A Method for Finding Missing Unit Tests

Daniel Gaston
University of Delaware
Newark, DE, USA
dgaston@udel.edu

James Clause
University of Delaware
Newark, DE, USA
clause@udel.edu

***Abstract*—Because tests are important to the development process, developers need to know when a test suite is missing tests. *Missing tests*—tests that should be included in a test suite but are not—reduce the utility that developers can derive from a test suite. Currently, developers find missing tests by using coverage information such as line coverage or mutation coverage. However, coverage metrics are limited in their ability to reveal missing tests and show only *what* code needs to be tested, not *how* to test it.**

We present a method for finding missing tests that addresses the shortcomings of coverage metrics based on the fact that similar code entities are often tested in the same way. We are able to find *what* code is missing tests by identifying code entities which are not tested in the same way as other similar entities. We then show *how* a code entity with a missing test should be tested by leveraging the tests written for those similar entities. Our results show that our approach offers several benefits over a coverage-based approach and is able to find missing tests in a range of software projects while generating few erroneous identifications of missing tests.

***Index Terms*—software testing, static analysis, maintenance**

I. Introduction

Test suites are a beneficial software artifact that can support many aspects of the software development process. For example, they can give developers confidence that their applications are working correctly (e.g., [1, 2]), enable large-scale changes (e.g., [3, 4, 5]), and serve as a form of documentation (e.g., [6, 7, 8]). Unfortunately, crafting test suites is a difficult and time-consuming process, especially for modern software, which is often large, complex, and imperfectly understood [9]. As a result, test suites are often *missing tests* (i.e., tests that should be included in the test suite are not). For example, a test suite may be missing a test for a particular method or it may be missing tests for exceptional or unusual circumstances. Such missing tests reduce the usefulness of the test suite and prevent it from supporting the development of software.

Coverage criteria (e.g., [10, 11, 12, 13]), in which we include various types of mutation testing (e.g., [14, 15, 16]), are one approach used to detect missing tests. In the context of testing, coverage metrics indicate which entities meet the coverage criteria (are covered/killed) and which do not. Unmet criteria presumably indicate missing tests—if an entity does not meet them, the test suite is not adequately covering certain use-cases of the software. If tests were added to execute the uncovered entities or kill the remaining mutants, the test suite would no longer be missing those tests.

In practice, several factors limit the usefulness of coverage criteria at eliminating missing tests. First, it is often difficult or impossible to completely meet a particular criterion in a program. As a result, developers must spend significant amounts of time to determine whether it is possible to cover uncovered entities or kill remaining mutants. Second, even if criteria are met, this does not guarantee the absence of missing tests. For example, even if a statement is covered, the test suite may be missing tests that exercise significant edge cases. Finally, even when coverage criteria indicate the presence of missing tests, they cannot provide developers with details about how to write tests that are missing, which increases the difficulty of writing these tests [17]. For these reasons, coverage metrics only provide a partial solution for the problem of missing tests.

In this paper, we propose a novel static analysis technique to address the problem of missing unit tests. The approach is motivated by the observation that applications frequently contain *sibling groups*, which are groups of methods that accomplish the same goal in different ways. For example, each implementor of the Java Collections interface has a method for adding an element to the collection. Because they attempt to accomplish the same goal, siblings (i.e., the methods in a sibling group) are often tested in the same way (e.g., each `add` method is checked to ensure it correctly adds elements). If a majority of siblings have a test with a particular purpose, such as checking that adding `null` to a collection fails, it is reasonable to infer that: (1) siblings that do not have a test with that purpose are missing a test, (2) what is missing is a test with that specific purpose, and (3) the existing tests with that purpose can serve as starting points that developers can use when writing the missing test. In this way, our approach not only identifies the presence of missing tests but also provides additional useful information about what tests are actually missing. This information can be used to reduce the costs of creating missing tests by guiding developers or automated test case generation approaches (e.g., [18, 19]).

To assess the feasibility of the approach, we implemented it as a plug-in to the IntelliJ IDE [20] that analyzes Java applications and test suites written using the JUnit framework [21]. With the implementation, we conducted an empirical study of the test suites of 10 open source Java applications. The results of this study demonstrate that the technique is effective in finding missing tests and that it provides several benefits over approaches that use coverage information.

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00019

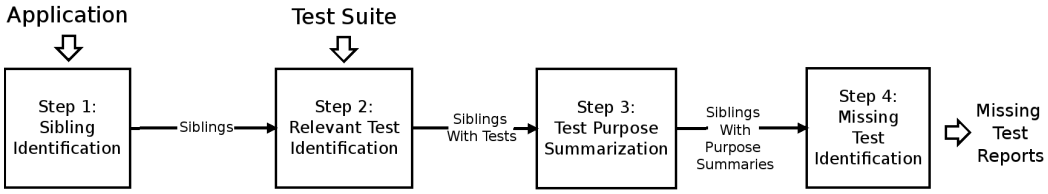

Fig. 1. Overview of our approach for finding missing tests.

This work makes the following contributions: a novel approach to address the problem of missing tests, motivated by presence of sibling groups in applications; a prototype implementation of our technique that analyzes Java applications and test suites written using the JUnit testing framework; and an evaluation of 10 applications/test suites that demonstrates that our approach can: (1) effectively find and report missing tests, (2) compare tests by determining if two tests have the same purpose, and (3) augment and extend the information provided to developers by code coverage.

II. Approach

Fig. 1 shows a high level overview of our approach for locating and reporting missing tests. The approach takes an application and its test suite as input and produces reports of missing tests as output in four main steps. First, it identifies sibling groups. Second, it associates each sibling with the tests that test it. Third, it summarizes the purpose of each test by determining how it tests its associated sibling. Finally, it identifies missing tests by determining whether a majority of siblings have a test with the same purpose. If so, siblings that are not tested in the same way are missing a test with the shared purpose.

To illustrate our approach, we will reference the simplified version of the Java Collections framework shown in Fig. 2. The `Collection` class declares several methods such as `remove` and overrides the `toString` method provided by its parent class. `Collection` also has two subclasses, `List` and `Set`, both of which override `remove`. The tests for each class are shown below their respective class in Fig. 2 (e.g., `CollectionTest` contains tests for `Collection`).

