Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems

Pavneet Singh Kochhar, Ferdian Thung, and David Lo
Singapore Management University, Singapore
{kochharps.2012, ferdiant.2013, davidlo}@smu.edu.sg

***Abstract*—During software maintenance, testing is a crucial activity to ensure the quality of program code as it evolves over time. With the increasing size and complexity of software, adequate software testing has become increasingly important. Code coverage is often used as a yardstick to gauge the comprehensiveness of test cases and the adequacy of testing. A test suite quality is often measured by the number of bugs it can find (aka. kill). Previous studies have analysed the quality of a test suite by its ability to kill mutants, i.e., artificially seeded faults. However, mutants do not necessarily represent real bugs. Moreover, many studies use small programs which increases the threat of the applicability of the results on large real-world systems.**

In this paper, we analyse two large software systems to measure the relationship of code coverage and its effectiveness in killing *real bugs* from the software systems. We use Randoop, a random test generation tool to generate test suites with varying levels of coverage and run them to analyse if the test suites can kill each of the real bugs or not. In this preliminary study, we have performed an experiment on 67 and 92 real bugs from Apache HTTPClient and Mozilla Rhino, respectively. Our experiment finds that there is indeed statistically significant correlation between code coverage and bug kill effectiveness. The strengths of the correlation, however, differ for the two software systems. For HTTPClient, the correlation is *moderate* for both statement and branch coverage. For Rhino, the correlation is *strong* for both statement and branch coverage.

Keywords*—*Code Coverage, Bugs, Test Suite Effectiveness

I. Introduction

Testing is an integral part of software development process and is crucial to improve the quality of software. Increasing size and complexity of software has necessitated the need to improve software testing. However, complete testing is often infeasible as there is a trade-off between the cost of testing and the number of faults it can find. Effectiveness of testing is often measured by the quality of a test suite, i.e., the ability of a test suite to uncover faults in a program. A test suite that reveals more bugs is considered of higher quality than the one which reveals less bugs.

One metric that is commonly used to measure the adequacy of testing is code coverage, that is, a measure of the set of lines of code or branches, that are executed by a set of tests[1]. Code coverage gives an idea of the thoroughness of a test suite by measuring the amount of code covered by the test suite. This is intuitive, since a test suite which does not cover a particular part of a code will not be able to reveal bugs in that part.

Two recent studies by Gopinath et al. [1] and Inozemtseva et al. [2] investigate the correlation between code coverage and bug detection capability. Gopinath et al. analyze hundreds of open-source projects and measure the quality of test suites of various coverage levels. They use test cases from the project as well as generate test cases using Randoop [3]. However, projects used in this study are very small ranging from 10 LOC to 10,000 LOC. Inozemtseva et al. perform a similar experiment that involves five large software systems. They analyse the relationship between test suite size, coverage and effectiveness. These two studies use mutants, i.e., artificially injected bugs, and measure test suite effectiveness by its ability to kill mutants. However, it is not clear whether the effectiveness of a test suite in killing mutants is representative to its effectiveness in killing real bugs.

In this study, we investigate two large projects Apache HTTPClient[2] and Mozilla Rhino[3] to investigate the relationship between test suite size, code coverage, and effectiveness. Since mutants do not necessarily represent real bugs, in this study, rather than using mutants, we use real bugs from the issue tracking systems of these projects. We use Herzig et al.'s manually classified dataset and fetch the bug id's of issue reports which have been manually labelled as bugs [4]. We measure test suite effectiveness by its ability to kill these real bugs. We use Randoop, a feedback directed random test generation tool, to generate test suites of various coverage levels. We then measure the correlation between the coverage levels of these test suites and their effectiveness.

In this study, we analyse the following research questions:

RQ1: *Is there a correlation between a test suite's size and its effectiveness?*
RQ2: *Is there a correlation between a test suite's coverage and its effectiveness?*

The contribution of this paper is as follows: We conduct a study on two large open-source software projects with the objective of understanding the correlation between the test suite size, coverage and effectiveness. This study is performed using real bugs rather than artificially injected mutants.

