A Replication Study

BINGQUAN WANG, University College London

Abstract

1 INTRODUCTION

Software testing is essential to increase quality of programs. It validates whether a software system is working correctly according to specification by executing the piece of code against test cases. Earlier studies have already shown, for any meaningful program that finding all the faults and proving the program is faults-free is in practice undecidable. As complexity and input data range increase quickly over the years, ensuring the software system to achieve the desired level of quality is more and more difficult. There is always a trade-off between cost of improving testing and cost of leaving undetected faults in a program. It is necessary for developers to measure and predict test quality, given only the actual program and test cases run against it.

Test coverage is a measure of how much code of a program has been executed against a particular test suite. It is believed that higher the coverage, higher the chance to detect faults, because tests can never find faults which have never been executed. However, it is still an open question how strong test coverage criteria is related to effectiveness of detecting faults. There are different studies on this question, but an agreement has not been reached.

A common understanding of code coverage is that it is useful in finding non-executed code, but is less useful in finding non-tested data input. Figure is an example of why code coverage may not be a good measure. Let us suppose we need a piece of code to tell us if input is greater than zero. And the specification is that only when input is greater than zero return true, otherwise return false. By having a test suite containing test cases greater than zero and less than zero, both code achieves 100% statement coverage. However, without testing zero specifically, faulty version of program can never be revealed.

The other technique of measuring test suite quality is mutation testing. Mutation testing tests whether test cases of the software system can detect small human seeded faults. The higher the mutation testing score, the better the test quality, which means it is able to find more human seeded faults. Mutation testing score is one of the most informative measure for quality of test suites as it is a direct measure of fault detection ability. However, there are also two problems with mutation testing. Firstly, human seeded faults are simulations of real faults which are not be the same as real software faults, so ability of finding human seeded faults may not be the same as ability of finding real faults. Another problem is that mutation is difficult to implement, and takes a long time to run. So mutation testing has not been recommended in industry.

Both testing measures have their limitations. And mutation testing is not commonly accepted by industry. It is important for developers and researchers to know if they can still safely use code coverage as a good measure for testing quality. This project is a replication study of a recent paper "Code Coverage is Not Strongly Correlated With Test Suite Effectiveness".

2 RELATED WORK

Most previous work considered correlation of code coverage criteria and ability of finding number of faults. It is natural for developers to manually insert faults to programs, and measure the number of

seeded faults a test suite can detect, which is essentially the same as mutation test. Budd et al. proved mutation testing score is a stronger metric compare to other coverage criteria for test suite evaluation. Li et al, compared three code coverage criteria(edge pair, all use, prime path) with mutation testing, found that mutation testing is the best in detecting hand seeded faults in small programs. Despite the studies did not use large-scale SUTs, with the methodologies that minimise the bias, it is highly possible that mutation testing is better at evaluating test suites.

Wong et al.[14] investigated correlation between fault detection effectiveness and block coverage and correlation between fault detection effectiveness and size. They found that there is a correlation between block coverage and fault detection effectiveness. Also correlation between block coverage and fault detection effectiveness is higher than correlation between size and fault detection effectiveness. For this study, they believe that increasing the number of test cases in a test suite without increasing its coverage does not increase fault detection effectiveness.

Hutchins et al. [7]

Andrew et al. [1] studied behaviour of random selecting test suites and test suites constructed by adding test cases to improve test coverage, that is for the latter test suite if a test case does not increase coverage for a test suite, it is discarded and a new test case is selected. They found that for test suites with the same size, test suites consider coverage results better test effectiveness. Indirectly, it is saying there is a correlation between test effectiveness and coverage.

A study by Namin et al. [11] claimed that both coverage and size have effect on test suite effectiveness. They found a non-linear relationship between test suite size, coverage and test suite effectiveness. They introduced a maths model (log(size) + coverage) which they believed can be used to predict the test suite effectiveness.

Cai et al. [3] investigated relationship between test effectiveness and coverage metrics under different test profile: whole test set, functional test, random test, normal test, exceptional test. They found that for exceptional test, there is a significant correlation between test effectiveness and code coverage. There is no correlation for normal test. There is moderate correlation for functional and random test.