Given this example as input, our approach will report that two tests are missing. The first is that the `remove` method defined by `Collection` is missing a test that checks whether an element that has been previously added can be removed correctly. This test is missing because both `Set` and `List` have a test (`testRemove`) with this purpose. The second is that the `remove` method defined by `Set` is missing a test that checks whether removing an element that has not previously been added returns `null`. This test is missing because both `Collection` (`removeShouldReturnNullWhenEmpty`) and `List` (`removeFromListShouldReturnNullWhenEmpty`) have tests with this purpose.

The remainder of this section describes the four steps of our approach in detail and illustrates how missing tests are reported for our running example.

A. Sibling Identification

The purpose of the first step is to attempt to locate siblings for each method in the application. Recall that siblings are groups of methods that accomplish the same goal. In general, there are many ways to determine whether or not methods accomplish the same goal. In this work, we leverage the fact that applications written using object-oriented languages like Java use inheritance to group entities that share common functionality. More specifically, an overriding method must have the name, parameters, and return type (or sub-type) as the method it overrides. Because of these constraints, we can consider a method and all of its overriders as siblings.

While a method and all of its overriders are siblings, not every sibling group satisfies the constraints imposed by the other steps of the approach. Therefore, sibling groups are filtered based on two criteria. First, siblings that are not *testable* are removed from their containing sibling groups. A method is testable if it is declared by a class which can be directly instantiated from a test, hence methods declared in interfaces, abstract classes, anonymous classes, and nested classes are not testable and are removed. Second, sibling groups that contain fewer than three siblings are removed. Such groups are discarded because the fourth step of the approach checks whether a majority of siblings have a test with the same purpose, and it cannot do this with fewer than three siblings.

Given our running example as input, this step would consider four sibling groups: one containing `toString` declared by `Collection`, one containing `remove` declared by `List`, one containing `remove` declared by `Set`, and one containing `remove` declared by `Collection`, `List`, and `Set`. While none of the methods in these sibling groups are removed because they are not testable, three of the sibling groups are discarded because they contain fewer than three siblings. The only sibling group that meets all criteria is the one containing `remove` declared by `Collection`, `List`, and `Set`.

B. Relevant Test Identification

The goal of the second step is to associate each sibling in the sibling groups identified by the first step with its tests. To do this, we first collect all tests by locating methods that (1) are in a test directory, and (2) have either the `@Test` annotation or follow the JUnit test naming convention (i.e., the method's name starts with "test"). Then each method call on the test's static call graph is examined. If the method being called is a sibling, the test is associated with that sibling. In our running example, the `remove` method declared by

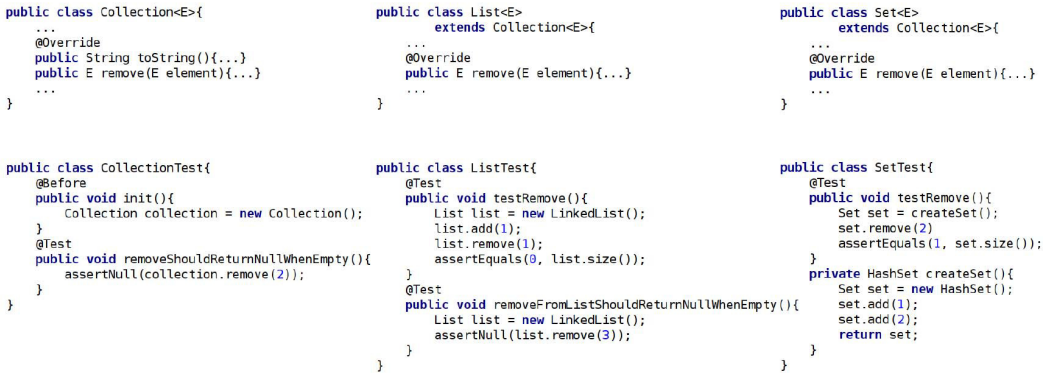

Fig. 2. Example application (top) and test (bottom) classes used to illustrate the approach.

the `List` class would be associated with the `testRemove` and `removeFromListShouldReturnNullWhenEmpty` tests declared by `ListTest`.

Because tests often rely on helper methods (i.e., methods called by the test that are declared in the same class) and setup methods (e.g., methods annotated with `@Before`), the bodies of such methods are considered to be part of the test and are inlined. For example, `testRemove` declared by `SetTest` calls `createSet` so the methods invoked in `createSet` are also considered to be part of `testRemove`. While assuming a test is testing each method that it calls is a coarse heuristic, it has the benefit of simplicity and does not appear to adversely impact the performance of the approach. If necessary, more advanced techniques could be applied, such as looking at the dynamic call trace of the test or identifying its focal methods [22]. After associating siblings with their tests, sibling groups that have fewer than 3 siblings with associated tests are discarded. Recall that the approach requires a majority to identify missing tests, which cannot exist in groups with fewer than 3 members.

C. Test Summarization

The goal of this step is to take the relevant tests identified in the previous step and produce a representation of each test that can be used to determine how each sibling method has been tested. To do this, we attempt to infer the purpose of each test. There are many possible ways to infer the purpose of a test; in this work, we use information that is readily available from the test itself—its name and its body. Ideally, a test name summarizes the test in a way that is understandable to other developers as a concise description of the test's purpose. However, because tests often have poor names [23], we also consider test bodies, which also describe the purpose of a test, albeit in a less human-understandable way.

While test names and bodies are the starting points for inferring the purposes of tests, they cannot be used directly. This is because the approach needs to determine whether siblings have tests with the same purpose, and comparing names and bodies directly (e.g., via string equality) would fail to account for variations that break equality but do not change the purpose. For example, a test for a method that consumes two objects could initialize the objects in either order without changing the semantics of the test. Similarly, developers may use slightly different names (e.g., `removeShouldReturnNullWhenEmpty` and `removeFromListShouldReturnNullWhenEmpty`). Such variations are common and can be introduced when multiple developers write tests or when tests are written across long periods of time. To account for these variations, we defined a summarization approach for each information source that abstracts the concrete names and bodies into a form that allows for more accurately comparing the purpose of the tests. These summarization approaches are described in more detail below.

