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, as shown in Algorithm 1 and Algorithm 2. 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.

ALGORITHM 1: Determine if input is positive (correct)

```
Data: Integer
Result: return true when input is greater than 0 else return false
if A > 0 then
    return true;
else
    return false;
end
```

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

ALGORITHM 2: Determine if input is positive (faulty)

Data: Integer A
Result: return true when input is greater than 0 else return false
if *A >= 0* **then**
  return ture;
else
  return false;
end

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 CONCEPT

2.1 Testing Terminology

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

- Test case/methods: an 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.

2.2 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 of the program code have been executed for a particular test suite. For 100% statement coverage, each statement need to be executed at least once.

Decision coverage is a measure of how many decision branches have been executed, such as if-else, switch and try-catch branches. The difference of decision coverage to statement coverage is that all *else* branches need to be executed even there is no code in *else* branch. For Algorithm 3, a test case of

condition value being true will give full statement coverage but only half decision coverage. To achieve full decision coverage, both true and false condition needs to be tested.

ALGORITHM 3: If-else with implicit else branch

```
Data: Boolean: condition
if condition then
    statements;
else
end
```

Modified condition coverage (MCC) is a deeper measure for if statements. It considers decision expressions in if statements. A condition is the simplest boolean expression, which can not be further break down. And a decision is composed by conditions and boolean operators. MCC subsumes modified condition/decision coverage (MC/DC) [9]. For 100% MC/DC coverage every condition (boolean expression) must at least once be shown to have effect to the overall result of decision. It is the most complicated and strongest coverage criterion out of three measured.

2.3 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 has been **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 4 is identical to Algorithm 5 in terms of functionality. A particular mutant changing Algorithm 4 to Algorithm 5 can never be detected. Mutation score by formal definition is the ratio of number of killed mutants over the number of non-equivalent mutants [14].

ALGORITHM 4: 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 surviving mutant is equivalent is undecidable for computers as shown in the previous study [2]. It requires human inspection to determine equivalent mutants which is time consuming. Currently, a very common practice in research is to assume the test suite is adequate and treat all surviving mutants as equivalent mutants [14]. This is a overestimation of equivalent mutants but allows researchers to work on large programs.

ALGORITHM 5: 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

3 RELATED WORK

Most previous work considered correlation of code coverage criteria and ability of finding number of faults. To determine the fault detection ability researches manually inserted known faults to programs, either previous fixed bugs or syntactic change, then measure the number of seeded faults a test suite can detect, which is essentially the same as mutation testing. 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.[19] 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. [11]

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. [15] 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.

Gligoric et al. [6] measured both Kendall and Pearson to examine correlations to mutation kill for a set of criteria, Their work considers 15 Java programs and 11 C programs, selected not randomly but primarily from container classes used in previous studies and the classic Siemens subjects. Their larger projects (JodaTime, JFreeChart, SQLLite, YAFFS2) were chosen opportunistically. The study suggested that the correlation between coverage and effectiveness in real systems is largely due to the correlation

between coverage and size; it also suggested that results from automatically generated and manually generated suites do not generalize to each other.

Gopinath et al. [7] 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. [12]. 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. [12] 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." [12]. "The normalized effectiveness measurement is the number of mutants a test suite detected divided by the number of non-equivalent mutants it covers." [12]. 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 only has 12 covered non-equivalent mutants and suite B has 200, suite A will have a higher normalised effectiveness than suite B. The definition of normalised effectiveness captures the idea that an effective suite is very good at finding faults with in the code that it runs.

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 saying size is a correlated with effectiveness.

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.

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 it 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 [17]: a Java API for Microsoft Documents
(2) Closure [5]: a tool for making JavaScript download and run faster.
(3) HSQLDB [10]: a Java SQL relational database.
(4) JfreeChart [13]: a Java chart library for users to display professional quality charts.
(5) Joda Time [18]: 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.