Gopinath et al. [5] measured mutation score and coverage for more than 200 programs for their master suite and automatically generated test suites. They found there is correlation between coverage criteria and fault detection effectiveness for master suites and automatically generated suites. The correlation for master suite is stronger compare to automatically generated suites. They also claim that adding automatically generated suites to master suite does not necessarily increase test effectiveness. So size of test suite is not strongly correlated with effectiveness. (This is Alex work, the other paper we wanted to replicate)

This replication study is about a resent work from Inozemtseva et al. [8]. They studied correlation between different coverage and test effectiveness.

Unlike ealier studies which most of them used mutation testing score directly as effectiveness of fault detection. Inozemtseva et al. [8] defined two effectiveness measures, *raw effectiveness measurement* and *normalised effectiveness measurement*. "The raw kill score is the number of mutants a test suite detected divided by the total number of non-equivalent mutants that were generated for the subject program under test." [8]. "The normalized effectiveness measurement is the number of mutants a test suite detected divided by the number of non-equivalent mutants it covers." [8]. A test suite cover the mutant means test cases in the test suite execute the mutated line of code. Or the mutant can be detected by the test suite.

The reason to introduce this normalised effectiveness is that when size of test suite is controlled, there are limited lines of code and mutants it covers. Authors are more interested in effectiveness relative to the test suite itself. For example, test suite A kills 10 mutants and test suite B kills 100 mutants, for raw

effectiveness suite B is certainly higher. If suite A has only 12 covered non-equivalent mutants and suite B has 200, suite A will have a higher normalised effectiveness than suite B.

The result of Inozemtseva et al.'s work is that when test suite size is not controlled, there is moderate to high correlation between all code coverage criteria and effectiveness; when test suite size is controlled, there is low to moderate correlation between all coverage types and effectiveness. Also indirectly saying size is a confounding factor.

As above studies show, most studies claim that there is some relationship between coverage, size and test effectiveness. But they do not agree to each other about how strong the relationship is.

3 CONCEPT

3.1 Testing Terminology

Terminologies used for testing in general and for this paper are defined as follows:

- Test case/methods: a independent unit test. It isolates a fragment of code (normally a functional method) and validates its correctness. Ideally, unit test should not go outside its own method boundary. When methods interact with each other, it is more difficult to identify which component is the cause of failure. Test cases are sometimes also called test methods.
- Test suite: a collection of test cases which are used to validate a set of behaviour.
- Master test suite: A test suite contain all test cases written by developer. In this paper, all test suites evaluated are strict subsets of the master test suite.

3.2 Mutation Testing

Mutation testing is a technique which tests if test suite can detect human seeded faults. Faulty programs are created by changing original program syntax which is called a **mutant**, each mutant contains a different syntactic change. Every mutant is executed against test suites, if the result is different to the original program, then the mutant is **killed**, otherwise the mutant has **survived**.

For a surviving mutant, there are two possible reasons for it to happen. Either the test suite does not contain test cases cover the fault, or the mutant is syntactically different but semantically the same. Those mutants semantically the same can never be killed, which are called **equivalent mutants**. For example, Algorithm 1 is identical to algorithm 2 in terms of functionality. A particular mutant changing algorithm 1 to algorithm 2 can never be detected. Mutation score by formal definition is the ratio of number of killed mutants over the number of non-equivalent mutants [10].

ALGORITHM 1: Return the number with highest value

Data: Integer A and Interger B
Result: Return the number with highest value
if $A > B$ **then**
  return A;
else
  return B;
end

ALGORITHM 2: Return the number with highest value

Data: Integer A and Interger B
Result: Return the number with highest value
if $A >= B$ **then**
  return A;
else
  return B;
end

Determine if a survived mutant is equivalent is undecidable for computers as shown in the previous study [2]. Currently, a very common practice in research is to assume the test suite is adequate and treat all surviving mutants as equivalent mutants [10]. This is a overestimation of equivalent mutants but allows researchers to work on large programs.

3.3 Coverage Criteria

There are three coverage criteria used in the project: statement, decision and modified condition coverage.

Statement coverage is the percentage of how many lines of statements have been executed for a particular test suite.