1) Summarizing Test Names: To summarize test names, we use a natural-language program analysis-based approach that is inspired by prior work on extracting meaning from test names [23]. First, the test name is split into its constituent words using a purpose-built identifier splitter. Then, undesirable words, such as a leading "test" and numeric strings, are removed. Finally, the remaining words are classified to indicate the role they play in the test name. In particular, words are classified as either part of the *subject*—words that describe what functionality is being exercised by the test, part of the *response*—words that describe the expected outcome of the test, or *modifiers*—words that describe any additional conditions that are applied to the subject. Classification is done using a combination of patterns and heuristics. For example, we found that the various parts of the name are typically indicated by keywords or their relative positions: the subject often comprises words at the start of the name, the response often comes after words such as "should," "is," and "returns," and modifiers often come after words such as "when," "given," and "with". As the words in the test name are traversed in order from left to right, they are added to the *subject* category until a word that is a *response* or *modifiers* keyword is reached, at which point the words are added to the appropriate category until a different keyword is reached. The keywords themselves are not included in the categories because they are often a source of inconsequential variation (e.g., different developers may prefer to use different keywords) and modifier keywords are often repeated. Note that the words are stored in an ordered list, since the order of words can be important in natural

language artifacts such as test names. The summary for a test's name will always have *subject* but, for some tests, the *response* and *modifiers* categories may be empty.

In our running example, the summary for `removeShouldReturnNullWhenEmpty` is that the *subject* of the test is ["remove"], the *response* is ["null"] and the *modifiers* are ["empty"].

2) Summarizing Test Bodies: To summarize test bodies, we extract the set of methods called by the test. As in the relevant test identification step, setup and helper methods are inlined into the test body so that the method calls that they contain are also included in the summary. We chose to summarize test bodies based on method calls because their addition, removal, and replacement tends to have a large impact on the semantics of a test. Other aspects of test bodies, such as input parameters and variable names, can often be changed without impacting the purpose of the test and therefore should not be part of the summary. For example, a method argument could be provided directly to a method call or it could be stored in a temporary variable that is used as the argument. The specific style that is used does not impact the purpose of the test. Additionally, we store the method calls as sets rather than ordered lists to allow for further variation between two tests. This is desirable because we have found that the addition of a method call that was already made or a change in the order of method calls does not necessarily change the purpose of a test.

In our running example, the summary for `removeShouldReturnNullWhenEmpty` declared by `CollectionTest` contains the calls to `remove` declared by `Collection` and `assertNull` from the JUnit framework.

D. Missing Test Identification

The last step of the approach is to generate missing test reports by analyzing the siblings and the summaries of their associated tests. Intuitively, a method is missing a test if it lacks a test with a summary that is shared by a majority of the tests of its siblings. More formally, we say that a sibling *supports* a summary s if at least one of its tests has a purpose that is the same as or *subsumes* s. Then, if a majority of siblings that have associated tests in a group support s, siblings which do not support s are missing a test with the purpose that s represents. Siblings without tests cannot provide support and therefore are not included in the majority calculation.

Subsumption is a broadening of the concept of equality that takes into account that a test can serve the same purpose *and more* of another test. For example, the purpose of `removeShouldReturnNullWhenEmpty` defined by `CollectionTest` subsumes the purpose of `testRemove` defined by `SetTest` because the focus of both tests is the `remove` method (i.e., the *subject* is ["remove"]) and `removeShouldReturnNullWhenEmpty` includes additional conditions on how `remove` should be tested (e.g., its *modifiers* are ["empty"]).

To determine whether a name summary SN_1 subsumes another name summary SN_2, the lists associated with each category are checked to see whether the *subject*, *response*, and *modifiers* of SN_2 are prefixes of the corresponding lists in SN_1. We use prefixes to determine subsumption as opposed to simple membership due to the fact that word order can carry meaning in test names. Note that in this comparison, an empty list is considered a prefix of all lists with a size greater than zero. If the lists of SN_2 are prefixes of the corresponding lists in SN_1, then SN_1 subsumes SN_2 meaning that the sibling associated with the test that SN_1 came from supports SN_2. To determine whether one body summary SB_1 subsumes another body summary SB_2, we simply do a set comparison between the two summaries, which are themselves sets of methods. SB_1 subsumes SB_2 if $SB_2 \subset SB_1$, meaning that the sibling associated with the test that SN_1 came from supports SN_2.

Once support is calculated for all summaries, a report is generated for any method that does not have a summary that is supported by a majority of its siblings. A report contains the location of the method missing the test, the name or body summary used to represent the purpose of the test, and the set of tests from which the summaries supporting the report were derived. Both the summary and the set of tests supporting that summary provide developers with a starting point with which to address the missing test report.

Only reporting a missing test when it has the support of a majority of siblings helps filter out noise that was introduced in Step 2. Unrelated tests and tests that are tailored for a particular method will not be able to amass enough support to be reported missing. Additionally, using a majority to decide which tests are missing allows us to back each reported missing test with some level of confidence. If a report with the same summary is generated for a method multiple times, this is considered a duplicate and only the report with the highest level of support is kept. This situation can arise if a method is in multiple sibling groups (i.e., it is inherited from a parent class, and is in turn overridden by a child class).

In our running example, a majority of methods—`remove` from `Collection` and `List`—have tests that support a name summary where the *subject* is ["remove"], the *response* is ["null"], and the *modifiers* are ["empty"], so we report it missing from the `remove` method in `Set`. This summary has a support of $\frac{2}{3}$ since it is present in the tests of two methods and missing from the tests of one method. Similarly, a body summary containing the method calls `add`, `remove`, `assertEquals`, and `size` is supported by the tests of the `remove` methods from `Set` and `List`, so it is reported missing from the `remove` method from `Collection`. This summary also has a support of $\frac{2}{3}$ since it is present in the tests of two methods and missing from the tests of one method.

III. Evaluation

We evaluated our technique by using a prototype implementation to answer the following research questions:

RQ1—Effectiveness: How effective is our approach in finding missing tests?

RQ2—Comparison of Summarization Approaches: Are both test summarization methods needed to find missing tests?

TABLE I: Subjects used in our evaluation.

