Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?

Rainer Niedermayr
University of Stuttgart, CQSE GmbH
Garching b. München, Germany
niedermayr@cqse.eu

Stefan Wagner
University of Stuttgart
Stuttgart, Germany
stefan.wagner@iste.uni-stuttgart.de

ABSTRACT

Mutation testing is a means to assess the effectiveness of a test suite and its outcome is considered more meaningful than code coverage metrics. However, despite several optimizations, mutation testing requires a significant computational effort and has not been widely adopted in industry. Therefore, we study in this paper whether test effectiveness can be approximated using a more light-weight approach. We hypothesize that a test case is more likely to detect faults in methods that are close to the test case on the call stack than in methods that the test case accesses indirectly through many other methods. Based on this hypothesis, we propose the minimal stack distance between test case and method as a new test measure, which expresses how close any test case comes to a given method, and study its correlation with test effectiveness. We conducted an empirical study with 21 open-source projects, which comprise in total 1.8 million LOC, and show that a correlation exists between stack distance and test effectiveness. The correlation reaches a strength up to 0.58. We further show that a classifier using the minimal stack distance along with additional easily computable measures can predict the mutation testing result of a method with 92.9% precision and 93.4% recall. Hence, such a classifier can be taken into consideration as a light-weight alternative to mutation testing or as a preceding, less costly step to that.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

software testing • test effectiveness • test metrics • minimal stack distance • mutation test prediction

ACM Reference Format:
Rainer Niedermayr and Stefan Wagner. 2019. Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?. In *Evaluation and Assessment in Software Engineering (EASE '19), April 15–17, 2019, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3319008.3319021

EASE '19, April 15–17, 2019, Copenhagen, Denmark

1 INTRODUCTION

Automated software tests are an important means for quality assurance in software projects and are used to reveal faults and prevent regressions in software applications. Different measures to evaluate test suites have been proposed. Most common are code coverage metrics [19, 48] expressing which portion of the application code is executed by test cases. They can be computed at different levels, for example, as line coverage, branch coverage, or decision coverage [9]. However, since code coverage metrics measure test completeness and do not assess oracle quality, they are not necessarily suitable for expressing the test effectiveness of a test suite [3, 20, 33]. More advanced approaches take data-flow criteria into account [39] and measure which portion of the covered statements is checked in assertions [41].

Another established, powerful technique to evaluate test suites is mutation testing [25]. The general idea behind mutation testing is to generate mutants by seeding faults into the code of a program and check whether the tests can kill (detect) these faults. Hence, compared to code coverage metrics, this technique takes oracle quality into account and can provide more meaningful results. However, mutation testing is—despite several optimization techniques—computationally complex due to the effort needed for generating and testing a large number of mutants. Despite its effectiveness, there are no indications that mutation testing is widely adopted as a test efficacy criterion in practice [21, 25].

Since mutation testing can be expensive and code coverage is not necessarily meaningful enough for assessing test suites, we study in this paper whether test effectiveness can be approximated using a more light-weight approach. We hypothesize that a test case that directly invokes a method is more likely to detect faults in that method than another test case that accesses the method indirectly through many others. Therefore, we propose a measure called *minimal stack distance*, which expresses how close any test case comes to a given method, and study whether methods with a high minimal stack distance value are more likely to be ineffectively tested. For that, we conduct a mutation analysis using the Descartes operator [46] and assess whether methods that contain surviving mutants exhibit a higher minimal stack distance than the remaining methods. Furthermore, we train a classifier using stack distance values and further measures, which can be collected in a single execution of a test suite, and evaluate the classifier's performance in predicting mutation testing results.

Research goal: We aim at reducing the effort to identify ineffectively tested code. In this paper, we investigate how well the stack distance measure correlates with and can be used to predict test effectiveness. This would allow us to use it as alternative to mutation testing.

Contributions: This paper makes two contributions: First, we propose and study the minimal stack distance measure, which characterizes the proximity of a method to any of its test cases. Second, we evaluate a machine-learning classifier based on method test-case characteristics and show that classifiers to predict mutation testing results can come into question as an alternative to mutation testing or as a preceding, less costly step to that. For example, this could allow the use in continuous integration where mutation testing would take too long or is not applicable for other reasons.

The remainder of the present paper is organized as follows. Section 2 discusses related work. Section 3 defines relevant terms. Afterwards, Section 4 describes the approach to compute the stack distance measure. Section 5 presents design and results of the empirical study. Then, Section 6 discusses the study's results and implications, and Section 7 explains threats to validity. Finally, Section 8 summarizes the main findings and sketches future work. Data to replicate the study is available at [34].

2 RELATED WORK

Mutation testing was first proposed by Lipton [29] in the 1970s and formalized by DeMillo et al. [11]. It has since then been extensively studied [25, 38, 45]. In general, mutation testing is computationally complex; to address this downside, researchers have suggested several approaches to reducing the cost of mutation analysis. Offutt et al. [37] classified these approaches as *do fewer, do smarter,* and *do faster. Do fewer* approaches comprise the use of a smaller, representative set of mutation operators [35, 36, 42], sampling of mutants [1], mutants clustering [23], and higher order mutation, in which multiple mutation operators are applied at once [24]. The most prominent *do smarter* approach is weak mutation, in which a mutant is immediately evaluated after its execution point instead of checking it at the end of a test execution [18, 25]. *Do faster* approaches comprise further run-time optimization techniques to speed up the generation and execution of mutants (e.g., bytecode mutants [30, 40], aspect-oriented mutation [6], or parallel mutation testing [12]).

In our work, we study whether measures describing the relationship between methods and test cases can uncover ineffectively tested methods representing surviving mutants. Hence, we propose an approach to predict the mutation testing result of a method in a light-weight way without the need for executing mutation testing.