Decision coverage

Modified condition coverage(MCC)

3.4 Effectiveness

As mentioned in Section 2, there are two effectiveness introduced in this paper: raw effectiveness measurement and normalised effectiveness measurement. Author did not give mathematical expression for two effectiveness.

Raw effectiveness is the number of killed mutants of a test suite divided by the number of killed mutants of master test suite. For a test suite t, master suite T and program P, raw effectiveness should be:

$$rawEffectiveness = \frac{\#killedMutants(t,P)}{\#killedMutants(T,P)}$$

For a test suite normalised effectiveness and is calculated by killed mutants of this suite over covered non-equivalent mutants of the same suite. Directly from authors' definiation we have the expression, for a test suite t and program P:

$$normalisedEffectiveness = \frac{\#killedMutants(t,P)}{\#coveredNon\text{-}equivalentMutants(t,P)}$$

Covered non-equivalent mutants need further decomposition. A mutant can have three status for a particular test suite: covered and killed by test cases in this test suite, covered but not killed by this test suite but killed by test cases outside this test suite and surviving or equivalent. Covered non-equivalent mutants are total mutants covered taking away surviving mutants. So the final equation of normalised effectiveness for a test suite t and program P is:

$$normalisedEffectiveness = \frac{\#killedMutants(t,P)}{\#totalCoveredMutants(t,P) - \#equivalentMutants(t,P)}$$

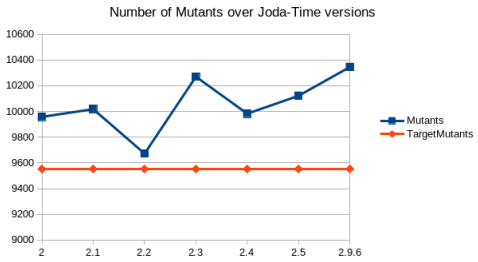

Fig. 1. Joda Time total mutants

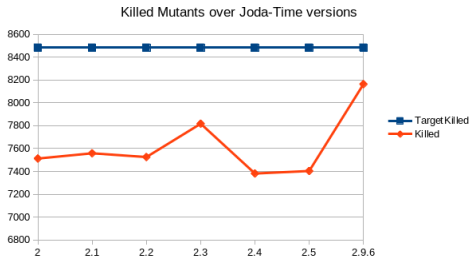

Fig. 2. Joda Time killed mutants

3.5 Correlation Measurement

4 METHODOLOGY

4.1 Procedure

The procedure used in this research is as follows:

(1) Use a mutation testing tool(PIT) to produce faulty program and run mutants against master test suite. Get mutant status for every test case.
(2) Generate a large number of test suites by randomly selecting tests cases from master test suite, until the test suite reaches its pre-defined size.
(3) For each test suite:
- Measure coverage criteria (CodeCover) for different test suites.
- Determine effectiveness of different test suites using mutation information and coverage criteria.

(4) Analyse correlation between different coverage criteria and effectiveness.

4.2 Subjects under test

The original paper used five following subject programs:

(1) Apache POI [12]: a Java API for Microsoft Documents
(2) Closure [4]: a tool for making JavaScript download and run faster.
(3) HSQLDB [6]: a Java SQL relational database.
(4) JfreeChart [9]: a Java chart library for users to display professional quality charts.
(5) Joda Time [13]: A replacement for Java time and date class

These five subjects are selected on purpose using the following criteria:

(1) The program needs to be large enough to have more than 100,000 SLOC;
(2) It needs to be an actively developed Java program;
(3) It contains at least 1000 test cases;
(4) The project needs to use Ant as its build system;
(5) The project uses JUnit as a test harness.

However no versions or git hashes of subject programs were given in the original paper or in its artefact page. As the original paper was published in 2014, various number of test subjects versions between 2012 to 2014 were examined with different PIT versions, but there was no result that was close enough to what was reported in the original paper. Figure 1 and figure 2 are examples of different versions of Joda Time running against PIT version 1.0.0. We contacted authors of the original paper, and acquired their project repository.