		# Application				# Test		# Reports			
Subject	**Version**	**Classes**	**Methods**	**Groups**	**Siblings**	**Classes**	**Methods**	**MT**	**CR**	**ER**	**Total**
Barbecue	1.5	75	649	7	47	25	171	35	6	0	41
Checkstyle	8.23	3,360	8,544	13	659	304	3,097	1,028	58	0	1,086
Comm. Coll.	4.3	544	4,545	148	56	215	1,138	78	0	2	83
Comm. Math	3.4.1	1,120	7,823	61	330	444	3,192	371	22	31	424
JFreeChart	1.5.0	684	8,782	89	648	341	2,176	317	47	38	402
Joda Time	2.9.7	321	4,738	34	134	135	4,238	29	13	15	57
Jopt	6.0	70	430	2	14	132	539	18	2	0	20
Tablesaw Core	0.36.0	291	3,646	25	187	81	924	176	9	24	209
Threadly	5.36	347	2,176	8	50	141	1,331	23	0	0	23
ZXing Core	3.3.3	255	1,576	2	16	129	563	11	0	0	11
Total		7,067	42,909	389	2,141	1,974	17,369	2,086	157	113	2,356

RQ3—Comparison to Coverage: How does our method compare to coverage as a method of finding missing tests?

The remainder of this section presents our: (1) prototype implementation, (2) experimental subjects and selection process, (3) methodology for generating experimental data, and (4) results and discussion for each research question.

A. Prototype Implementation

Our technique is implemented as an IntelliJ plugin. This platform was chosen due to its popularity among software developers as well as the ease of use of its plugin API, which gives access to IDE internals. The plugin API provides convenient wrapper functions to search for different types of nodes on the abstract syntax tree, which is heavily annotated with semantic information. Our plugin provides a menu item that can be used to analyze an application loaded in the IDE.

B. Subject selection

We chose the 10 open source applications shown in Tab. I for our evaluation. The first and second columns show the name and the version of each subject. The two columns under the header *Application* show the number of application classes and methods in each subject and the two columns under the header *Test Suite* show the number of test classes and methods in each test suite. For example, the first row shows that Barbecue has 75 application classes, 649 application methods, 25 test classes, and 171 test methods. The number of classes and methods was calculated using the IntelliJ plugin API, which can enumerate various types of program elements.

We chose these subjects for several reasons. First, they come from a variety of organizations and developer teams, such as the Apache Software Foundation and Google. Second, they cover a wide variety of application domains. For example, Commons Math is a mathematical library and JFreeChart is a chart creation framework. Third, they have relatively large test suites that we were able to run to obtain coverage information, which is important for answering our third research question. Fourth, they vary in size and structure, which can be seen by looking in Tab. I at the numbers of application classes and methods and the ratio of classes to methods. Finally, many of these applications have served as subjects for other software testing research (e.g., [24, 25, 26]). These factors give us confidence that we have selected representative subjects that do not present a threat to the generality of our results.

C. Methodology

The experimental data used in our evaluation was obtained by loading each application into the IntelliJ IDE and analyzing it with our plugin. The plugin executed in under 30 seconds for most subjects and under 3 minutes for the largest subjects. In total there were 2,356 reports generated for our subject applications. We have released these results along with the tool: doi.org/10.5281/zenodo.3987298. The number of reports for each subject is shown in column *Total* in Tab. I.

In order to evaluate the approach, each of 2,356 reports was manually examined and classified as either a missing test report, a candidate for refactoring report, or an erroneous report. Each author performed this classification independently using the information provided by the report as well as the application and its test suite. Agreement was high and cases of disagreement were handled by discussion among the authors until a consensus was reached.

Missing test reports are reports that correctly indicate that a test is missing. For example, a body report with a support level of $\frac{120}{121}$ (99 %) indicating that the `clone` method declared by the `SamplingXYLineRenderer` class from JFreeChart should have a test that calls `clone`, `getClass`, `equals`, and `assertTrue` is considered to be a missing test report because `clone` has no tests like this.

Candidate for refactoring reports are reports where the test indicated by the report does exist but is inconsistent with the tests shared by its siblings in a minor way. For example, a name report with a support level of $\frac{119}{121}$ (98 %) indicating that the `clone` method declared by the `TimeTableXYDataset` class from JFreeChart does not have a test with the *subject* ["cloning"] is considered to be a candidate for refactoring report because `clone` has a test with the similar but different *subject* ["clone"]. While they do not indicate missing tests, we believe such reports to be useful, as these inconsistencies are trivial to fix and may inhibit comprehension and maintenance. In future work, we plan to extend the summarization approaches to handle these inconsistencies (e.g., by applying stemming and lemmatization). This would allow many candidate for refactoring reports to be automatically classified.

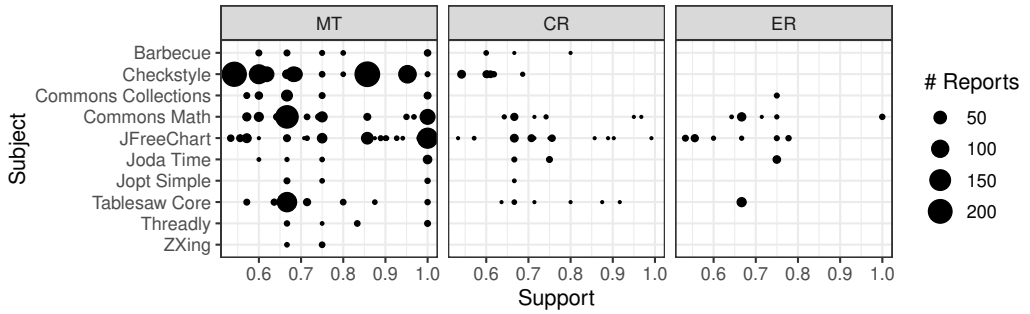

Fig. 3. Reports shown by subject and support level for each category of report.

Erroneous reports are reports where the test indicated by the report cannot exist. For example, a name report with a support level of $\frac{3}{4}$ ($75\,\%$) indicating that the `getYValue` method declared by the `XYBarDataset` class from JFreeChart does not have a test with the *subject* ["add" "series"] is considered to be an erroneous report because there is no `addSeries` method declared in `XYBarDataset`. Therefore such a test is impossible to write. Note that while we consider such reports erroneous it is possible that they may provide useful information about missing functionality in an application or highlight situations that warrant further investigation. For example, why do `DefaultXYDataset`, `DefaultXYZDataset`, and `DefaultIntervalXYDataset` declare an `addSeries` method while `XYBarDataset` does not? Perhaps `addSeries` should be part of the parent interface `XYDataset`. In addition, in future work we plan to add support for automatically identifying cases where a summary indicates that a method that does not exist should be called. This will allow for automatically filtering some of the erroneous reports.