Namin et al. [42] used linear models to predict the overall mutation score, and Jalbert et al. [22] also predicted that score using machine learning models. However, both did not perform predictions on individual methods. Strug et al. [43, 44] calculated the structural similarity of mutants, predicted based on results of similar mutations whether a given test would detect a mutant or not, and thereby reduced the number of mutants to be executed. However, their approach still requires a mutation analysis of a subset of mutants. The most related work to ours is from Zhang et al. [47], who predicted the mutation testing result of individual mutations and achieved promising results. They also included mutations that are not covered by any test case and hence cannot be killed. In contrast to the work of Zhang et al., we predict the mutation testing result of a method and not of single mutations, exclude methods that cannot be killed since they are not covered, and include the proposed minimal stack distance measure in the prediction model.

Stack distance as a measure was first defined and used by Mattson et al. to evaluate storage hierarchies [31]. Caşcaval et al. used it to estimate cache misses [7]. Barford et al. used it for web servers to measure the likelihood that a requested file will be requested again in the near future [5]. In this paper, we define stack distance in the context of testing to characterize the proximity between test cases and methods.

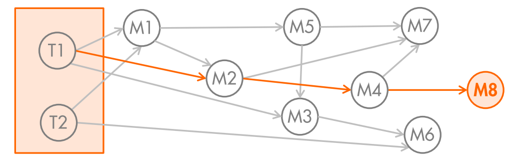

Figure 1: The minimal stack distance of method *M8* is 3. No test case can access *M8* through fewer method invocations.

3 DEFINITIONS

We define the *minimal stack distance between a method m and a test case t* as the length of the shortest path from t to m.[1] Hence, the value is one for a method that is directly invoked by a given test case and, for example, two for a method that is indirectly invoked by a given test case through one other method.

We define the *minimal stack distance of a method m* as the shortest distance between m and any of its covering test cases $T(m)$. It corresponds to the minimal distance on the call stack between the method m and all test cases. Figure 1 illustrates an example.

We call a method *covered* if it is executed by at least one test case. The *mutation testing result* of a covered method can either take the value *ineffectively tested* or *effectively tested*. We consider a covered, non-empty method as *ineffectively tested* if its whole logic can be removed without causing any test case to fail. Such ineffectively tested methods are also known as *pseudo-tested* methods [33, 46]. The idea behind pseudo-testedness is that if no single test case can detect such an extreme transformation, test cases will not be able to detect more subtle mutations. Pseudo-tested methods can be detected with the Descartes mutation operator, which works as follows [46]. For void methods, the operator removes the whole method body. For methods with a return type, depending on the type, one or two mutants are created, which replace the method body with a statement returning a value satisfying the declared return type. Table 1 presents the return values per type. When two mutants are created, a method is only considered pseudo-tested if both mutants cannot be killed; hence, the use of two mutants avoids that equivalent mutants influence the mutation testing result of a method.

We further use common mutation testing terms as defined in literature [25]: A *mutation operator* is a transformation rule that generates a *mutant* by applying syntactical changes to the original program. A mutant is said to be *killed* if at least one test case of the test suite fails due to the changes; otherwise it is said to have

[1] In this paper, we define and apply minimal stack distance based on *methods*. However, the definitions are also applicable to *functions* in non-object-oriented programming languages.

Table 1: Return values of the Descartes operator.

Return Type Class	Mutant 1	Mutant 2
void	*(void)*	*(not created)*
boolean	`false`	`true`
byte, short, int, long	`0`	`1`
float, double	`0.0`	`0.1`
char	`' '`	`'A'`
string	`""`	`"A"`
T[]	`new T[]{}`	*(not created)*
reference type	`null`	*(not created)*

survived. An *equivalent mutant* is—despite syntactical changes—semantically equivalent to the original program and can therefore not be killed.

4 COMPUTATION OF MINIMAL STACK DISTANCE

In the following, we describe the computation of the minimal stack distance for Java applications; nonetheless, this measure is applicable to other programming languages as well. The steps to compute the minimal stack distance comprise the instrumentation of the code, the replacement of Java's `Thread` class, and the recording of the method invocations during the test execution. Figure 2 presents an overview of the computation.

1) Instrumentation: We instrument each method of the source code so that it notifies our stack-recorder class when a method is entered and exited. To instrument a method, we introduce a new try-finally block and move the original code into the try block. We then insert a statement before the try block, which calls our recorder class with the signature of the considered method. Next, we insert a further statement into the newly created finally block, which informs the recorder that the method invocation needs to be removed from the current stack. The finally block is always invoked when the method is left (even if an exception is raised or propagated).

To conduct the code instrumentation, we developed a Maven-plugin, which operates at the byte-code level and uses the ASM[2] library. The decompiled source code of an instrumented method might look as follows:

```
public int getSize() {
  InvocationLogger.push(
      "org.SampleClass.getSize()");
  try {
    /* BEGIN ORIGINAL CODE */
    return this.size;
    /* END ORIGINAL CODE */
  } finally {
    InvocationLogger.pop(
       "org.SampleClass.getSize()");
  }
}
```

2) Thread class replacement: To achieve a thread-aware computation of the minimal stack distance, we need to be aware of the current stack height of each thread and know which thread was started by which other thread. For that, we need to be notified when a new thread is started. Since Java's `Thread` class does not provide the possibility to register listeners, we took the original code from the JDK and adjusted it so that our stack-recorder class gets informed about the start of a new thread. We compiled the modified thread class and put it into the "endorsed" folder of the JDK. The replacement of the thread class does not influence test results.

3) Recording: Finally, we need to execute the test suite and record the distances between test cases and methods. We use Maven's Surefire plugin for the execution of unit tests and Failsafe plugin for integration tests and register our stack-recorder class as test listener in these plugins. Hence, the recorder will be notified when a new test case execution begins and can assign all subsequent method invocations to that test case. When a test case execution starts and an instrumented method is entered, the method's signature is pushed onto the recorder's stack for the current thread. Then, the stack's height is counted and, if appropriate, the distance from the executed test case to the start of the current thread is added. If the resulting distance constitutes a new minimum for a given method test-case pair, the pair's minimal stack distance value is updated. When an instrumented method is left, its signature is taken down from the stack of the appropriate thread.