The structure of this paper is as follows. In Section II, we briefly describe bug linking, automatic test generation, code coverage, and point biserial correlation test. In Section III, we describe our dataset and overall framework for this study. We present the results of our empirical study and threats to validity in Sections IV and V respectively. Related work is presented in Section VI. Section VII concludes and describes future work.

[1]In this work, we focus on line and branch coverage. We do not consider other more expensive coverage criteria, e.g., path coverage, etc.

[2]http://hc.apache.org/httpclient-3.x/

[3]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

II. Preliminaries

In this section, we describe automatic test generation, specifically its importance and the test generation tool used in this paper, namely Randoop. We then briefly describe code coverage as the metric used to measure the quality of a test suite. Lastly, we describe about point biserial correlation, which we used to measure the correlation between code coverage and test suite effectiveness.

A. Automatic Test Generation and Randoop

With the goal to free developers from the need to generate their own test cases and thus allow them to focus more on the development of the software, researchers have proposed a number of automated test cases generation techniques. This would allow test phase to be performed fully automatically and to get a higher code coverage than the manually generated ones, as well as significantly reduce the generation time. In our work, we use Randoop to automatically generate the test cases.

Randoop is a feedback-directed random testing approach for Java [3]. It generates JUnit random tests for Java programs by considering the output feedback of the generated test cases when executing the test inputs. It is a fully automatic approach, in terms that it does not require any specific parameters that would need some tuning for different cases. Randoop has been shown to be able to increase the code coverage and to improve the error detection capability. Specifically, it was able to find real bugs in the real software systems. Thus, it is a suitable random test generation tool for our purpose, where we want to generate random test cases that can uncover real bugs.

B. Code Coverage

Code coverage is a set of measures for test quality that calculates the percentage of source code that is executed by a given test suite [5]. Code coverage can be measured at different granularity levels such as function coverage, statement coverage, branch coverage, and condition coverage. In our work, we use statement and branch coverage.

Statement coverage measures the proportion of the statements that are executed by the test suite.

Branch coverage measures the proportion of branches in the control structure (e.g., *if* and *case* statements) that are executed by the test suite.

C. Point Biserial Correlation

Point Biserial Correlation is used to measure the correlation between two variables when one of them is naturally dichotomous [6]. This means that the variable is naturally takes value of 0 or 1 instead of discretized to 0 or 1. This correlation is suitable for our purpose, where one of our variable (i.e., whether bug is killed or not) is a naturally dichotomous variable. Given a continuous variable x and a dichotomous variable y, it is formulated as follows.

$$r_{pb} = \frac{M_1 - M_0}{s_n} \times \sqrt{\frac{n_1 n_0}{n^2}}$$

where $r_p b$ is the point biseral correlation, M_1 and M_0 are the mean of group having value of y equals to 1 and 0, respectively, s_n is the standard deviation of x, n is the total number of data points, and n_1 and n_0 are the number of data points having value 1 and 0, respectively.

To interpret the strength correlation, we use the interpretation given in [7], i.e., $r_{pb}{}^2 \geq 0.81$ means very strong, $0.49 \leq r_{pb}{}^2 < 0.81$ means strong, $0.25 \leq r_{pb}{}^2 < 0.49$ means moderate, $0.09 \leq r_{pb}{}^2 < 0.25$ means weak, and $0.00 < r_{pb}{}^2 < 0.09$ means very weak.

III. Methodology

In this section, we first describe the dataset (subject programs) that we use for this study. We then describe the overall process that we follow for this empirical study, which includes details on data collection as well as its analysis.

A. Subject Programs

We analyse two large systems developed in Java: HTTPClient and Rhino. The first, HTTPClient, is a Java library of components for building client side HTTP services. The second, Rhino, is an open source JavaScript engine, which is usually embedded into Java applications to provide scripting to end users. We selected these two projects as they have different functionalities and are developed by two different organisations i.e., Apache and Mozilla. HTTPClient uses Maven to build the system, whereas Rhino uses Ant. Moreover, both the systems use different bug tracking systems: HTTPClient uses JIRA and Rhino uses Bugzilla. Both the projects use git version control system and are hosted on GitHub[4]. Table I shows the lines of code for both the projects.[5]

TABLE I: Lines of Code of HTTPClient and Rhino

Project	Lines of Code
HTTPClient	122,288
Rhino	116,065

B. Overall Process

Figure 1 shows the overall process, which is divided into three different phases: linking, fixed version analysis and prefix version analysis.