The number of reports in each category for each subject is shown in the seventh, eighth, and ninth columns in Tab. I: column *MT* shows the number of missing test reports; column *CR* shows the number of candidate for refactoring reports; and column *ER* shows the number of erroneous reports.

Note that we do not calculate the false negative rate (tests that are missing but not identified by our approach) due to the following complications inherent in this calculation: (1) there are possibly infinite tests that could be written for an application, and (2) there is not a straightforward way of identifying a subset of these to use as a golden set of missing tests against which to judge our approach.

D. RQ1—Effectiveness

The goal of our first research question is to evaluate the effectiveness of the approach in producing useful reports. In order to investigate this research question, we consider the classification of the reports produced for our subject applications as well as their support levels.

Fig. 3 presents this data as a series of scatter plots, one for each category. In each plot, each point represents the number of reports in the corresponding category with the position along the x-axis showing the support level and the position on the y-axis showing the subject the report came from. The legend to the right of the plot shows to what extent dots vary in size depending on how many reports they represent. For example, the left scatter plot shows that Checkstyle has a large number of missing test reports with a support level of 0.5 and the middle scatter plot shows that Tablesaw Core only has a small number of candidate for refactoring reports with a support level of ≈ 0.8. The large number of reports with certain levels of support is partially due to the fact that support is not evenly distributed because sibling groups can only produce certain support levels (e.g., reports for a group with size 3 can only have a support level of $\frac{2}{3} \approx 0.6$ or $\frac{3}{3} = 1.0$).

Based on the data presented in Fig. 3, we can make several observations. First, there are missing test reports generated for each subject. This indicates that the problem of missing tests is not a phenomenon unique to a small number of applications. Second, the number of reports for a subject is positively correlated with the size of the application and the size of its test suite in terms of the number of application and test classes/methods. This suggests that larger applications and test suites are more likely to have missing tests, possibly because in these situations it is more difficult for a developer to have a sufficiently broad understanding of the various parts of the application and how it should be tested. Third, the majority of reports are missing test reports ($2{,}086$) and the number of useful reports (i.e., missing test and candidate for refactoring reports) far outweigh the number of erroneous reports ($2{,}243$ compared to 113). This is significant because high false positive rates (e.g., erroneous reports) are one of the main reasons that developers abandon static analysis tools [27]. The erroneous reports are not evenly distributed across the subjects, with the majority ($\approx 80\,\%$) coming from three subjects, Commons Math, JFreeChart, and Tablesaw Core. Further investigation revealed the cause of this to be the presence of groups whose siblings cannot always share the same tests. For example, in Tablesaw Core, the implementation of the `where` method from the `Column` interface is used for selecting a subset of data. Some of the `where` siblings, such as the ones declared in `TextColumn` and `StringColumn`, have tests for selection based on string case, which does not make sense for `where` methods declared in classes such as `BooleanColumn` and `DoubleColumn`. The presence of such subgroups increases the rate of erroneous reports due to differences that exist between siblings in different subgroups.

Finally, reports with higher levels of support are more likely to be missing test reports than erroneous reports. For example, with the exception of Commons Math, none of the subject applications have an erroneous report with support greater than 0.8. This suggests that developers can reduce the number of erroneous reports by filtering reports based on support level. However, this may also reduce the number of useful reports by filtering out both missing test and candidate for refactoring reports. For example, while filtering reports with support less than 0.8 would lower the number of erroneous reports to 6, it would also eliminate 1,470 useful reports (1,365 missing test reports and 105 candidate for refactoring reports).

Based on above observations, we conclude that our approach is effective at providing developers with useful information about how to improve the quality of their test suites.

E. RQ2—Comparison of Summarization Approaches

The goal of our second research question is to compare our test summarization approaches. Specifically, we are interested in finding out if the reports produced using one summarization approach are (1) duplicates of reports produced using the other summarization approach, and (2) found to be useful at a higher rate than reports produced using the other summarization approach. Understanding differences between the summarization approaches will provide valuable feedback for improving our tool and guidance for how it should be used in practice.

1) Report Duplication: First, we investigated whether reports produced using one summarization approach are duplicates of reports produced using the other summarization approach. We are interested in knowing this because the existence of many duplicates would indicate that only one summarization approach is required.

To determine if duplicate reports exist, we need to be able to compare reports. However, name and body reports cannot be directly compared, as they have different representations (ordered lists and sets, respectively). Instead, we use the support sets of the reports as the basis for comparison because they are in terms of test methods for both types of report. We have the option of using several existing set comparison algorithms (Szymkiewicz-Simpson coefficient, Jaccard similarity, Sørensen-Dice coefficient, etc.). We chose Jaccard similarity because it takes differences in set size into account. For example, a set of size 3 and a set of size 100 will always have a low Jaccard similarity even if the former is a subset of the latter. This is desirable because the size of the support sets is an important factor in the similarity of two reports.

Using Jaccard similarity we can find the similarity of the most similar report generated by the other summarization approach as follows: $\text{similarity}(r, R) = \max_{r' \in R} \text{jaccard}(r, r')$ where r is a report based on one summarization approach and R is all reports generated using the other summarization approach. The results of this equation range from 0 to 1, with 1 indicating that the support sets are identical (i.e., both approaches report the same missing test) and 0 indicating that the support set r has no overlap with any report in R (i.e., it is unique to one summarization approach). Note that this calculation is not always symmetric (i.e., n may be the most similar name report for body report b but b may not be the most similar body report for n).

TABLE II: Percentage of reports in each category.

	% of Non-Duplicate Reports					
	MT		CR		ER	
Subject	**N**	**B**	**N**	**B**	**N**	**B**
Barbecue	89	84	11	16	0	0
Checkstyle	93	96	7	4	0	0
Commons Collections	92	100	0	0	8	0
Commons Math	80	90	7	6	12	4
JFreeChart	84	65	8	21	8	14
Joda Time	27	50	27	50	45	0
Jopt Simple	80	93	20	7	0	0
Tablesaw Core	94	96	6	4	0	0
Threadly	100	100	0	0	0	0
ZXing	100	0	0	0	0	0
All	88	89	7	8	5	3

Fig. 4 shows the results of our similarity calculations as a histogram that plots the distribution of name and body report similarities. The x-axis divides the similarity range into 20 equally spaced bins and the y-axis shows the number of reports that fall into a given bin. Each bin has two bars: one for name reports (light) and one for body reports (dark). For example, the left bar for the 0.75 bin shows that the number of name reports (29) whose highest similarity score with a body report is 0.75, and the right bar shows that the number of body reports (30) whose highest similarity score with a name report is 0.75.