Note that if a method is invoked recursively, the height of the stack increases with each invocation; however, we are only interested in the *minimal* stack distance of each method test-case pair.

In short, the recorder class holds the so far minimal stack distance of each executed method test-case pair, the method invocations on the stack of each thread, and the relations between the threads. At the end of each test case execution, the minimal stack distance values are persisted.

Note that another imaginable approach that computes the stack height by requesting the current thread to dump its stack trace (as done when creating exceptions) is not fast enough to be viable for doing the computation in test executions.

Limitations are as follows: We applied the instrumentation to all methods except constructors. We excluded constructors, because it is tricky to instrument a constructor in a way so that its beginning is correctly intercepted, because a constructor's very first statement unavoidably delegates to another constructor or a super constructor such that the code there gets executed first. Consequently, constructor invocations will not be counted when computing the stack distance; notwithstanding the above, methods invoked by constructors are still considered. Furthermore, external libraries are not instrumented; therefore, method invocations in external

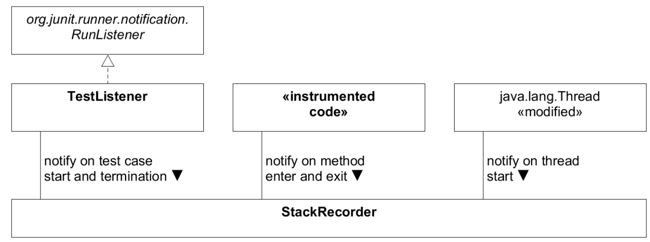

Figure 2: Overview of the stack distance computation.

[2] http://asm.ow2.io/

libraries are not counted. The consequence of both limitations is that the computed stack distance will in some cases be slightly lower than the actual distance. Hence, the computed minimal stack distances should be considered as a lower bound.

5 EMPIRICAL STUDY

This section reports on the design and results of the empirical study that we conducted to investigate the influence of the minimal stack distance between test case and method on test effectiveness. We further examined how well the mutation testing result of a method can be predicted using this measure.

5.1 Research Questions

We investigate the following research questions:

RQ 1: Are methods with a higher stack distance to the test cases more likely to be ineffectively tested? With this research question, we want to find out whether the minimal stack distance of a method is correlated with the property how well a method is tested. We hypothesize that a test case that never comes close to a given method is not effective in detecting faults in that method. Consequently, we expect a method tested only by distant test cases to be less effectively tested. In other words, we hypothesize that methods with a high minimal stack distance are more likely to contain surviving mutants. The answer to this question helps determining whether stack distance can be a useful predictor for test effectiveness.

RQ 2: How well can the mutation testing result of a method be predicted using test-relationship measures? Since mutation testing is costly, we want to find out whether a more light-weight approach can approximate results gained from mutation analysis. We are interested in predicting the mutation testing result of a method based on measures characterizing relationships between methods and test cases. If such a prediction approach works well, it could be used as an alternative to mutation testing or as a preceding, less costly step to that.

5.2 Study Objects

We selected study objects from GitHub[3] based on the following criteria: The projects need to be written in Java, contain test cases designed for the JUnit test framework, and use Maven as build system. We manually selected five Apache projects (Commons Geometry, Commons Imaging, Commons Lang, Commons Math, Commons Statistics), and JFreechart, which are popular open-source projects used in several empirical test studies (e.g., in [17, 20, 26]). We selected additional study objects that satisfy the previously mentioned criteria by searching GitHub for recently updated projects with at least five forks (to require a certain popularity). We excluded a project if it was not possible to build it (e.g., due to compilation problems or unresolvable dependencies), if more than 5% of the test cases failed in a local execution of the original test suite, or if the mutation analysis was not successful (e.g., due to special test runners or class loading mechanisms).

[3]https://github.com

The selected study objects are from different domains and contain both single- and multi-module projects. Their characteristics are presented in Table 2. *LOC* (lines of code) refers to the application code (i.e., code without test and sample code) and was measured with Teamscale [16]. *# Tests* refers to the number of test cases as reported by Maven. *Line* and *branch coverage* were computed with JaCoCo[4]. The largest project, biojava, consists of 240.6 k LOC. Commons Math contains with 5,254 the most test cases. The line coverage of the projects ranges between 28.0% and 95.0%.

5.3 Study Design

RQ 1: We hypothesize that the higher the minimal stack distance of a method is to any test case, the less likely the method is effectively tested. To test this hypothesis, we analyze whether a correlation exists between a method's minimal stack distance to any test case and its mutation testing result (i.e., whether a method is ineffectively tested by *all* test cases or not). For that, we compute for each project the Spearman rank correlation coefficient, which expresses the strength of this relationship (between -1 and $+1$), and the p-value. We use a significance level of 0.05. Moreover, we present plots illustrating the proportion of ineffectively tested methods per minimal stack distance value.

RQ 2: To answer this research question, in which we train and evaluate a classifier to predict mutation testing results, we collect further measures besides stack distance for each covered method. We chose the following method measures because they can easily be computed during a single execution of a test suite:

- Line count: number of coverable lines of code in the method
- Branch count: number of branches
- Line coverage: proportion of covered lines out of coverable lines
- Branch coverage: proportion of covered branches out of coverable branches (100% for covered methods without branches)
- Number of covering test cases: number of test cases that execute the method
- Scope of covering test cases: minimum number of covered methods of any of the method's covering test cases
- Maximum invocation count: maximum number of invocations of the method during the execution of any covering test case
- Return type of the method: void, boolean, numeric, string, array, reference to object

For each project, we train one machine-learning classifier to predict the mutation testing result of a method with respect to all covering test cases, and one to predict the mutation testing result of a method test-case pair.