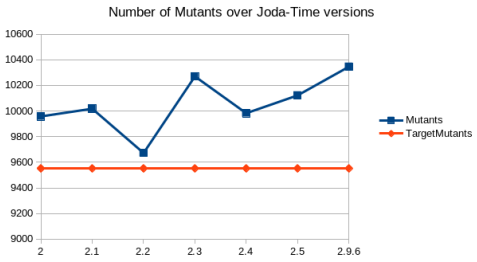

Fig. 1. Joda Time total mutants

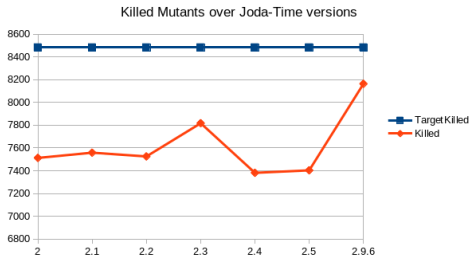

Fig. 2. Joda Time killed mutants

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 suspected that authors made some change after they got result for Apache POI and Closure. So 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:

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

- 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 [16], which is one of the most widely used open source Java mutation testing system. Version of PIT used for the original paper was 0.30-SNAPSHOT. PIT run tests against automatically generated mutants. Firstly it performs statement coverage for the tests, then use coverage information to pick test cases targeting a particular mutant.

For a mutant, PIT reports one of the following result status:

- Killed: mutant has been killed by the test suite.
- Survived: mutants have at lease one test case execute it but survived.
- No coverage: mutants lived because no test case executed the line of code where the mutant was created.
- Time out: the mutants causes an infinite loop, for example mutate loop counter in a for loop.
- Non viable: the mutant is invalid and can not be loaded by the Java Virtual Machine (JVM).
- Memory error: the mutation requires more memory to be used by system.
- Run error: there is something wrong when running the mutant.

Under normal circumstances, there should be no non viable mutants or errors.

Although author did not mention in the paper, she has made some modification to PIT. For the original PIT, if a mutant is killed by a test case, the test case is logged, then PIT moves to next mutant. As a result, PIT firstly provides a boolean information of whether mutant is killed, then only the first test case killed the mutant was reported. The research wanted information about all test cases that detect a mutant, so modification is required. For the modified PIT, a mutant run against all possible test cases target it, PIT moves to next mutant until covered test cases are executed, and result for every test case and its corresponding mutant is logged in a log file. The modification is valid for 4 out of 5 subjects, but generated different results for HSQLDB compare to PIT default report. This will be discussed in more detail in section 5.1

4.4 Test suite generation

For each subject program, Javas reflection API was used to identify all of the no-argument test methods in the programs master test suite. Then new test suites were generated size by randomly selecting a subset of master suite without replacement until predefined size was reached.

Size of the test suite follow the pattern: 3, 10, 30, 100, up to the largest number follow this pattern and less than the total test cases for the project. The largest suite used for Apache POI and JFreeChart was 1000, and for Closure and Joda Time was 3000.

But in replication, when we selected random subsets of the project master test suites, some of the random test suites fail to pass unit test, even though they pass in the master test suite. We contacted author for this problem, and got reply that there were failing tests excluded, but we can not find the list of excluded tests in the archive provided.

we investigated the chance of appearance of failing random test suite. And we found that it happened rarely as shown in Table 2. Running random test suite analysis is very time consuming, finding all failing test combination is impossible for the given time. So we decided to ignore the failing methods as they do not have significant influence on results.

Table 2. Number of failing tests appeared in random test suite selection

Size	Apache POI	Closure	JFreeChart	Joda Time
3	-	-	-	-
10	-	-	-	1
30	-	-	1	-
100	-	-	1	-
300	-	-	-	1
1000	-	-	-	-
3000	-	-	-	-

4.5 Coverage Measurement

Coverage criteria are measured using an open source Java coverage analysis tool CodeCover [4]. Statement, decision and modified condition types are used in the project.

Statement coverage is measured for the following statement type:

- assignments and method calls
- variable and member declaration with an assignment
- break
- continue
- return
- throw
- empty statements

All condition statements such as if and for are not considered for statement coverage.

Decision branches are measured for the branches:

- if-else statements one branch for if and another for else
- switch-case statements a branch for each case and one for default

- try-catch statements one bran for every catch, one branch for exceptions not in catch cases and one branch for successful execution.

MCC coverage consider the boolean expressions in:

- if
- for
- while and do...while

Boolean expressions not in decision or looping statements are ignored, such as assignments.

