Interpreting Coverage Information Using Direct and Indirect Coverage

Chen Huo
University of Delaware
Newark, DE, USA
huoc@udel.edu

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

***Abstract*—Because of the numerous benefits of tests, developers often wish their applications had more tests. Unfortunately, it is challenging to determine what new tests to add in order to improve the quality of the test suite. A number of approaches, including numerous coverage criteria, have been proposed by the research community to help developers focus their limited testing resources. However, coverage criteria often fall short of this goal because achieving 100% coverage is often infeasible, necessitating the difficult process of determining if a piece of uncovered code is actually executable, and the criteria do not take into account how the code is covered. In this paper, we propose a new approach for interpreting coverage information, based on the concepts of direct coverage and indirect coverage, that address these limitations. We also presents the results of an empirical study of 17 applications that demonstrate that indirectly covered code is common in real world software, faults in indirectly covered code are significantly less likely to be detected than faults located in directly covered code, and indirectly covered code typically clusters at the method level. This means that identifying indirectly covered methods can be effective at helping testers improve the quality of their test suites by directing them to insufficiently tested code.**

I. Introduction

Testing is one of the primary ways that software developers assess the quality of their software. Beyond assessing quality, tests have numerous other benefits such as enabling large scale changes and serving as a form of documentation. Because of their benefits, many developers have a strong desire for more tests in their projects [8]. However, writing additional tests can be both difficult and costly. According to a recent study of practicing software developers, a significant portion of this cost is due to the difficulty of identifying which parts of the code to test [8].

To help developers identify which code should be tested, researchers have proposed numerous code coverage metrics (e.g., [5, 6, 17, 24, 25, 30–32, 36]). In the context of testing, such coverage metrics indicate which entities (e.g., functions, statements, branches, etc.) in a program are executed (covered) by a test suite and which are not. Presumably, uncovered entities indicate deficiencies in a test suite—if an entity is not executed, the test suite can not determine whether it is correct—and identify code that is insufficiently tested.

In practice, several factors limit code coverage's effectiveness at helping developers identify where to focus their limited testing resources. First, it is often difficult or, in the case of infeasible code, impossible to cover every entity in a program. As a result, developers must spend significant amounts of time to determine whether it is possible for an uncovered piece of code to be executed. Second, code coverage does not consider how an entity is covered, only whether it is executed by the test suite. As we demonstrate in Section III, faults located in code that is *directly covered* are more likely to be detected than faults located in code that is *indirectly covered*. Intuitively, elements that are directly covered (e.g., methods with depth 1 in a test's dynamic call graph) are the focus of a test while elements that are indirectly covered (e.g., methods that are not directly covered by any test in a test suite) are only peripherally considered.

In this paper, we propose a new approach, based on the concepts of direct coverage and indirect coverage, for interpreting coverage information. The goal of the approach is to help developers focus their limited testing resources on insufficiently tested code. At a high-level, the approach identifies methods that contain a high proportion of indirectly covered entities as being insufficiently tested. In addition to taking into account how entities are covered by a test suite, the approach also eliminates the need for testers to manually identify whether the code indicated by the approach can be executed. Because they are covered by the test suite, the methods identified by the approach are guaranteed to be feasible. This means that developers do not have to spend time investigating whether it is possible to execute the identified code.

Our work makes the following contributions:

- The definition of a new approach to interpret coverage information based on the concepts of direct coverage and indirect coverage
- An empirical study on the statement coverage of 17 real applications and their test suites that demonstrates: (1) real test suites indirectly cover large portions of their corresponding applications, (2) faults located in code that is indirectly covered are significantly less likely to be detected than faults that are located in code that is directly covered, (3) the majority of methods are either completely directly covered or completely indirectly covered, and (4) a significant portion of indirectly covered methods are likely due to testers improperly considering inheritance or method overloading relations.