We evaluate the performance of the models with respect to within-project and cross-project predictions. Within-project evaluations show how well predictions work when models are trained on a data-subset of the same project, cross-project evaluations indicate how well models can be generalized to conduct predictions in other projects. For within-project predictions, we apply repeated ten-fold cross-validation [27]. For cross-project predictions, we test each project with a model that is trained on the respective remaining projects.

[4]https://www.eclemma.org/jacoco/

Table 2: Study objects.

Name ↓	Purpose	LOC	# Tests	Line Cov.	Branch Cov.	Git Revision
Apache Commons Geometry	geometric utilities	19.4 k	643	76.9%	70.7%	`be34ad93`
Apache Commons Imaging	image library	48.4 k	575	71.3%	58.9%	`eb98398b`
Apache Commons Lang	utility classes for Java	77.0 k	4,053	95.0%	91.1%	`1f0dfc31`
Apache Commons Math	mathematics library	186.3 k	5,254	89.8%	84.8%	`eafb16c7`
Apache Commons Statistics	statistics library	6.1 k	358	91.5%	87.6%	`aa5cbad1`
biojava	biological data processing	240.6 k	1,181	40.5%	38.5%	`523c78e1`
bitcoinj	Java Bitcoin library	59.1 k	5,222	67.5%	61.3%	`911f6d49`
geometry-api-java	spatial data processing	87.0 k	408	71.6%	59.4%	`3704c220`
google-gson	JSON serialization	14.8 k	1,039	84.4%	79.2%	`57085d62`
Google HTTP Java Client	HTTP client library	30.1 k	635	54.9%	58.8%	`df0e9f2a`
graphhopper	route planning library and server	60.5 k	1,680	65.4%	60.9%	`e954f008`
jackson-databind	databinding for JSON data	103.0 k	2,159	77.8%	70.7%	`bf604125`
javaparser	parser and AST for Java	118.4 k	1,284	59.8%	48.1%	`1cca4c46`
JFreechart	chart library	222.8 k	2,175	55.5%	46.4%	`39dfee3c`
jsoup	HTML and CSS parser	18.2 k	671	81.4%	77.8%	`220b7714`
openwayback	web wayback machine	66.8 k	320	28.0%	26.8%	`680fba15`
pdfbox	PDF document manipulation	227.6 k	1,587	49.7%	43.3%	`d9930344`
scifio	scientific image format IO	79.4 k	1,019	37.1%	19.3%	`281e7ce2`
traccar	server for GPS tracking	59.6 k	310	56.4%	49.0%	`6d259427`
urban-airship	library for marketing platform	37.9 k	706	79.3%	46.0%	`98edb3ca`
vectorz	fast vector mathematics	61.9 k	456	61.1%	63.8%	`a05c69d8`

Table 3: Example of a full mutation matrix.

Method	Test Case	Mutation Testing Result
m_1	t_1	ineffectively tested
m_1	t_2	effectively tested
m_2	t_2	ineffectively tested

We measure model performance by computing precision, recall, and F-score. Following Zhang et al. [47], we predict both outcomes (ineffectively and effectively tested) and use the weighted average of the performance metrics (i.e., "each metric is weighted according to the number of instances with the particular class label"). In addition, we report the performance of the outcome *ineffectively tested*, because methods with this outcome represent the minority class and are therefore more difficult to predict.

Furthermore, we exemplary show the prediction model's computed variable importances for one project.

5.4 Data Collection and Processing

To collect data for the study, we first executed the test suite of each study object and recorded the minimal stack distance of each method test-case pair. The recording of the stack distance was carried out as defined in Section 3 and described in Section 4. Note that we were working on the existing test suites of the projects; we did not generate test cases.

Second, we conducted a mutation analysis for each study object. For that, we used *Pitest* (PIT) [10] in version 1.4.0 with the *pit-mp* extension to support multi-module projects. Pitest is a well-known mutation testing tool for Java applications and has been used in several studies (e.g., [2, 13, 14]). As performance optimization, Pitest aborts the analysis of a mutant after the mutant is first killed by a test case. However, for this study, we need a full mutation matrix, which contains the result (killed or survived) of each mutant for each covering test case. Therefore, we adjusted Pitest to compute a full mutation matrix as proposed by [2]. Table 3 presents an example of such a matrix.

To gain further insights, we made an additional adjustment to Pitest and recorded for each killed mutation by what event it was killed. Hence, we know for a mutation whether it is detected by a test case because of a failing assertion (`AssertionError`) or because of another implicit exception being thrown (e.g., `NullPointerException`, or `ArithmeticException` due to a division by zero).

We used Pitest with the *Descartes* plugin [46], which implements the mutation operator to uncover pseudo-tested methods (see Section 3). More details on the mutation operator can be found in [33].

We excluded empty methods and methods solely returning `null` from the analysis because their mutation would result in an equivalent mutant. We also excluded `hashCode` methods because we are convinced that mutation testing is not suitable for assessing their testing state.[5] We further excluded constructors because, as described in the limitations of the stack distance computation in Section 4, we cannot compute reliable stack distance values of these special methods. Moreover, we excluded generated code, which was present for example in bitcoinj, because the code is re-generated during the build process and not designed to be tested.

[5] As long as a `hashCode` method considers no additional fields for computing an object's hash value, it still fulfills its contract even if fewer fields are considered or another computation formula is used. The used mutation operator does not introduce additional field accesses.

Table 4: Overview of the mutation analysis results.