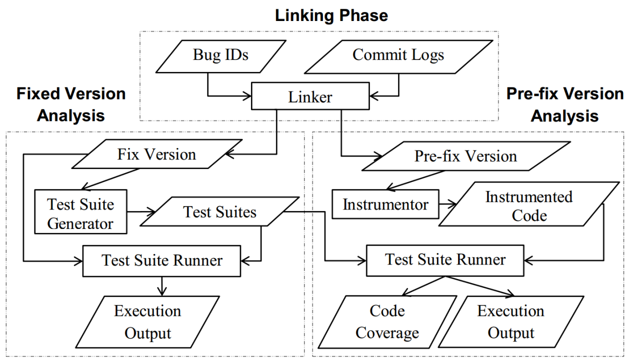

Fig. 1: Overall Process

[4]Located at https://github.com/apache/httpclient for HTTPClient, and https://github.com/mozilla/rhino for Rhino

[5]We count the lines of code of the latest version of these projects.

1) Linking: In the linking phase, we link each bug id to its corresponding commits. We use a dataset from Herzig et al. where they characterize issue reports into 14 different categories. They show that a large number of issue reports are misclassified, i.e., every third issue report marked as a bug is not a bug. We use the issue reports which are manually classified as bugs by Herzig et al. In total, we have 305 bug ids from HTTPClient and 302 bug ids from Rhino. We analyze the commit logs from the git version control systems of the project repositories. Commits logs often contain bug id as part of the commit message. We link the bug reports to their corresponding commits by finding the occurrence of bug id in the commit logs. For each bug id, we collect the fixed version (non-buggy version), i.e., version where the bug was fixed, and the pre-fix version (buggy version), i.e., version right before the bug fixing commit. In total, we find 145 links for HTTPClient and 242 links for Rhino. We ignore bugs which have more than one bug fixing commits since for such cases it is difficult to segregate the non-buggy and buggy version, for which the difference between the two versions is only the buggy behavior and nothing else. Some irrelevant commits might be made in between the bug fixing commits which introduce non-buggy behavioral differences.

2) Fixed Version Analysis: In this phase, we analyse the fixed (non-buggy) version of each bug in the repository. We perform the following steps:

a) Compiling source code: The projects in our study use different build systems, i.e., HTTPClient uses maven whereas Rhino uses ant. Projects using Maven can be built using information stored in the project object model (POM) file, pom.xml. Projects using ant can be built using build.xml, which describes the build process and its dependencies. Although we used maven and ant to automate the build process, still some commits fail to compile due to missing dependencies such as jar files. Since the commits we analyse are spread over a long period, some of the versions failed due to incorrect Java version. We tried to manually resolve such issues and re-compile the source code files. If the version still failed to compile despite our attempt, we omit those bugs from our analysis.

b) Generating Test Suites: We use Randoop to generate test cases. The reason behind using Randoop is to vary the number of test cases to get a range of coverage values and analyse the effectiveness of test suites of varying coverage values to catch the bugs. We use Randoop to generate test cases for 5 minutes on the non-buggy version. Based on our analysis, we observe that Randoop can generate a large number of test cases even when ran for a short duration. We divide each generated test suite into smaller suites of varying sizes to get a range of coverage values. For each test suite that Randoop generates, we create 5 other test suites of size 0.2%, 0.5%, 1%, 5%, and 10% of the original test suite. To create these smaller test suites, we randomly pick test cases from the original test suite. Thus for each bug, we have 6 test suites: the original test suite, and the 5 smaller test suites.

c) Running Test Suites: After the test suites are generated, we run these test cases on the fixed version using junit runner. We observe that some test cases in the test suites produce different outputs when we execute them multiple times. Thus, we repeat the process of running test suites 50 times to account for non-determinism, i.e., test cases which show different behaviours on different executions. If a test case provides different output under the same condition, it is difficult to interpret the output of the test case when we change the condition. Thus, in order to have an unbiased analysis, we filter out non-deterministic test cases from the test suite. We also store the execution output for each deterministic test case at this step which is used to compare with the execution output of the pre-fix version. We remove bugs where there are test failures (i.e., exceptions or assertion failure) and no execution output could be generated.