The rest of the paper is organized as follows: Section II describes the details of our approach for interpreting coverage information including formal definitions of direct coverage and indirect coverage, an algorithm to compute direct coverage and indirect coverage, and an example illustrating the approach. Section III presents our empirical study investigating various aspects of direct coverage and indirect coverage on 17 real-world software projects. Section IV discusses several areas of related work. Finally, Section V draws conclusions from the empirical study and discusses possible areas of future work.

II. Direct and Indirect Coverage

This section describes background information necessary for understanding the remainder of the paper. First, it defines the concepts of direct coverage and indirect coverage. Second, it describes how the direct coverage and indirect coverage of a test suite can be calculated. Third, it explains how insufficiently tested methods can be identified. Finally, it provides a concrete example of the concepts in terms of statement coverage.

A. Definitions

The concepts of direct coverage and indirect coverage are relatively simple: rather than identifying only *whether an entity is covered* by a test suite, direct and indirect coverage identify *how an entity is covered*. Formally, the concepts of coverage, direct coverage, and indirect coverage are defined as follows:

Definition 1. An entity e is *covered* by a test suite T iff there exists a test $t \in T$ such that e is executed by t.

Definition 2. An entity e is *directly covered* by a test suite T iff (1) e is covered by T, and (2) there exists a test $t \in T$ such that m, the method that contains e, is directly invoked by t (i.e., the depth of m in t's dynamic call graph is 1).

Definition 3. An entity e is *indirectly covered* by a test suite T iff e is (1) covered by T, (2) not directly covered by T, and (3) contained within a method that is publicly accessible.

B. Calculating Direct Coverage and Indirect Coverage

Algorithm 1 shows the procedure for computing the direct coverage and indirect coverage of a test suite. As input, the algorithm requires five pieces of information. The first is AUT, the application under test. The second is T, the test suite of the application under test. The third is $\mathrm{Coverage}$, a mapping that associates each test in the test suite to the set of entities that are covered when the test is executed. The fourth is Direct, a mapping that associates each test in the test suite to the set of methods that are directly invoked by the test (i.e., the set of methods that have a depth of 1 in the test's dynamic call graph). And the fifth is $\mathrm{MethodOf}$, a mapping that associates each entity in the application under test with the method that contains the entity. The application under test and the associated test suite are provided by the tester while the mappings can be computed using various straightforward static and dynamic analyses.

Algorithm 1 Compute the direct coverage and indirect coverage for a test suite.

Input: AUT, the application under test.
Input: T, the test suite of the application under test.
Input: $\mathrm{Coverage}$, a mapping that associates each test $t \in T$ to the set of entities covered by t.
Input: Direct, a mapping that associates each test $t \in T$ to the methods that have a depth of 1 in t's dynamic call graph.
Input: $\mathrm{MethodOf}$, a mapping that associates each entity e to the method that contains e.
Output: DC, a mapping that associates each method $m \in AUT$ to the set of entities in m that are directly covered.
Output: IC, a mapping that associates each method $m \in AUT$ to the set of entities in m that are indirectly covered.

```
1:  function COMPUTEICDC
2:    DC ← {}
3:    IC ← {}
4:    for t ∈ T do
5:      E ← Coverage[t]
6:      M ← E map {e ⇒ MethodOf[e]}
7:      (M_d, M_i) ← M partition {m ∈ Direct[t]}
8:      for m ∈ M_d do
9:        E_m ← E filter {e ⇒ m = MethodOf[e]}
10:       DC[m] ← DC[m] ∪ E_m
11:       IC[m] ← IC[m] \ E_m
12:     end for
13:     for m ∈ M_i do
14:       E_m ← E filter {e ⇒ m = MethodOf[e]}
15:       IC[m] ← IC[m] ∪ (E_m \ DC[m])
16:     end for
17:   end for
18:   return DC, IC
19: end function
```

As output, the algorithm produces two mappings, DC, which associates each method in the application under test to the set of entities contained in the method that are directly covered and, IC, which associates each method in the application under test to the set of entities contained in the method that are indirectly covered. Given DC and IC, it is straightforward to compute the set of entities directly covered or indirectly covered by the entire test suite.