Project	# ineffectively tested methods	% ineffectively tested out of all *covered* methods ↓
SCIFIO	154	32.0%
PDFBOX	829	26.3%
BIOJAVA	1147	24.4%
TRACCAR	193	22.4%
Commons Imag	244	21.4%
OPENWAYBACK	166	18.6%
JFreechart	754	17.7%
Google HTTP	145	16.5%
JAVAPARSER	293	14.0%
GRAPHHOPPER	252	11.5%
GEOMETRY API	224	9.9%
VECTORZ	339	8.0%
JACKSON-DB	307	7.8%
BITCOINJ	77	4.7%
JSOUP	37	4.4%
URBAN-AIRSHIP	78	3.5%
Commons Geom	20	2.8%
GSON	15	2.8%
Commons Math	129	2.7%
Commons Stat	7	2.6%
Commons Lang	43	1.7%
median	*166*	*9.9%*

For RQ 2, we collected further measures to enhance the prediction model. We used *JaCoCo* to compute a method's number of lines and branches as well as line and branch coverage values. The number of covering test cases per method and their scope was computed based on the full mutation matrix. The method's invocation count during a test execution was collected alongside the stack distance recording. Finally, the return type of a method was deduced from the mutation testing output.

We used the statistical software *R* to process data. We trained and evaluated prediction models with R's *caret* package [28]. We chose *Random Forest* as machine-learning algorithm because preliminary experiments on our datasets revealed that it achieved the best performance. *adaboost* achieved an almost equal performance, but was about eleven times slower. Zhang et al. also used Random Forest for their predictions [47].

5.5 Results

This section presents the results to the research questions. Data to reproduce the results are available at [34].

Before addressing the research questions, we present in Table 4 the absolute and relative number of ineffectively tested methods of each project as computed in the mutation analysis. Depending on the project, between 1.7% and 32.0% of the *covered* methods are ineffectively tested methods. According to these measurements, methods in GSON and four of the Apache projects are especially well tested compared to the other projects. In contrast, the results of SCIFIO, PDFBOX, and BIOJAVA are below average.

Table 5: RQ 1: Spearman's correlation coefficient for a method's mutation result and its minimal stack distance. Absolute coefficient values ≥ 0.2 and p-values < 0.05 are highlighted.

Project	coefficient ↓	p-value
JFreechart	**+0.58**	<0.001
SCIFIO	**+0.48**	<0.001
JAVAPARSER	**+0.41**	<0.001
Commons Stat	**+0.35**	<0.001
TRACCAR	**+0.33**	<0.001
PDFBOX	**+0.31**	<0.001
BIOJAVA	**+0.29**	<0.001
GRAPHHOPPER	**+0.24**	<0.001
Commons Lang	**+0.21**	<0.001
BITCOINJ	**+0.20**	<0.001
JACKSON-DB	+0.18	<0.001
JSOUP	+0.18	<0.001
Commons Geom	+0.17	<0.001
Commons Imag	+0.16	<0.001
GEOMETRY API	+0.15	<0.001
OPENWAYBACK	+0.14	<0.001
GSON	+0.13	0.003
URBAN-AIRSHIP	+0.11	<0.001
Commons Math	+0.08	<0.001
VECTORZ	+0.07	<0.001
Google HTTP	-0.17	<0.001

RQ 1: Are methods with a higher stack distance to the test cases more likely to be ineffectively tested? Table 5 shows the results of the Spearman correlation test between a method's minimal stack distance and mutation testing result.

We observe that a statistically significant correlation exists in all 21 projects (p-value < 0.05). The positive correlation coefficients indicate that the proportion of ineffectively tested methods increases with increasing stack distance values. The strongest correlation is achieved in the project JFreechart with a correlation coefficient of 0.58. When looking at this project's test code, it was striking that the test cases contain many assertions. A moderate correlation with a coefficient between 0.3 and 0.5 is present in five further projects. A weak correlation is present in the remaining projects. In the project Google HTTP a weak negative correlation is observed; however, in this project, the minimal stack distance does not exceed the value 2 in 81% of the methods.

The red line in Figure 3 presents the proportion of ineffectively tested methods per minimal stack distance value. In the project JFreechart, more than 50% of the methods with a minimal stack distance higher than 3 are ineffectively tested.

The illustration in Figure 4 indicates that the correlation between a method's minimal stack distance and its mutation testing result is stronger in larger projects with a high proportion of ineffectively tested methods. (The correlation between the project's correlation coefficient and these two project characteristics is each 0.4.)

> Methods with a higher minimal stack distance to covering test cases are more likely to be ineffectively tested.

Figure 3: RQ 1: The charts present the — proportion of ineffectively tested methods as red line and the proportion of methods per minimal stack distance value as gray bars. The hypothesis is that the proportion of ineffectively tested methods increases with increasing minimal stack distance values. The x-axis is cropped when the proportion of methods per distance value falls below 0.5%.

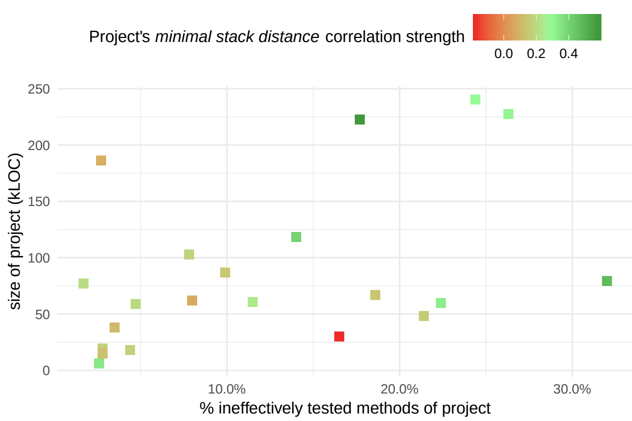

Figure 4: The projects’ proportion of ineffectively tested methods (x-axis), project size in kLOC (y-axis), and the strength of the correlation between a method’s minimal stack distance and its mutation testing result from Table 5 (color).