4.6 Effectiveness

As mentioned in Section 3, 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:

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

The number of covered non-equivalent mutants is the maximum mutants a test suite can possibly detect. 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:

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

4.7 Correlation Measurement

Kendalls τ correlation coefficient is used in the original paper. Kendalls τ is similar to the more common Pearson coefficient but does not assume that the variables are linearly related or that they are normally distributed. Rather, it measures how well an arbitrary monotonic function could fit the data. Kendalls τ was used to avoid unnecessary assumptions.

The original paper used the Guildford scale [8] for verbal description, in which correlations with absolute value less than 0.4 are described as low, 0.4 to 0.7 as moderate, 0.7 to 0.9 as high, and over 0.9 as very high.

5 RESULTS

5.1 PIT results

At the beginning of our replication, with authors' SUT repositories and her suggested modified version of PIT, we were not able to reproduce results that were close enough to the original paper. We contacted author for help, she helped us to understand her log file and how she got the results.

As mentioned in section 4.3, for authors' modified PIT, there are information for every mutant and its corresponding test cases. For the number of surviving mutants, she counted the number of mutants logged as survived in her modified log. The number of detected mutants was calculated by subtracting number of surviving mutants from number of total generated mutants.

However, PIT is more complicated than just have killed and surviving mutants as shown in section 4.3. Under normal circumstances, mutants can also have outcomes of not covered and time out. Whether not covered mutants are killed or surviving can not be detected by current test suite, so they are not reported as killed or surviving mutants. Time out mutants are considered as killed as they cause a infinite loop and force PIT to move to next mutant. Therefore the way she generate killed mutants was wrong, she did not consider mutants with outcome of not covered. Table 3 includes information extracted from her repository and paper without running PIT.

For Apache POI, number of equivalent mutants reported in the paper is the same as adding surviving mutants and no covered mutants in PIT default report. Also number of killed mutants match with PIT default report. However for the other four projects, number of surviving mutants in the paper match with modified PIT log, and the number of killed mutants are calculated using total number of generated mutants subtract number of surviving mutants. From this we can tell for all five mutants, the numbers reported in the paper were wrong for two reasons. Firstly, not covered mutants were not considered. Secondly, the way she calculated the number of killed mutants was inconsistent.

Furthermore, looking at PIT default report and modified report, number of survived mutants do not match. For Apache PIT, Closure, JfreeChart and Joda Time, the difference is very small and can be accepted. But for HSQLDB, the difference is too significant to ignore. We investigated the cause, during the running of PIT, there were lots of standard error output. For normal process, if there are errors, PIT aborts and reports them. However, for HSQLDB, PIT run successfully along with reporting errors. PIT is a very complex system, it was not possible for us to resolve the problem and modify it to give correct results in the given time. Thus, we decided to discard usage of HSQLDB.

Table 3. PIT Report

	Property	Apache POI	Closure	HSQLDB	JFreeChart	Joda Time
Original Paper	Generated Mutant	27565	30779	50302	29699	9552
	Detected mutant	17935	27325	50125	23585	8483
	Equivalent mutant	9630	3454	177	6114	1069
PIT default log	Generated Mutant	27565	30779	50302	29699	9552
	Killed mutant	17935	23178	8150	9808	7503
	Survived	3458	3447	5376	6106	1066
	Not covered	6172	4154	36775	13785	983
	Survived + no covered	9630	7601	42152	19891	2049
Modified log	Surviving mutant	3469	3454	177	6125	1069

Table 4 is PIT mutation testing replication result. For Apache POI and Closure, we fetched repository from GitHub as authors' could not compile and test straight away. For JFreeChart, and Joda time, we used repository as author provided. We used authors' modified PIT, as it provides essential mutant outcome for correlation analysis. For these four projects, we have got similar PIT results to original paper. The difference can be caused by randomness of PIT automation and different Java compiler version. The replication PIT results are used for further analysis in this study.