Given the necessary inputs, the algorithm proceeds as follows. First the output mappings, DC and IC, are initialized to empty maps (Lines 2–3). Then the for loop from Line 4 to Line 17 iterates over each test contained in the application under test's test suite.

In the body of the loop Line 5 retrieves E, set of entities covered by the t from the Coverage mapping. Line 6 calculates M, the set of methods covered by test t, by transforming the set of entities covered by t to their containing method using MethodOf. Line 7 partitions M into two sets based on whether a method is directly invoked by t: M_d contains the methods that are directly invoked while M_i contains the methods that are indirectly invoked.

The for loop from Line 8 to Line 12 iterates over the directly covered methods to update the DC and IC mappings. Line 9 computes E_m, the subset of entities covered by t that are contained in method m, by filtering the set of all entities covered by t. In practice, instead of filtering the set of entities for each directly covered method, an additional mapping $M_{m \to E} \subseteq M_d \times \mathcal{P}(E)$ could be computed for all methods in M before Lines 8. Line 10 updates the DC mapping by adding E_m to m's directly covered entities. Conversely, Line 11 updates the IC mapping by removing E_m from m's indirectly covered entities. Recall that an entity is only indirectly covered if it is not directly covered by any test. As such, the entities contained in E_m are not indirectly covered.

The for loop from Line 13 to Line 16 iterates over the indirectly covered methods to update the DC and IC mappings. Line 14 again computes E_m, the subset of entities covered by t that are contained in method m, by filtering the set of all entities covered by t. Line 15 first removes all of the directly covered entities contained in m from E_m. Again, if an entity is directly covered, it can not be indirectly covered. Second, the IC mapping is updated by adding the resulting set to the set of entities indirectly covered in m.

Finally, at Line 18, after each test has been processed, the two mappings, DC and IC, are returned.

Note that, while the concepts of direct coverage and indirect coverage are agnostic to the type of entity that is being covered, in the remainder of the paper we will focus on statement coverage. We choose to focus on statement coverage because, due to its simplicity and availability of tool support, it is the most commonly used coverage metric in practice.

C. Identifying Insufficiently Tested Methods

Given the DC and IC mappings produced by Algorithm 1, it is possible to identify methods that are insufficiently tested by computing each method's direct and indirect coverage scores:

$$DirectCoverage(m) = \frac{|DC[m]|}{|DC[m]| + |IC[m]|}$$

$$IndirectCoverage(m) = \frac{|IC[m]|}{|DC[m]| + |IC[m]|}$$

Analogously to how traditional coverage scores can indicate insufficiently tested code by identifying areas where large numbers of entities are uncovered, direct and indirect coverage scores indicate insufficiently tested methods by identifying methods that have a small percentage of directly covered entities or a high percentage of indirectly covered entities.

```
 1. public class Example {
 2.
 3.   public int m1(int a, int b) {
 4.     return a + b + m4(a, b);
 5.   }
 6.
 7.   public int m2(int a, int b) {
 8.     return a + (2 / b);
 9.   }
10.
11.   public int m3(int a) {
12.     return m2(a, 2);
13.   }
14.
15.   public int m4(int a, int b) {
16.     return a - b;
17.   }
18. }
```

(a) Application under test.

```
public class ExampleTest {
  public void test1() {
    Example e = new Example();
    assertEquals(3, e.m1(1, 2));
  }

  public void test2() {
    Example e = new Example();
    assertEquals(3, e.m3(1) + e.m4(1, 2));
  }
}
```

(b) Corresponding test suite.

Fig. 1: Example code for illustrating direct coverage and indirect coverage.

D. Prototype implementation

The implementation of our technique consists of three components, a coverage profiler,[1] a call graph tracer[2], and a direct coverage calculator. The coverage profiler uses the T.J. Watson Libraries for Analysis (WALA), a static analysis framework developed by IBM, to find out the tests in applications. It then executes each tests individually and uses the Jacoco framework to record the test's coverage. The output of running each test is used to define the Coverage mapping explained in Section II-B. It then builds up the mapping MethodOf to relate the entities towards the methods with the help of WALA. Finally, the Direct mapping is built using the JVMTM Tool Interface (JVMTI) to track the direct invocations from the tests. Every time a method is invoked, the tracer will check if the caller is a test or some auxiliary method in the test suite. If so, the callee is considered directly covered. The tracer stores the execution traces for direct invocations. Finally, we implemented the algorithm described in Section II-B to compute the direct coverage and indirect coverage of the test suite.