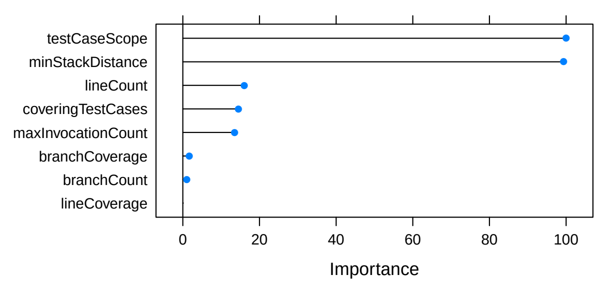

Figure 5: RQ 2: Variable importance of JFreeChart’s prediction model (scaled to one).

RQ 2: How well can the mutation testing result of a method be predicted using test-relationship measures?

Table 6 presents the classifier’s precision, recall, and F-score of the within-project prediction of the mutation testing result of a method. As described in Section 5.3, the performance measures constitute the weighted average of the outcomes *ineffectively* and *effectively tested*. Median precision is 92.9%, and median recall is 93.4%. When conducting cross-project prediction for the same scenario, median precision and recall deteriorate to 85.6% resp. 88.1%.

Ineffectively tested methods represent the minority class and are therefore more difficult to predict. Table 7 shows the within-project prediction performance for identifying ineffectively tested methods. Median precision of this outcome is 70.7% and median recall is 34.3%. In the best case, 96.6% precision and 100.0% recall are still achieved (Commons Stat).

Table 6: RQ 2: Performance when predicting a *method's mutation result.*

Project	Precision	Recall	F-score ↓
Commons Stat	99.9%	99.9%	99.9%
Commons Lang	98.8%	98.9%	98.7%
gson	97.5%	97.7%	97.1%
Commons Math	96.7%	97.5%	96.7%
Commons Geom	96.2%	97.2%	96.4%
urban-airship	96.3%	96.9%	96.3%
Google HTTP	95.1%	95.1%	94.9%
jsoup	94.1%	95.6%	94.3%
bitcoinj	93.7%	95.3%	94.0%
JFreechart	93.1%	93.4%	93.1%
javaparser	92.9%	93.2%	92.8%
vectorz	92.5%	93.5%	92.4%
jackson-db	91.5%	93.0%	91.7%
graphhopper	89.5%	90.8%	89.3%
geometry API	86.6%	90.0%	87.1%
traccar	86.8%	87.1%	86.9%
Commons Imag	87.2%	87.7%	86.8%
biojava	85.1%	85.7%	85.1%
pdfbox	84.1%	84.7%	83.8%
openwayback	81.3%	83.5%	81.4%
scifio	78.7%	79.0%	78.8%
median	*92.9%*	*93.4%*	*92.8%*

Table 7: RQ 2: Performance when predicting *ineffectively tested methods.*

Project	Precision	Recall	F-score ↓
Commons Stat	96.6%	100.0%	98.2%
Google HTTP	94.6%	74.8%	83.5%
JFreechart	87.0%	73.4%	79.6%
javaparser	84.1%	63.4%	72.3%
traccar	72.6%	68.0%	70.2%
biojava	76.4%	59.9%	67.1%
pdfbox	78.4%	57.6%	66.4%
scifio	68.5%	63.8%	66.1%
Commons Imag	81.1%	55.5%	65.9%
Commons Lang	85.5%	41.3%	55.7%
graphhopper	70.6%	34.3%	46.2%
vectorz	70.7%	32.5%	44.5%
openwayback	60.7%	32.5%	42.4%
urban-airship	64.2%	27.6%	38.6%
jackson-db	61.4%	26.6%	37.1%
gson	87.5%	23.3%	36.8%
Commons Math	60.3%	15.9%	25.2%
Commons Geom	50.0%	15.0%	23.1%
bitcoinj	50.0%	13.0%	20.6%
jsoup	51.5%	11.5%	18.8%
geometry API	46.9%	11.0%	17.9%
median	*70.7%*	*34.3%*	*46.2%*

Table 8: RQ 2: Performance when predicting the mutation result *of a method test-case pair.*

Project	Precision	Recall	F-score ↓
scifio	92.8%	92.8%	92.8%
Commons Stat	92.1%	92.4%	91.7%
Commons Geom	90.8%	91.2%	90.4%
javaparser	90.1%	90.2%	90.1%
urban-airship	89.1%	90.2%	88.7%
Google HTTP	88.4%	88.6%	88.2%
gson	87.9%	88.0%	87.9%
Commons Lang	87.5%	87.8%	86.8%
JFreechart	86.5%	86.5%	86.4%
bitcoinj	86.1%	86.1%	86.1%
Commons Math	85.2%	85.7%	85.1%
traccar	85.1%	85.0%	85.1%
vectorz	85.4%	86.6%	84.9%
jsoup	84.4%	84.9%	84.1%
pdfbox	83.8%	83.8%	83.8%
Commons Imag	82.5%	82.7%	82.2%
biojava	81.7%	81.7%	81.5%
openwayback	80.9%	80.8%	80.8%
graphhopper	80.6%	80.7%	80.5%
geometry API	77.6%	78.1%	77.4%
jackson-db	72.4%	72.4%	72.4%
median	*85.4%*	*86.1%*	*85.1%*

Figure 5 exemplary presents the variable importance of JFreechart's within-project prediction model. The figure shows that the minimal stack distance and the minimal scope value of a method's covering test cases (the scope of a test case expresses how many methods it covers) are the most important variables for the prediction model.

Cross-project prediction for identifying ineffectively tested methods only achieves a poor performance. Even when applying the over-sampling technique *SMOTE*[6] to pre-process training sets, median precision is only 19.2% and median recall is 43.2%. Hence, cross-project prediction is not well suited for uncovering ineffectively tested methods.

> The mutation testing result of a method can on average be predicted with 92.9% precision and 93.4% recall. Cross-project prediction is more challenging and achieves a weaker performance.