Table 4. Replication PIT Report

	Property	Apache POI	Closure	JFreeChart	Joda Time
PIT default log	Generated Mutant	27565	30779	29699	9552
	Killed mutant	17923	22841	9868	7486
	Survived	3466	3515	6109	1074
	Not covered	6176	4423	13722	992
Modified log	Surviving mutant	3472	3518	6122	1076

5.2 Is Coverage Correlated With Effectiveness When Size Is Ignored?

When we analysis authors' data used for raw effectiveness, we found that the number of total killed mutants used in the calculation was wrong. Recall section 5.1 that row of detect mutant in table 3 also counts not covered mutant for three project: Closure, JFreeChart and Joda Time. These numbers were used as total killed mutants when calculating raw effectiveness. Thus the actual raw effectiveness for those three projects is: For random subtest t, master suite T and program P

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

However this misuse of killed mutants did not affect overall correlation result. Adding total not covered mutants does not effect the ranking of raw effectiveness. Since correlation only interests in ranking of value, In addition we validated the results of raw effectiveness by changing total number of killed mutants to what was reported in PIT default log and got identical results. So the raw effectiveness correlation of original is still valid despite the mistake of detected mutants.

Our first interest is whether coverage is correlated with fault detection effectiveness when size is ignored. Table 5 and table 6 show the Kendall τ correlation coefficients for raw effectiveness and normalised effectiveness discussed in section 4.6. All 4 projects showed high correlation between raw effectiveness and coverage criteria. When normalisation applied, the correlation decreased for every subject. JFreeChart drooped the most, from very high correlation to moderate while other projects still maintain high. So we can conclude that there is moderate to high correlation between coverage criteria and fault detection effectiveness when size is ignored. In other words, when size of test suite is ignored, the higher the coverage the more faults a test suite can detect fault.

Table 5. Kendall correlation between raw effectiveness and coverage criteria when size is ignored

	Original			Replication		
Project	Statement	Decision	MCC	Statement	Decision	MCC
Apache POI	0.94	0.94	0.94	0.92	0.94	0.94
JFreeChart	0.91	0.95	0.92	0.90	0.92	0.90
Joda Time	0.85	0.85	0.85	0.94	0.94	0.93

Note: All entries are significant at 99% level

Table 6. Kendall correlation between normalised effectiveness and coverage criteria when size is ignored

	Original			Replication		
Project	Statement	Decision	MCC	Statement	Decision	MCC
Apache POI	0.75	0.76	0.77	0.77	0.77	0.81
JFreeChart	0.50	0.53	0.53	0.55	0.57	0.57
Joda Time	0.80	0.80	0.80	0.84	0.84	0.84

Note: All entries are significant at 99% level

5.3 Is Coverage Correlated With Effectiveness When Size Is Controlled?

We computed the Kendall correlation coefficient between effectiveness and coverage for each project, each suite size, each coverage type, and both effectiveness measures.

In general, when test suite size was controlled, raw effectiveness measurements had moderate correlations with coverage and normalized effectiveness measurements had low to moderate correlations.

For Joda Time, our replication result was very different to original paper as shown in Table 13 and Table 14. For both raw and normalised effectiveness, original paper had near zero correlations for all type of coverage and any size. However, according to our replication, there is moderate correlation between raw effectiveness and all three coverage, and there is low correlation between normalised effectiveness and coverage.

Table 7. Apache POI Raw

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	0.85	0.84	0.85	0.80	0.81	0.80
10	0.75	0.73	0.77	0.75	0.77	0.77
30	0.61	0.67	0.67	0.56	0.58	0.58
100	0.51	0.48	0.46	0.55	0.52	0.55
300	0.67	0.64	0.63	0.60	0.63	0.60
1000	0.77	0.63	0.69	0.79	0.72	0.64

Note: All entries are significant at 99% level

6 DISCUSSION

7 FUTURE WORK

8 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.

Table 8. Apache POI Normalised

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	-0.17	-0.13	-0.07	-0.09⋆	-0.08⋆	-0.01⋆
10	0.14	0.16	0.22	0.18	0.20	0.20
30	0.18	0.27	0.28	0.22	0.21	0.21
100	0.15	0.21	0.21	0.27	0.30	0.32
300	0.41	0.40	0.42	0.52	0.45	0.46
1000	0.49	0.46	0.49	0.49	0.48	0.48

Note: All entries are significant at 99% level

Table 9. Apache POI Raw

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	0.79	0.80	0.80			
10	0.71	0.72	0.69			
30	0.69	0.73	0.70			
100	0.70	0.66	0.57			
300	0.65	0.62	0.56			
1000	0.52	0.52	0.46			