Based on the data shown in Fig. 4, we can make two observations. First, there are many fewer duplicate reports, 376 ≈16 %, than there are unique reports, 894 (456 name and 429 body) ≈38 %. Note that as a sanity check, we confirmed that all duplicated reports were classified in the same category. The small number of duplicate reports show that the summarization approaches generate different reports, which means both should be used. Second, there are a large number of reports that have high similarity scores (i.e., scores above 0.9). This suggests that there are many tests with identical names but small differences in the body or vice versa.

2) Effectiveness: Second, we investigated whether each summarization approach produces useful reports.

To investigate this, we partitioned the non-duplicate reports by summarization approach. Since duplicated reports are in the same category regardless of which approach was used to generate them, they do not contribute to answering this research question and therefore are excluded.

The results of this partitioning can be seen in Tab. II. The first column shows the subject applications. The next six columns show the percentage of non-duplicate reports for the respective subject application that were generated by each summarization approach, **N**ame or **B**ody, grouped by report category—missing tests, candidates for refactoring, and erroneous reports. For example, in Commons Math, 80 % of name reports and 90 % of body reports are missing test reports, 7 % of name reports and 6 % of body reports are candidate for refactoring reports, and 12 % of name reports and 4 % of

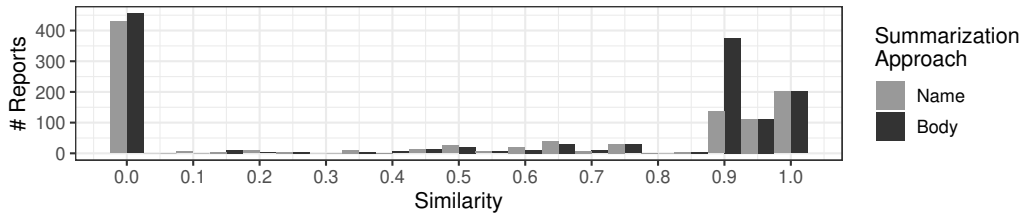

Fig. 4. Histogram of the similarity of reports.

body reports are erroneous reports. The last row shows the same break down across all subject applications.

Based on the data shown in Tab. II, we make two main observations. First, as shown in the *All* row, both summarization approaches generate reports at nearly identical rates across all subjects, differing at most by $2\,\%$. This indicates that both approaches are effective at generating useful reports.

Second, there are exceptions to this trend on a per-application basis. The largest discrepancy between the rates exists in Joda Time, where $45\,\%$ of name reports and $0\,\%$ of body reports are erroneous. Further investigation revealed that these erroneous reports were for the `getChronology` method declared by the `Partial` class. Its siblings, declared in `LocalDateTime` and `LocalDate`, are used in tests for constructors of their respective classes, such as `testConstructor_long1`, which checks the constructors of the `LocalDateTime` and `LocalDate` classes with a `long` input parameter. However, the `getChronology` declared in `Partial` cannot have this test, since `Partial` has no constructors that have a `long` parameter. This indicates that the effectiveness of the summarization approaches can depend on specific details of the test suite of the application.

3) Conclusions: In summary, we have shown that both name and body summaries can be used to find unique missing tests. Additionally, both summarization approaches can be effectively used to find missing tests. Further, they generate reports in each category at similar rates across all subjects, with exceptions to this trend occurring in a minority of our subjects. Thus, we conclude that there is merit in using both summarization approaches to find missing tests.

F. RQ3—Comparison to Coverage

The goal of our third research question is to compare our approach for finding missing tests against an approach based on statement coverage. As coverage is the most common way of finding missing tests, this is a useful comparison to make.

The typical workflow for finding missing tests using statement coverage starts with running the test suite and recording the execution statistics. These statistics are usually aggregated in a report document, which allows developers to see coverage statistics at various levels of granularity (e.g. package, class, method). We carried out this procedure on each of our subject applications and present the results at the method level.

Tab. III shows the number of methods that are uncovered ($0\,\%$ coverage), partially covered ($1\,\%$ to $99\,\%$ coverage), and fully covered ($100\,\%$ coverage) in each subject. For example, in Barbecue there are 245 uncovered methods, 23 partially covered methods, and 238 fully covered methods. This data was obtained by running the test suite of each application and collecting coverage information using the JaCoCo tool [28]. Note that in some cases the numbers of methods presented here are different than the ones presented in Tab. I. This is because IntelliJ and JaCoCo count methods differently. For example, JaCoCo does not include abstract methods or empty methods as these cannot be covered.

After coverage information is obtained, it must be manually examined to determine if there are any missing tests. Because coverage information is reported for every method (between 430 and $8{,}782$ for our subjects), developers must prioritize where to spend their limited time and effort. The typical prioritization strategy is to ignore fully covered methods and focus on those that are uncovered or partially covered. While this does provide some benefit, the number of methods that need to be examined can still be large (between 38 and $4{,}157$ for our subjects). In addition, developers must understand more than just the method itself in order to understand if it is missing a test. At a minimum, they must examine and understand the purposes of all the existing tests for that method. If they have detailed knowledge of the method's requirements, then this might be enough to identify whether or not a method is missing tests. If not, they must broaden their investigation to include other parts of the application and its test suite as well as external resources. Even when focusing only on uncovered and partially covered methods, identifying whether or not a test suite is missing tests is an arduous, time-consuming process. Finally, if developers decide that a test is missing, they must write that test from scratch. Again, this requires not only a detailed understanding of the method and its requirements but also the ability to match those requirements to concrete test cases. Overall, addressing the problem of missing tests using coverage information alone clearly has many drawbacks.

In contrast, our approach provides developers with a simpler and easier process that has several benefits. The first benefit is that the reports generated by our approach provide contextual information such as the support set, which explains why the approach believes a test is missing and simplifies the process of understanding whether or not a method is missing a test. Coverage information provides nothing beyond which elements were executed (and potentially how many times).

The second benefit is that the number of methods which developers need to examine is much smaller. While the number of uncovered and partially covered methods that developers would need to examine ranges from 38 to $4{,}157$ for our subjects, the number of reports generated by our approach

TABLE III: Coverage of methods in each subject.