The above results concern the prediction of a method's mutation testing result with respect to all test cases. For other use cases, e.g., for enhancing test case prioritization with test effectiveness information, it can also be useful to predict the mutation testing result of a method test-case *pair.* Table 8 presents the within-project performance when predicting the mutation testing result of a method test-case pair. In this scenario, median precision and recall are 84.8% resp. 85.3%. When focusing on the outcome *ineffectively tested,* median precision and recall still achieve 82.4% resp. 71.7%.

[6]Synthetic Minority Over-Sampling Technique [8]

Table 9: Duration of analyses (in hours) and slowdown factor based on the normal test suite execution.

Project	Test Suite Execution	Test Suite Execution + Stack Dist. Recording	Mutation Analysis with Early Abort	Mutation Analysis with Full Matrix
BIOJAVA	00:27:45	01:31:00	23:00:00	46:49:00
	(1.0)	*(3.3)*	*(49.7)*	*(101.2)*
BITCOINJ	00:01:40	00:02:45	00:43:26	03:36:00
	(1.0)	*(1.7)*	*(26.1)*	*(129.6)*
JFREECHART	00:00:13	00:00:17	00:09:07	00:13:38
	(1.0)	*(1.3)*	*(42.1)*	*(63.0)*
PDFBOX	00:01:38	00:07:56	02:33:00	05:14:00
	(1.0)	*(4.9)*	*(93.7)*	*(192.0)*

Hence, the prediction achieves promising results when working on method test-case pairs. A reason for this is that, unlike when predicting the result of a method with respect to all test cases, test case metrics are not aggregated.

> Ineffectively tested method test-case pairs can be predicted with 82.4% precision and 71.7% recall on average.

Zhang et al. [47] achieved precision and recall values of around 90% (depending on project and scenario). They only present performance measures aggregated of both outcomes. Although an in-depth comparison with their results does not seem sensible—because they predicted for different mutation operators, used other metrics, and included methods not covered by any test—we can still say that the prediction performance is roughly comparable.

6 DISCUSSION

The study's results show that the correlation between a method's minimal stack distance and its mutation testing result is moderate to strong in six projects and present in further projects to a lower degree. In general, the correlation is stronger in larger projects (JFREECHART, BIOJAVA, PDFBOX), which also exhibit higher minimal stack distance values. In large, multi-module projects some methods are only tested by integration tests, which usually have a higher distance to many of the covered methods than a unit test does. In such projects, the minimal stack distance can provide valuable insights about the testing state of methods and thereby provide an additional value to coverage information.

The evaluation of the prediction models shows that machine learning models can successfully predict the mutation testing result of a method. Hence, such models can be considered as a light-weight alternative to mutation testing. To point out possible time savings, Table 9 presents the duration of different analyses exemplarily of four projects. The current—not yet performance-optimized—implementation for recording the minimal stack distance has an influence on the duration of the test execution. It slows the execution down by a low but perceptible single-digit factor. Nonetheless, a prediction model using this metric can achieve significant savings compared to the execution of a mutation analysis. The analysis with the state-of-the-art mutation testing tool Pitest takes about 50–200 times as long as a single execution of the corresponding test suite. In the largest project (BIOJAVA), the computation of a full mutation matrix took more than 46 hours (101 times the duration of the test execution) and an analysis that stops assessing a mutant after having found the first killing test case still needed 23 hours. Consequently, such prediction models can also be taken into consideration in projects in which a mutation analysis is not applicable due to a long duration.

7 THREATS TO VALIDITY

We separate the threat to validity into internal and external threats.

The computation of the stack distance is a threat to internal validity. Although we developed the computation logic with great care, the implementation could contain faults that affect the outcome. To mitigate this threat, we verified computed values of different code samples and developed automated tests to check the implementation. In addition, the source code of our tool can be inspected on GitHub [32].

The same applies to the conducted extension of the Pitest mutation testing tool to enable computing a full mutation matrix. To mitigate this threat, we created a pull request, which was carefully reviewed and merged by the head developer of Pitest [4].

Some of the generated mutants may be equivalent mutants, which differ only syntactically but not semantically from the original source code, and, hence, cannot be killed [15]. Therefore, some of the mutants that were regarded as surviving could be equivalent mutants and affect the results. Due to the design of the mutation operator (cf. Section 3) and the exclusion of empty methods and methods returning `null`, hardly any equivalent mutants are generated [33]. A manual review on a sample confirmed this observation.

Although we selected 21 study objects with different characteristics, the selection of the projects poses a threat to external validity. Since we chose only open-source projects that use Maven as build system and in which nearly all tests succeed, well-engineered projects with mature test suites may be over-represented in our sample. Hence, future work is necessary to validate whether the results are generalizable for Java projects and projects in other programming languages.

8 CONCLUSION

In this paper, we proposed and studied the minimal stack distance measure, which describes the proximity of a method to any of its test cases. Our results indicate that a correlation exists between this measure and a property indicating whether a method is ineffectively tested (pseudo-tested). Classifiers that predict the mutation testing result of a method achieve a median precision of 92.9% and recall of 93.4%. The measures needed for such a classifier can be computed in a single test suite execution, while mutation testing may take—depending on the size of an application—several hours or days. Therefore, we suggest considering such classifiers as a light-weight alternative to mutation testing or as a preceding, less costly step to that. In particular, the classifiers can be a reasonable alternative in continuous integration. Furthermore, they can be useful for projects in which a mutation analysis is not applicable (due to the analysis duration or class loading issues).

For future work, we plan to investigate more measures, such as, information about assertions in tests, and incorporate them into the prediction models to further improve their performance. In addition, we want to enhance cross-project predictions. For that, we plan to include project characteristics into the model and focus model training on projects with similar properties.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “SOFIE, 01IS18012A”. The responsibility for this article lies with the authors.