3) Pre-fix Version Analysis: In this phase, we check out the pre-fix or the buggy version of the code. As we want to measure the code coverage of this version, we use Cobertura[6], which instruments the byte code of the compiled classes. Cobertura stores the information about the instrumented classes in a *.ser file* and is used by Cobertura while running test cases. After the instrumentation step, we use the deterministic test cases generated in the fixed version as we want to observe the behavior of the same test cases on the fixed and pre-fix versions. We run these test cases to get the coverage information as well as the execution outputs. We exclude bugs where test cases fail to compile or run and produce zero coverage. These include cases where the test suite invokes methods that do not exist in the pre-fix version.

At the end of this last phase, we are left with 67 and 92 real bugs from HTTPClient and Rhino. Each of the bug has six test suites of size 0.2%, 0.5%, 1%, 10%, and 100% of the original test suite that Randoop creates. We tried other percentages, e.g., 40%, 50%, etc., but for test suites built with these percentages, their coverage levels were very close to the coverage achieved when we use all test cases in the original test suite. Thus, to generate test suites with varying coverage levels, we only use the following percentages: 0.2%, 0.5%, 1%, 5%, 10%, 100%. Table II shows the average number of test cases (i.e., JUnit methods) for each test suite size across all bugs. Table III shows the average coverage achieved by test suites of different sizes.

TABLE II: Average Test Suite Size

Project	% of Original Test Suite Size					
	0.2	0.5	1	5	10	100
HTTPClient	7.43	15.62	39.13	197.82	396.17	3967.00
Rhino	7.64	16.01	40.10	202.52	405.46	4059.28

TABLE III: Average Coverage

Project	Coverage	% of Original Test Suite Size					
		0.2	0.5	1	5	10	100
HTTPClient	Line	7.5	11.0	17.2	28.0	31.8	37.4
	Branch	2.8	4.4	7.6	14.4	17.2	22.5
Rhino	Line	6.4	8.7	11.6	17.0	19.4	27.1
	Branch	3.0	4.2	5.8	9.0	10.5	16.5

C. Measuring Effectiveness

We measure effectiveness by the ability of a test suite to kill a bug. The underlying assumption of our analysis is: a test suite that runs successfully (i.e., all test cases run successfully) on a non-buggy version and fails on the buggy version (i.e., one of the test cases fails) kills the bug.

[6]http://cobertura.github.io/cobertura/

IV. Results

In this section, we report the results of our empirical study which answer the research questions.

A. RQ1: Size vs Effectiveness

Motivation: One way to improve the effectiveness of a test suite is by increasing the size of the test suite. It might seem that with more test cases the chance for a bug to get killed by one of the test cases is higher. However, can we really improve a test suite effectiveness by simply adding test cases to a test suite? To answer this question, we investigate the correlation between test suite size and test suite effectiveness.

Approach: Using the methodology described in Section III, we have a set of test suites of varying sizes. We also know whether each of these test suites is effective to kill a bug or not. We compute the point biserial correlation described in Section II to check whether there is a correlation between test suite size and test suite effectiveness.

Findings: The null hypothesis is that there is no correlation between test suite size and test suite effectiveness, whereas the alternate hypothesis states that there is a correlation between these two variables. Table IV shows the correlation between the size of a test suite and its effectiveness, i.e., whether a bug is killed by the test suite or not. We can observe that there is a *strong* correlation between test suite size and its effectiveness for HTTPClient (i.e., $0.49 \leq {r_{pb}}^2 < 0.81$). For Rhino, the correlation is *weak* (i.e., $0.09 \leq {r_{pb}}^2 < 0.25$). The p-values for the correlations are statistically significant i.e., < 0.05. Thus, we can reject the null hypothesis and conclude that there is a correlation between test suite size and test suite effectiveness.