E. An Illustrative Example

As a concrete example of computing direct statement coverage and indirect statement coverage, consider Figure 1 which

[1] Available at: https://bitbucket.org/huoc/icdc

[2] Available at: https://bitbucket.org/huoc/icdctracer

shows the four methods that constitute an application under test (Figure 1a) and the application under test's corresponding test suite (Figure 1b).

The input mappings, Coverage, Direct, and MethodOf for this example are shown below:

$$\text{Coverage} = \begin{cases} \text{test1} & \rightarrow \{\text{s4}, \text{s16}\} \\ \text{test2} & \rightarrow \{\text{s8}, \text{s12}, \text{s16}\} \end{cases}$$

$$\text{Direct} = \begin{cases} \text{test1} & \rightarrow \{\text{m1}\} \\ \text{test2} & \rightarrow \{\text{m3}, \text{m4}\} \end{cases}$$

$$\text{MethodOf} = \begin{cases} \text{s4} & \rightarrow \text{m1} \\ \text{s8} & \rightarrow \text{m2} \\ \text{s12} & \rightarrow \text{m3} \\ \text{s16} & \rightarrow \text{m4} \end{cases}$$

The coverage mapping indicates that test1 covers Statements 4 and 16 while test2 covers Statements 8, 12, and 16; the direct invocation mapping indicates that test1 directly invokes method m1 while test2 directly invokes methods m3 and m4; and the containing method mapping indicates that Statement 4 is contained in method m1, Statement 8 is contained in method m2, Statement 12 is contained in method m3, and Statement 16 is contained in method m4.

The DC and IC mappings computed by Algorithm 1 are as follows:

$$DC = \begin{cases} \text{m1} & \rightarrow \{\text{s4}\} \\ \text{m2} & \rightarrow \emptyset \\ \text{m3} & \rightarrow \{\text{s12}\} \\ \text{m4} & \rightarrow \{\text{s16}\} \end{cases}$$

$$IC = \begin{cases} \text{m1} & \rightarrow \emptyset \\ \text{m2} & \rightarrow \{\text{s8}\} \\ \text{m3} & \rightarrow \emptyset \\ \text{m4} & \rightarrow \emptyset \end{cases}$$

For this example, the mappings show that Statements 4, 12, and 16 are directly covered by the test suite, while Statement 8 is indirectly covered. Statement 8 is the only indirectly covered entity because all of the other statements are directly covered by either test1 or test2. Given this output, the direct coverage scores of methods m1, m2, and m4, are $100\,\%$ while the direct coverage score of method m3 is $0\,\%$. Although the test suite achieves $100\,\%$ statement coverage, the direct coverage scores suggest that method m2 may be insufficiently tested.

In this particular case, the fact that Statement 8 is never directly covered means that the potential division by zero error it contains can not be detected. Because m2 is only called by m3, the argument to m2 is always 2. By pointing out that method m2 has low proportion of directly covered statements, the approach guides testers to the portions of their applications that can benefit from additional testing.

TABLE I: Considered applications.

Subject	LoC	# Tests
Apache XML Security	20,396	52
Barbecue	4,129	172
Commons Beanutils	11,375	973
Commons CLI	1,978	187
Commons CLI2	11,231	470
Commons Collections	26,414	2,563
Commons IO	26,614	882
Commons Language	23,070	2,044
Crammer	20,034	185
Crystal VC	8,031	80
DecentXML	12,741	714
HTML Parser	31,216	764
HttpClient	48,994	894
JDom	16,154	1,257
JFreeChart	92,252	2,234
Joda-Time	86,797	3,962
Numerics4j	3,647	194

III. Empirical Study

We will conduct an empirical study that applies our proposed approach on real world applications. This section describes the design details of our empirical study, including the research questions and subject applications. This empirical study is designated to answer the following research questions:

RQ1—Presence Does indirect coverage exist in real world applications?