REFERENCES

[1] Allen Troy Acree Jr. 1980. *On Mutation*. Technical Report. Georgia Institute of Tech.
[2] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. 2016. Can Testedness Be Effectively Measured?. In *Proc. 24th International Symposium on Foundations of Software Engineering (FSE'16)*. ACM.
[3] Vard Antinyan, Jesper Derehag, Anna Sandberg, and Miroslaw Staron. 2018. Mythical Unit Test Coverage. *IEEE Software* 35, 3 (2018).
[4] Author 1. 2018. Pitest: pull request for computing a full mutation matrix. (2018). https://github.com/hcoles/pitest/pull/511.
[5] Paul Barford and Mark Crovella. 1998. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26. ACM.
[6] Bartosz Bogacki and Bartosz Walter. 2006. Evaluation of Test Code Quality with Aspect-Oriented Mutations. In *Proc. 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP'06)*. Springer.
[7] Calin Caşcaval and David A Padua. 2003. Estimating Cache Misses and Locality Using Stack Distances. In *Proc. 17th International Conference on Supercomputing (ICS'03)*. ACM.
[8] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-Sampling Technique. *Journal of Artificial Intelligence Research (JAIR)* 16 (2002).
[9] John Joseph Chilenski and Steven P Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* 9, 5 (1994).
[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool For Java. In *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM.
[11] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978).
[12] Vladimir N Fleyshgakker and Stewart N Weiss. 1994. Efficient Mutation Analysis: A New Approach. In *Proc. 3rd International Symposium on Software Testing and Analysis (ISSTA'94)*. ACM.
[13] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How Hard Does Mutation Analysis Have to Be, Anyway?. In *Proc. 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE.
[14] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the Limits of Mutation Reduction Strategies. In *Proc. 38th International Conference on Software Engineering (ICSE'16)*. IEEE.
[15] Bernhard JM Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *Proc. International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09)*. IEEE.
[16] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. 2014. Teamscale: Software quality control in real-time. In *Companion Proc. 36th International Conference on Software Engineering (ICSE'14 Companion)*. ACM.
[17] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *Proc. 15th International Conference on Software Quality, Reliability and Security (QRS'15)*. IEEE.
[18] William E. Howden. 1982. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering (TSE)* 4 (1982).
[19] JC Huang. 1975. An Approach to Program Testing. *ACM Computing Surveys (CSUR)* 7, 3 (1975).
[20] Laura Inozemtseva and Reid Holmes. 2014. Coverage Is Not Strongly Correlated With Test Suite Effectiveness. In *Proc. 36th International Conference on Software Engineering (ICSE'14)*. ACM.
[21] Goran Petrović Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *Proc. 13th International Workshop on Mutation Analysis (MUTATION'18)*.
[22] Kevin Jalbert and Jeremy S Bradbury. 2012. Predicting mutation score using source code and test suite metrics. In *Proc. 1st International Workshop on Realizing AI Synergies in Software Engineering (RAISE'12)*. IEEE.
[23] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. 2009. A Novel Method of Mutation Clustering Based on Domain Analysis.. In *Proc. 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09)*, Vol. 9.
[24] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proc. 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*. IEEE.
[25] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *Transactions on Software Engineering (TSE)* 37, 5 (2011).
[26] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. 23rd International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM.
[27] Ron Kohavi and others. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14.
[28] Max Kuhn, the R Core Team, and further contributors. 2017. *caret: Classification and Regression Training*. https://CRAN.R-project.org/package=caret R package version 6.0-76.
[29] Richard J Lipton. 1971. *Fault Diagnosis of Computer Programs*. Technical Report.
[30] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An Automated Class Mutation System. *Software Testing, Verification and Reliability (STVR)* 15, 2 (2005).
[31] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9, 2 (1970).
[32] Rainer Niedermayr. 2018. TestAnalyzer. (2018). https://github.com/cqse/test-analyzer/ Computation of the Minimal Stack Distance (V3).
[33] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. 2016. Will My Tests Tell Me If I Break This Code?. In *Proc. 1st International Workshop on Continuous Software Evolution and Delivery (CSED'16)*. ACM.
[34] Rainer Niedermayr and Stefan Wagner. 2019. Dataset: Is the Stack Distance Between Method and Test Case Correlated With Test Effectiveness? (2019). DOI: http://dx.doi.org/10.6084/m9.figshare.7543409.v1
[35] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996).
[36] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proc. 15th International Conference on Software Engineering (ICSE'93)*. IEEE Computer Society Press.
[37] A Jefferson Offutt and Roland H Untch. 2001. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century*. Springer.
[38] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2017. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (2017).
[39] Sandra Rapps and Elaine J Weyuker. 1982. Data Flow Analysis Techniques for Test Data Selection. In *Proc. 6th International Conference on Software Engineering (ICSE'82)*. IEEE Computer Society Press.
[40] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient Mutation Testing by Checking Invariant Violations. In *Proc. 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM.
[41] David Schuler and Andreas Zeller. 2013. Checked Coverage: An Indicator for Oracle Quality. *Software Testing, Verification and Reliability (STVR)* 23, 7 (2013).
[42] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. 2008. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proc. 30th International Conference on Software Engineering (ICSE'08)*. ACM.
[43] Joanna Strug and Barbara Strug. 2012. Machine learning approach in mutation testing. In *Proc. 24th International Conference on Testing Software and Systems (ICTSS'12)*. Springer.
[44] Joanna Strug and Barbara Strug. 2018. Evaluation of the prediction-based approach to cost reduction in mutation testing. In *Proc. 39th International Conference on Information Systems Architecture and Technology (ISAT'18)*. Springer.
[45] Macario Polo Usaola and Pedro Reales Mateo. 2010. Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software* 27, 3 (2010).
[46] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. 2018. Descartes: a PITest engine to detect pseudo-tested methods-tool demonstration. In *Proc. 33rd International Conference on Automated Software Engineering (ASE'18)*. ACM Press.
[47] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive Mutation Testing. *Transactions on Software Engineering (TSE)* (2018).
[48] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997).