TABLE IV: Point Biserial Correlation Values for Test Suite Size and Test Suite Effectiveness

	HTTPClient	Rhino
r_{pb}	0.70	0.37
${r_{pb}}^2$	0.49	0.14
p-value	$4.69e^{-20}$	$4.48e^{-11}$

Test suite size is weakly to strongly correlated with test suite effectiveness.

B. RQ2: Coverage vs Effectiveness

Motivation: Coverage gives information about parts of the code ran by test cases. Intuitively, if a segment of the code is not covered by any test case, it is difficult to find bugs in the code. So, a highly covered code should have lower chances of containing a bug. Several past studies have shown that such a correlation exists when we use seeded faults [2], [8]. We want to examine if this phenomenon stands true when experimenting with real bugs.

Approach: Using the methodology described in Section III, we have a set of test suites of varying levels of coverage. We also know whether each of these test suites is effective to kill a bug or not. We compute the point biserial correlation described in Section II to check whether there is a correlation between test suite coverage and test suite effectiveness.

Findings: The null hypothesis is that there is no correlation between the coverage of a test suite and its effectiveness, whereas the alternate hypothesis states that there is a correlation between these two variables. Table V shows the correlation between the coverage of a test suite and its effectiveness, i.e., whether a bug is killed by the test suite or not. We can observe that there is a *moderate* correlation between (statement and branch) coverage and the effectiveness of a test suite for HTTPClient (i.e., $0.25 \leq {r_{pb}}^2 < 0.49$). For Rhino, the correlation between (statement and branch) coverage and the effectiveness of a test suite is *strong* (i.e., $0.49 \leq {r_{pb}}^2 < 0.81$). The p-values for all the correlations are statistically significant, i.e., < 0.05. Thus, we can reject the null hypothesis and conclude that there is a correlation between coverage and the effectiveness of a test suite even when we use real bugs.

TABLE V: Point Biserial Correlation Values for Test Suite Coverage and Test Suite Effectiveness

	Statement		Branch	
	HTTPClient	Rhino	HTTPClient	Rhino
r_{pb}	0.57	0.77	0.60	0.74
${r_{pb}}^2$	0.33	0.59	0.36	0.55
p-value	$8.71e^{-14}$	$8.59e^{-42}$	$3.55e^{-15}$	$3.67e^{-39}$

Code coverage of a test suite is moderately to strongly correlated to its effectiveness.

V. Discussion

Our study is the first that empirically analyses the correlation between code coverage and test suite effectiveness using real bugs from large systems. Many prior studies either analyze small systems, e.g., [8], [9], or use mutants rather than real bugs, e.g., [1], [2]. Our study finds that test suite effectiveness is weakly to strongly correlated with test suite size, and test suite effectiveness is moderately to strongly correlated with coverage. It supports prior studies by highlighting that developers have more likelihood to catch bugs by adding test cases especially those that can increase coverage.

There are several threats that potentially affect the validity of our study including threats to internal validity, threats to external validity, and threats to construct validity. Threats to internal validity refers to experimenters' biases and errors. We have rechecked our implementation to make sure there are no errors. However, there may still be errors that we do not realize. In the linking process, we link bug reports to commits by looking for the occurrence of bug ids in commit logs. This is a standard step done in many prior studies, e.g., [10]. However, there might cases where developers forget to enter bug ids in the logs of relevant commits. These commits will be missed by our linking process. Also, we have only used test cases that are generated by running Randoop for 5 minutes. We find that even if we run Randoop for 1 hour, although the number of test cases generated is much larger, the additional coverage gained is little (less than 5 percent). Threats to external validity refers to generalizability of our findings. In this preliminary study, we have only experimented on two large software systems. In the future, we plan to reduce this threat by investigating more systems. Threats to construct validity refers to the suitability of our evaluation metrics. We use point biserial correlation metric for our experiment. Point biserial correlation is the recommended metric when we have a variable which is naturally dichotomous [6]. Thus, we believe there is little threat to construct validity.