	# Methods		
Subject	**None**	**Partial**	**Full**
Barbecue	245	23	238
Checkstyle	3,867	290	3,629
Commons Collections	420	260	2,401
Commons Math	669	556	4,642
JFreeChart	2,900	1,078	3,889
Joda Time	254	206	2,729
Jopt Simple	28	10	362
Tablesaw Core	1,195	285	1,881
Threadly	141	111	1,013
ZXing	195	308	1,043
Total	9,914	3,127	21,827

ranges from 11 to 1,086. In some cases, the number of reports is an order of magnitude smaller than the number of uncovered and partially covered methods.

To gain a more detailed understanding of how the reports generated by our approach compare to what can be learned from coverage information, we matched each missing test report with the coverage of the method it reports. Fig. 5 shows the results of this process as a histogram. In the figure, the x-axis shows the three levels of coverage: uncovered (0 % coverage), partially covered (1 % to 99 % coverage), and fully covered (100 % coverage). The y-axis shows the number of reports whose reported method is in each coverage group. Coverage data was unavailable for the methods of 27 reports, so the results represent 2,059 out of 2,086 missing test reports.

From this data, we can see that there is some overlap in the reports generated by our approach and uncovered and partially covered methods. In particular, there are 323 reports whose method is uncovered and 91 reports whose method is partially covered. These are reports that indicate missing tests that could be found using the coverage-based approach detailed above. However, our approach directly presents these cases to developers without the need for extensive manual analysis. Note that we are not claiming that our approach finds all missing tests. It is possible that developers may find more or different missing tests using coverage information. Nevertheless, we believe that the savings in terms of manual effort provided by our approach are a significant benefit.

Fig. 5 also highlights the third benefit of our approach: it can identify missing tests for methods that are fully covered. In fact, the majority of missing test reports generated for our subjects (1,645 of 2,059) are for methods with 100 % coverage. These are reports indicating missing tests that *could not* be found using the coverage-based approach. Locating them would require analyzing all methods regardless of their coverage level. The ability of our approach to identify such cases is especially promising, as developers often incorrectly believe that fully covered methods are also sufficiently tested.

The final benefit of our approach is that it provides developers with guidance when they implement the missing tests. Again, coverage only provides information about what entities are executed. In the coverage-based approach, developers must either write tests from scratch or manually search for example tests within the test suite. Our approach provides developers with example tests directly, eliminating the need for this. Moreover, the information in the reports can help developers maintain consistency with existing related tests, which can prevent the introduction of deviations that can impact comprehension and maintenance tasks. For example, consider a name report with a support level of $\frac{3}{3}$ (100 %), which indicates that the `hashCode` method declared by the `TimePeriodValues` class from JFreeChart does not have a test name summary whose *subject* is [“hash”,“code”].

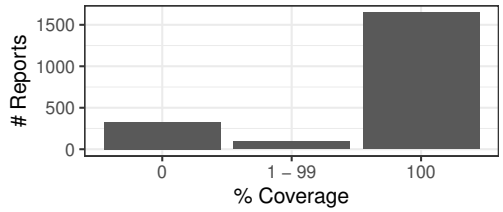

Fig. 5. Histogram of the coverage of reported methods.

Fig. 6a shows two representative tests from the siblings of `TimePeriodValues` for their `hashCode` methods that support the report. Due to space limitations we have elided part of each test. The removed parts are four repetitions of the fifth through eighth lines in each test with different arguments to the `add` method. Fig. 6b shows the test that we believe developers would write to address the report. As these figures show, only trivial differences exist between the tests for the siblings (the other sibling differs in the same manner) and the test that would be written to address the missing test report. In particular, the types of `s1` and `s2` must be changed to `TimePeriodValues` which necessitates changing the constructor calls and the parameters to the `add` method.

In summary, our approach provides significant benefits over a coverage-based approach for finding and addressing missing tests. First, developers have fewer reports to consider. Second, the reports require less manual analysis. Third, our approach can find missing tests that cannot be found using the coverage-based approach. Finally, the reports provide information that can be used to more easily implement the missing tests.

G. Threats To Validity

Although a diverse selection of subjects was used for evaluation, the results might not generalize to other subjects. Additionally, all subjects examined are written in Java and use the JUnit testing framework and the results obtained may not generalize to other programming languages and testing disciplines. We relied on heuristics provided by the Intellij plugin API in order to find test classes and determine which of their methods are test methods. Inaccuracies in these heuristics could have led to some test classes or methods not being identified. Additionally, the results are sensitive to how sibling groups are formed since disparate functionality can be grouped together in an unexpected ways, so a different method of grouping could lead to a different outcome. However, the low erroneous report rate shows that the impact of these situations is low.

```java
@Test
public void testHashCode() {
    TimeSeries s1 = new TimeSeries("Test");
    TimeSeries s2 = new TimeSeries("Test");
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());

    s1.add(new Day(1, 1, 2007), 500.0);
    s2.add(new Day(1, 1, 2007), 500.0);
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());
    //fragment
```

```java
@Test
public void testHashCode() {
    XYSeries s1 = new XYSeries("Test");
    XYSeries s2 = new XYSeries("Test");
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());

    s1.add(1.0, 500.0);
    s2.add(1.0, 500.0);
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());
    //fragment
```

(a) Example tests from siblings of `TimePeriodValues`.

```java
@Test
public void testHashCode() {
    TimePeriodValues s1 = new TimePeriodValues("Test");
    TimePeriodValues s2 = new TimePeriodValues("Test");
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());

    s1.add(new TimePeriodValue(new Day(), 55.75));
    s2.add(new TimePeriodValue(new Day(), 55.75));
    assertEquals(s1, s2);
    assertEquals(s1.hashCode(), s2.hashCode());
    //fragment
```

(b) Test that should be written for `TimePeriodValues`.

Fig. 6. An example of how existing tests can be used to fill in a missing test.