RQ2—Significance Is the possibility of finding a fault influenced by whether the code containing the fault is directly or indirectly covered?

RQ3—Distribution How are indirectly covered statements distributed throughout an application?

RQ4—Categorization What are the potential causes for indirectly covered methods?

A. Considered Applications

To investigate our research questions, we selected 17 Java applications with their associated developer-provided test suites as our research subjects. Table I lists the specific applications that we chose. The first column, *Subject*, shows the names of the selected projects. These projects were taken from a variety of open source repositories including: (1) the Software-artifact Infrastructure Repository (SIR),[3] which provides a variety of open-source projects for empirical software engineering, (2) SourceForge,[4] a popular repository for open–source projects, and (3) Apache Commons,[5] a collection of reusable components. The second column, *LoC*, shows the number of source lines of code in the Java files of each subject. The third column, *# Tests*, shows the number of tests included in each application's test suite.

We choose these specific applications for several reasons. First, in general, they are popular and widely used. Second, the applications cover a variety of subject domains. For

[3] http://sir.unl.edu

[4] https://sourceforge.net

[5] http://commons.apache.org

TABLE II: Direct and indirect coverage in percentage.

Subject	Coverage	% DC	% IC
Apache XML Security	42.4	79.4	20.6
Barbecue	84.0	75.3	24.7
Commons Beanutils	74.1	57.9	42.1
Commons CLI	93.1	66.2	33.8
Commons CLI2	95.7	71.3	28.7
Commons Collections	84.7	65.9	34.1
Commons IO	80.0	83.4	16.6
Commons Language	91.8	90.6	9.4
Crammer	59.2	53.7	46.3
Crystal VC	40.9	63.5	36.5
DecentXML	72.8	40.5	59.5
HTML Parser	61.1	50.3	49.7
HttpClient	69.2	77.8	22.2
JDom	71.2	77.1	22.9
JFreeChart	69.0	67.0	33.0
Joda-Time	89.1	84.4	15.6
Numerics4j	94.2	77.0	23.0

example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, Joda-Time is a library for handling dates and times, etc. Third, the applications vary in size. For example, JFreeChart has over 90,000 lines of code, while Commons CLI has approximately 2,000 lines of code. Finally, the test suites also vary in size. The test suites for some of the applications contain nearly 4,000 tests while others contain fewer than 100. Selecting test suites and applications of various sizes and subject domains improves the generalizability of our results.

B. Presence

The purpose of our first research question is to determine how the statements in our subject applications are covered by the test suites. Because we are proposing to identify insufficiently tested code based on indirect coverage, it is important to understand how commonly indirectly covered code occurs in real applications. To answer this question, we first computed the overall direct and indirect coverage scores for each of our subject applications. The results of this computation are shown in Table II.

In Table II, the first column, *Subject*, again shows the names of our experimental subjects. The second column, *Coverage*, shows the overall statement coverage achieved by the application's test suite. The third and fourth columns, *% DC* and *% IC*, show the application's direct coverage score and indirect coverage score, respectively. That is, of the covered statements, the percentage that are directly covered and the percentage that are indirectly covered. The data shows that the percentage of indirect coverage ranges roughly from 10 % to 60 %. This suggests that, even for real test suites, the proportion of indirectly covered code in an application can be significant.

To gain some additional insights into this data, we checked whether there is any correlation between an application's statement coverage score and the percentage of statements that are indirectly covered. To compute the correlation, we used R version 2.14.1's implementation of the Pearson correlation coefficient. The computed correlation coefficient is -0.35 which indicates a very weak negative correlation. That means that, in practice, it is not possible to infer the amounts of direct or indirect coverage by considering only the traditional coverage score. It is necessary to compute direct coverage and indirect coverage scores directly.