Note: All entries are significant at 99% level

Table 10. Closure Normalised

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	0.12	0.14	0.18			
10	-0.14	-0.13	-0.04⋆			
30	-0.27	-0.22	-0.16			
100	-0.04⋆	-0.03⋆	-0.01⋆			
300	0.12	0.12	0.15			
1000	0.12	0.10	0.13			
3000	0.13	0.17	0.19			

Note: All entries are significant at 99% level

[4] CodeCover. [n. d.]. CodeCover. ([n. d.]). Retrieved Aug 26, 2017 from http://codecover.org

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

[6] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on*

Table 11. Closure Raw

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	0.70	0.84	0.56	0.73	0.85	0.57
10	0.66	0.83	0.68	0.59	0.68	0.56
30	0.65	0.78	0.69	0.40	0.40	0.37
100	0.53	0.68	0.57	0.33	0.39	0.38
300	0.46	0.64	0.56	0.40	0.54	0.51
1000	0.46	0.62	0.60	0.38	0.60	0.59

Note: All entries are significant at 99% level

Table 12. JfreeChart Normalised

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	-0.25	-0.08	-0.20	-0.20	-0.06⋆	-0.18
10	-0.42	-0.26	-0.33	-0.32	-0.21	-0.26
30	-0.28	-0.17	-0.19	-0.02⋆	0.03⋆	0.05⋆
100	-0.09	-0.01⋆	0.03⋆	0.00⋆	0.08	0.14
300	0.03⋆	0.11	0.20	0.01⋆	0.10	0.21
1000	0.06	0.13	0.20	0.06⋆	0.13	0.18

Note: Entry marked with ⋆ is not significant at 99% level

Table 13. Joda Time Raw

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	-0.04⋆	-0.03⋆	-0.05⋆	0.48	0.56	0.50
10	0.01⋆	0.00⋆	0.00⋆	0.52	0.60	0.54
30	0.00⋆	0.00⋆	0.00⋆	0.56	0.61	0.56
100	0.00⋆	0.01⋆	0.04⋆	0.55	0.60	0.55
300	0.03⋆	0.04⋆	0.04⋆	0.60	0.63	0.59
1000	0.00⋆	0.01⋆	0.00⋆	0.69	0.72	0.64
3000	-0.01⋆	0.00⋆	0.00⋆	0.59	0.69	0.53

Note: Entry marked with ⋆ is not significant at 99% level

Software Testing and Analysis. ACM, 302–313.

[7] 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.

[8] Joy Paul Guilford. 1942. Fundamental statistics in psychology and education. (1942).

Table 14. Joda Time Normalised

	Original			Replication		
Size	Statement	Decision	MCC	Statement	Decision	MCC
3	-0.00⋆	-0.00⋆	-0.00⋆	-0.00⋆	-0.00⋆	-0.01⋆
10	-0.02⋆	-0.01⋆	-0.01⋆	0.19	0.21	0.20
30	-0.00⋆	-0.00⋆	-0.01⋆	0.15	0.17	0.21
100	-0.00⋆	-0.01⋆	0.02⋆	0.27	0.30	0.32
300	0.03⋆	0.04⋆	0.04⋆	0.33⋆	0.35	0.36
1000	0.00⋆	0.00⋆	0.00⋆	0.39	0.42	0.43
3000	-0.04⋆	-0.03⋆	- 0.02⋆	0.10	0.15	0.18

Note: Entry marked with ⋆ is not significant at 99% level

[9] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. 2001. A practical tutorial on modified condition/decision coverage. (2001).

[10] HSQLDB. [n. d.]. HSQLDB. ([n. d.]). Retrieved Aug 25, 2017 from http://hsqldb.org/

[11] 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.

[12] 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.

[13] JfreeChart. [n. d.]. JfreeChart. ([n. d.]). Retrieved Aug 25, 2017 from http://www.jfree.org/jfreechart/

[14] 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.

[15] 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.

[16] PIT. [n. d.]. PIT. ([n. d.]). Retrieved Aug 26, 2017 from http://pitest.org

[17] Apache POI. [n. d.]. Apache POI. ([n. d.]). Retrieved Aug 25, 2017 from https://poi.apache.org/

[18] Joda Time. [n. d.]. Joda Time. ([n. d.]). Retrieved Aug 25, 2017 from http://www.joda.org/joda-time/

[19] 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.