Further, we used a conservative heuristic for establishing test to code links. We chose to use the static call graph to associate tests to code because it is simple to implement since IntelliJ provides this information and it is conservative in that any method that could be called by a test is associated with that test. However, related work has demonstrated that there are many other approaches for performing test-to-code traceability (e.g., [29, 30, 31]). Because of the different performances of such approaches, using them instead of the static call graph may impact the results of our work. For example, using a less conservative technique (e.g., the approach proposed by Ghafari et al. which identifies focal methods $\approx$85 % of the time [29]) would likely reduce the number of siblings and sibling groups considered by the approach because there would be fewer test-to-code links. However, without additional studies it is not clear how using an alternative technique would impact the overall results of our approach.

Next, the performance of our approach may also be impacted by various test smells. Eager tests [32], would be associated with each method that the test calls. This could lead to tests being included in more sibling groups. However, this would only lead to false positives when the same pattern of calls is repeated among a majority of siblings in these additional groups. Tests that rely on dependencies could also impact the formation of sibling groups in a similar way. Both of these issues could potentially be addressed by using an alternative test-to-code traceability technique as discussed above. Additionally, an expanded evaluation could examine the causes of erroneous reports more in depth and quantify the impact of these test smells. Finally, note that assertion roulette [32] is unlikely to impact our results. Because body summaries are sets of methods, calling the same method multiple times will not change the outcome.

Finally, the reports were categorized with the subjective judgment of people who are not developers for any of the subjects. This lack of precise domain knowledge could have influenced how reports are categorized, which could call into question the number of reports determined to be in each category. In order to mitigate this, we formed guidelines for categorizing reports, which ensured consistency.

IV. Related Work

A large amount of research has been conducted in evaluating and improving test suites. Given the controversial relationship between coverage metrics and fault finding capabilities [33, 12, 34, 11, 15, 10], many alternative approaches have been proposed to better assess and improve test suites. These include checked coverage [35], (in)direct coverage [24], as well as various forms of mutation testing [26] [36]. These metrics have also been used to drive test case generation [16, 19, 18, 37]. Our approach does not seek to replace, but rather to complement this body of work, since we use existing tests to propose improvements without attempting to provide a comprehensive solution to address all insufficiently covered code. Directed test case generation in particular is an area in which information learned using our approach can be incorporated to potentially improve performance. Additionally, our method takes a lightweight static analysis approach to find potential missing tests as opposed to the more expensive approaches taken by other methods such as mutation testing.

Prior work has found that code clones are both common in in test suites [38] and require novel forms of code clone detection [39]. Other researchers have investigated how this redundancy can be leveraged to improve test suites. Landhäußer and Tichy propose cloning test cases to reuse for similar components, but their method lacks the ability to automatically identify those components [40]. Mirzaaghaei et al. use existing tests to support test suite evolution, using a coverage driven approach [41]. Mining open source test suites to augment an application's test suite has also been proposed [42, 43].

V. Conclusions

In this work, we have presented a novel method for detecting test suite deficiencies based on static analysis of program structure and the different ways of summarizing tests to facilitate finding tests with similar purposes. We presented the results of an empirical study of 10 real world applications that demonstrate: (1) missing tests are a common problem in test suites and, but also (2) our technique is capable of providing feedback to address them. Consequently, our results show that existing tests can be leveraged by developers as a straightforward way of improving test suites.

References

[1] P. Bourque, R. E. Fairley *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[2] M. R. Lyu, "Software reliability engineering: A roadmap," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 153–170.

[3] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 13, no. 3, pp. 277–331, 2004.

[4] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.

[5] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *International Conference on Product Focused Software Process Improvement*. Springer, 2010, pp. 3–16.

[6] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 2007, pp. 213–222.

[7] J. Van Geet, A. Zaidman, O. Greevy, and A. Hamou-Lhadj, "A lightweight approach to determining the adequacy of tests as documentation," *Proc. PCODA*, vol. 6, pp. 21–26, 2006.

[8] T. Haendler, S. Sobernig, and M. Strembeck, "An approach for the semi-automated derivation of uml interaction models from scenario-based runtime tests," in *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, vol. 1. IEEE, 2015, pp. 1–12.

[9] D. Kumar and K. Mishra, "The impacts of test automation on software's cost, quality and time to market," *Procedia Computer Science*, vol. 79, pp. 8–15, 2016.

[10] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 72–82.

[11] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 255–268.

[12] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[13] M. M. Hassan and J. H. Andrews, "Comparing multi-point stride coverage and dataflow coverage," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 172–181.

[14] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[15] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[16] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.

[17] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[18] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[19] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, "The emerging field of test amplification: A survey," *arXiv preprint arXiv:1705.10692*, 2017.

[20] "Jetbrains intellij plugin development," http://www.jetbrains.org/intellij/sdk/docs/welcome.html, 2020, accessed: 2019-01-15.

[21] "Junit," https://junit.org, 2020, accessed: 2020-01-09.

[22] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation*, 2015, pp. 61–70.

[23] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 625–636.

[24] C. Huo and J. Clause, "Interpreting coverage information using direct and indirect coverage," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 234–243.

[25] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?" in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. ACM, 2016, pp. 23–29.

[26] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *Empirical Software Engineering*, pp. 1–31, 2018.

[27] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 672–681.

[28] "Jacoco java code coverage library," https://www.jacoco.org/jacoco/, 2020, accessed: 2020-01-15.

[29] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *2015 IEEE 15th International Working Conference on*

Source Code Analysis and Manipulation (SCAM). IEEE, 2015, pp. 61–70.

[30] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[31] V. Csuvik, A. Kicsi, and L. Vidács, "Evaluation of textual similarity techniques in code level traceability," in *International Conference on Computational Science and Its Applications*. Springer, 2019, pp. 529–543.

[32] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, 2001, pp. 92–95.

[33] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[34] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 57–68.

[35] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 90–99.

[36] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[37] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 394–397.

[38] W. Hasanain, Y. Labiche, and S. Eldh, "An analysis of complex industrial test code using clone analysis," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 482–489.

[39] B. van Bladel and S. Demeyer, "A novel approach for detecting type-iv clones in test code," in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. IEEE, 2019, pp. 8–12.

[40] M. Landhäußer and W. F. Tichy, "Automated test-case generation by cloning," in *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 83–88.

[41] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 231–240.

[42] W. Janjic, "Reuse-based test recommendation in software engineering," Ph.D. dissertation, 2014.

[43] S. Makady and R. J. Walker, "Test code reuse from oss: Current and future challenges," in *Proceedings of the 3rd Africa and Middle East Conference on Software Engineering*. ACM, 2017, pp. 31–36.