Table 1. Java Compiler Setting

		Java 6		Java 7		Java 8	
Project	Version	Compile	Test	Compile	Test	Compile	Test
Apache POI	3.9(author)	Success	Fail	Success	Fail	Success	Fail
	3.9(GitHub)	Success	Success	Success	Success	Success	Fail
Closure	20130227(author)	Fail	Fail	Fail	Fail	Fail	Fail
	20130227(GitHub)	Success	Success	Success	Success	Success	Fail
HSQLDB	2.2.8	Success	Success	Success	Success	Success	Fail
JfreeChart	1.0.8	Success	Success	Success	Success	Success	Fail
Joda Time	2.0	Success	Success	Success	Success	Success	Fail

There are some environment settings need to be adjusted locally. Mutation testing tool PIT used for this project requires green test suite, which means all tests must pass. During my replication, some projects provided by the author could not compile or did not pass all tests. The first Java compiler used in the replication was Java 8, but non of the projects passed all tests. HSQLDB, JfreeChart, Joda Time worked with no errors using Java 7. According to ant build system, recommanded Java compiler for Closure was Java 6, but non of the compiler version complied it successfully. For Apache POI, Java 6 and 7 compilation was successful but resulted failing tests.

There was no command to exclude failing tests in author's repository, and the report log suggests the Apache POI and Closure were working correctly. I downloaded source code from GitHub according to author's repository. Table 1 is a summary of projects and compiler version. Java 7 was selected as final replication compiler.

The final environment settings used in this replication study are:

- Operating System: Ubuntu 14.04.5 LTS;
- Java Compiler: 1.7.0_u131;
- ant build system: 1.9.3;
- JUnit: 3.8 for compiling projects, 4.10 for running PIT;
  Other systems should work, as long as using Java compiler 7, ant version greater than 1.8; and having JUnit 3 and JUnit 4 at the same time.

4.3 Mutation Testing

Mutation testing of the program was conducted using an automated mutilation testing tool PIT.

4.4 Test suite generation

4.5 Coverage Measurement

4.6 Measuring Effectiveness

5 RESULTS

5.1 PIT Laura's results

	Property	Apache POI	Closure	JFreeChart	Joda Time
Table 2	Generated Mutant	27565	30779	29699	9552
	Detected mutant	17935	27325	23585	8483
	Equivalent mutant	9630	3454	6114	1069
PIT default	Generated Mutant	27565	30779	29699	9552
	Killed mutant	17935	23178	9808	7503
	Covered live	3458	3447	6106	1066
	Not covered	6172	4154	13785	983
	Covered live + not covered	9630	7601	19891	2049
Laura's	Surviving mutant	3469	3454	6125	1069

5.2 Correlation Replication

Table 3, normalised effectiveness replicate

	coverage type	JFreeChart
Table 3	Statement	0.50
	Decision	0.53
	Mod. Cod.	0.53
My random suite result	statement	0.55
	Decision	0.57
	Mod. Cod.	0.57

Table 4, raw effectiveness replicate

	coverage type	JFreeChart
Table 4	Statement	0.91
	Decision	0.95
	Mod. Cod.	0.92
My random suite result	statement	0.90
	Decision	0.92
	Mod. Cod.	0.91

6 CONCLUSION

REFERENCES

[1] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[2] Timothy A Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta informatica* 18, 1 (1982), 31–45.

[3] Xia Cai and Michael R Lyu. 2005. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT software engineering notes* 30, 4 (2005), 1–7.

[4] Closure Compiler. [n. d.]. Closure Compiler. ([n. d.]). Retrieved Aug 25, 2017 from https://github.com/google/closure-compiler

[5] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.
[6] HSQLDB. [n. d.]. HSQLDB. ([n. d.]). Retrieved Aug 25, 2017 from http://hsqldb.org/
[7] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*. IEEE, 191–200.
[8] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 435–445.
[9] JfreeChart. [n. d.]. JfreeChart. ([n. d.]). Retrieved Aug 25, 2017 from http://www.jfree.org/jfreechart/
[10] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.
[11] Akbar Siami Namin and James H Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 57–68.
[12] Apache POI. [n. d.]. Apache POI. ([n. d.]). Retrieved Aug 25, 2017 from https://poi.apache.org/
[13] Joda Time. [n. d.]. Joda Time. ([n. d.]). Retrieved Aug 25, 2017 from http://www.joda.org/joda-time/
[14] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. 1994. Effect of test set size and block coverage on the fault detection effectiveness. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*. IEEE, 230–238.