VI. Related Work

The most related work to ours are the works done by Gopinath et al. [1] and Inozemtseva et al. [2]. Gopinath et al. experimented on hundreds of projects from Github [1]. They calculated test cases' coverage and performed mutation analysis on human generated and automatically generated test cases. Human generated test cases are directly collected from the projects' Github repositories. Automatically generated test cases are produced by running Randoop. They performed cross validation and found that statement coverage is a good indicator of test suite effectiveness. Inozemtseva et al. experimented on five open source systems and generated 31,000 test suites for them [2]. They measured statement, decision, and modified condition coverage and performed mutation testing to measure test suite effectiveness. As their paper title explicitly states, their experiments show that code coverage is not strongly correlated with test suite effectiveness. Different from the above studies, we conducted experiments on real bugs rather than mutants that appear in real large software systems.

Many works have been conducted on code coverage as a measure for test suite quality. Gligoric et al. showed that branch coverage is the best measure for evaluating test quality, but acyclic intra-procedural path coverage might be better when there is a tie [11]. Frankl and Weiss compared branch coverage with def-use coverage and found that def-use is more effective than branch coverage [12]. Gupta et al. compared block, branch, and predicate coverage and found that predicate coverage kills more mutants than other tested coverage criteria [13]. Namin and Andrews showed that test suite size and coverage is correlated with test suite effectiveness [14]. The above studies either analyze small programs (less than 10,000 LOC) [12], [13] and/or only use mutants (i.e., artificially seeded faults) [11], [13], [14].

There are a number of studies that investigate test adequacy of open-source projects. Kochhar et al. investigated 50,000 open-source projects from GitHub to understand the correlation between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams [15]. They extended their study to include characteristics such as number of bugs, number of bug reporters and the programming languages [16]. Moreover, in their latest study they investigated over 300 large open-source projects to measure the adequacy of testing by analysing correlations between code coverage and software metrics such as lines of code, cyclomatic complexity, and number of developers [17]. In this work, we investigate whether code coverage is correlated to test suite quality. However, we perform our empirical study on large software systems rather than small ones, using real bugs rather than mutants.

VII. Conclusion and Future Work

Test cases are important artefacts in any software project as they allow developers to check the reliability of their code and produce reliable software. Code coverage is often used to evaluate the thoroughness and adequacy of testing and to find areas of code not touched by test cases. Past studies have analysed correlation between code coverage and effectiveness of test suite using manually seeded faults. In this work, we conduct a study using *real* bugs from two large systems to evaluate the relationship between test suite size, coverage and effectiveness.

Our study highlights the following results:

1) Test suite size is weakly to strongly correlated with test suite effectiveness.
2) Code coverage is moderately or strongly correlated to the effectiveness of a test suite.

As a future work, we plan to investigate other experiment settings that we did not consider. For example, rather than generating test cases from the fixed version, we can generate test cases from the pre-fix version. Also, we plan to use human-generated tests rather than Randoop-generated tests. Furthermore, we plan to investigate more projects which will reduce the threats to external validity. It would also be interesting to investigate patterns/features of tests that would lead to more effectiveness.

Acknowledgement

We would like to thank Kim Herzig, Sascha Just, and Andreas Zeller for making their issue report datasets publicly available.

References

[1] Rahul Gopinath, Carlos Jensen, and Groce Alex. Code coverage for suite evaluation by developers. In *ICSE*, pages 72–82, 2014.

[2] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE*, pages 435–445, 2014.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[4] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *ICSE*, pages 392–401, 2013.

[5] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[6] James Dean Brown. *Understanding research in second language learning: A teacher's guide to statistics and research design*. Cambridge University Press, 1988.

[7] M. A. Pett. *Nonparametric Statistics for Health Care Research: Statistics for Small Samples and Unusual Distributions*. Sage Publications, Inc., 1997.

[8] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38:235–253, 1996.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[10] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM*, pages 23–32, 2003.

[11] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA*, pages 302–313, 2013.

[12] Phyllis G Frankl and Stewart N Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19:774–787, 1993.

[13] Atul Gupta and Pankaj Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer*, 10:145–160, 2008.

[14] Akbar Siami Namin and James H Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA*, pages 57–68, 2009.

[15] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects-a preliminary study on 50, 000 projects. In *CSMR*, pages 353–356, 2013.

[16] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *QSIC*, pages 103–112, 2013.

[17] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall. An empirical study on the adequacy of testing in open source projects. In *APSEC*, 2014.