, the correlation drops to nearly zero for both projects.

However, when we compare the static coverage with raw mutation scores the correlation while ignoring size, the correlation is 0.99 for JodaTime and 0.77 for JFreeChart($p < 0.0005$). When we account for size, a low correlation of 0.37-0.66 for the different suite sizes in JFreeChart remains.

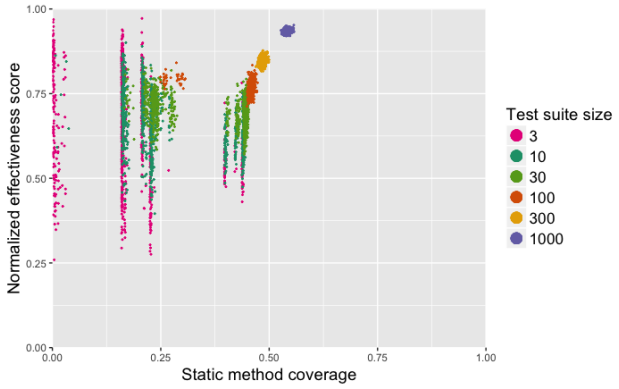

Figure 1: The results from the static method coverage algorithm against the normalized effectiveness score

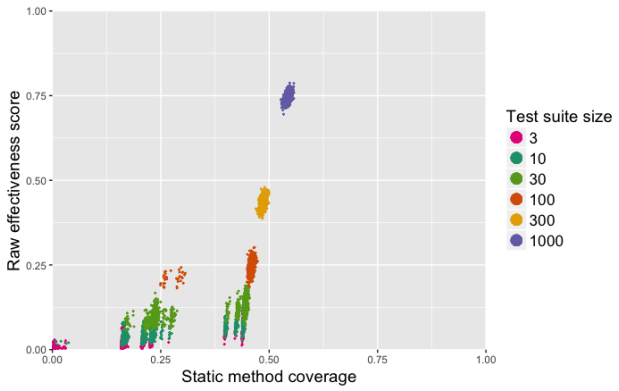

Figure 2: The results from the static method coverage algorithm against the raw effectiveness score

We note that the data set is relatively small. The results are not yet sufficiently conclusive as some metrics need further improved. However, we remark that size and project play an important role. Furthermore, we reproduced the finding that the correlation between dynamic coverage and normalized effectiveness ranged from 0.80 to 0.85 when suite size was ignored, but

[3] https://github.com/jfree/jfreechart
[4] https://github.com/jodaorg/joda-time
[5] Calculated using R's cor.test method

dropped significantly when suite size was controlled for." [6].

7 Limitations

As our experiment builds on previous work [6], the same threats to validity apply. This includes the treatment of equivalent mutants and the way we controlled for size. Also, currently the experiment is only based on two Java projects. This means that the result is not directly generalizable to other programming languages and possibly only limited to other Java projects.

We noticed that there does not seem to be a good predictor for normalized mutation scores which is why we shifted to exploring the raw mutation scores. Further analysis is needed on the differences in impact of normalized and raw mutation scores.

8 Conclusion and Future Work

We research the relation between the dynamically calculated metrics and metrics based on static source code analysis. Based on our preliminary findings, some metrics from TQM could predict the effectiveness of a test suite.

Such a TQM requires tools for the static source code analysis. Additionally, we can gain a deeper insight by simulation of test suites of specific sizes and by aggregating the metrics of individual tests. Such a tool benefits from a large dataset, containing at least the dynamic metrics for each individual test of a set of projects.

In the short term, we will broaden our research contribution by adding more projects to our dataset, including some from the industry. The long term work would be improving and refining the set of metrics *e.g.*, there are already several heuristics that can be applied to improve static analysis of code coverage.

References

[1] Tiago L. Alves and Joost Visser. Static estimation of test coverage. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 55–64, Sept 2009.

[2] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. volume 40, pages 1100–1125, Nov 2014.

[3] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[4] Tom Ball Gunnar Kudrjavets, Nachi Nagappan. Assessing the relationship between software assertions and code quality: An empirical investigation. Technical report, May 2006.

[5] Ferenc Horváth, Béla Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely, and Tibor Gyimóthy. Test suite evaluation using code coverage based metrics. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST́15)*, pages 46–60, 2015.

[6] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[7] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[8] Nachiappan Nagappan. *A Software Testing and Reliability Early Warning (Strew) Metric Suite*. PhD thesis, 2005. AAI3162465.

[9] Akbar S. Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[10] Rudolf Ramler, Michael Moser, and Josef Pichler. Automated static analysis of unit test code. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 2, pages 25–28. IEEE, 2016.

[11] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, Dávid Havas, and Tibor Gyimóthy. Beyond code coverage - an approach for test suite assessment and improvement. In *Software Testing, Verification and Validation Workshops (ICSTW), International Conference on*, pages 1–7. IEEE, 2015.

[12] Kevin van den Bekerom. Detecting test bugs using static analysis tools. Master's thesis, University of Amsterdam, 2016.

[13] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229. IEEE, 2008.

[14] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. volume 29, pages 366–427, New York, NY, USA, December 1997. ACM.