C. Significance

The purpose of our second research question is to determine whether how a statement is covered impacts the effectiveness of the test suite. More specifically, we are interested in knowing if faults located in indirectly covered statements are less likely to be detected by the application's test suite than faults located in directly covered statements.

Because it is difficult to identify a suitable number of real faults with the necessary uniform distribution among directly and indirectly covered code, we chose to consider injected faults. More specifically, we considered mutants. Although mutants are artificial, recent work has shown that they can be a valid substitute for real faults [3, 19].

To generate the necessary mutants for our subjects, we used the MAJOR framework.[6] While MAJOR is a state of the art mutation testing framework, it was unable to complete the mutation testing process for five of our subject applications.

As part of performing mutation testing, MAJOR collects several pieces of useful information for each mutant it generates, including: the location of the mutant, in terms of the containing class and method names and the line number; the mutation operator used to generate the mutant; and whether the mutant was detected by the application's test suite (i.e., if the mutant was killed). Because we know the location of the mutant, we can identify whether each mutant is located in directly covered code or indirectly covered code.

To determine whether there is a significant difference in the ability of a test suite to detect a fault depending on how the fault is covered, we used the binomial test. As an example of how to compute the test, let N_d be the number of mutants located in directly covered code, N_i be the number of mutants located in indirectly covered code, K_d be the number of mutants located in directly covered code that are killed by the test suite and K_i be the number of mutants located in indirectly covered code that are killed by the test suite. Then the binomial distribution $B(N_d, K_i/N_i)$ is used to calculate the probability of K_d or more kills in a sample of size of N_d, given the assumption that the probability of killing a mutant is K_i/N_i. Informally, our null hypothesis is that the location of the mutant does not affect the likelihood that it is killed. We used R version 2.14.1's implementation of the test (i.e., `binom.test`) with the one-sided option (i.e., `alternative="greater"`).

Table III shows the twelve subjects for which MAJOR was able to generate mutants. The first column, *subject*, shows the name of each subject. The second and third columns show the

[6] http://mutation-testing.org

TABLE III: Mutants covered and killed.

Subject	# Mutants		# Killed		Mutation Score			
	IC	DC	IC	DC	IC	DC	% Change	p value
Barbecue	81	668	35	362	43.2	54.2	11.0	7.7×10^{-9}
Commons Beanutils	567	1,130	292	678	51.5	60.0	7.5	5.6×10^{-9}
Commons CLI	185	453	135	357	73.0	78.8	5.8	2.5×10^{-3}
Commons CLI2	476	1,110	309	911	64.9	82.1	17.2	2.2×10^{-16}
Commons Collections	1,150	4,011	712	3,004	61.9	74.9	13.0	2.2×10^{-16}
Commons IO	519	3,570	394	3,000	75.9	84.0	8.1	2.2×10^{-16}
Commons Language	1,031	12,575	726	9,496	70.4	75.5	5.1	2.2×10^{-16}
HTMLParser	2,163	2,443	1,220	1,558	56.4	63.8	7.4	7.3×10^{-14}
JDom	491	1,059	298	703	60.7	66.4	5.7	7.4×10^{-5}
Joda Time	693	2,111	465	1,532	67.1	72.6	5.5	3.2×10^{-8}
JFreeChart	10,001	16,681	2,915	9,171	29.2	55.0	25.8	2.2×10^{-16}
Numerics4j	65	665	35	484	30.8	72.5	41.7	2.2×10^{-16}

number of mutants generated by MAJOR that are located in indirectly covered code (*IC*) and the number that are located in directly covered code (*DC*). The fourth and fifth columns show the number of mutants, located in either directly covered code (*DC*) or indirectly covered code (*IC*) that were detected by the application's test suite. The sixth and seventh columns show the mutation scores for indirectly covered mutants (*IC*) and directly covered mutants (*DC*), that is the ratio of killed mutants to total mutants. The eighth column, *% Change* shows the percentage change in mutation score when comparing the mutation score for mutants located in indirectly covered code to the mutation score for mutants located in directly covered code. Finally, the last column, *p value*, shows the p value computed by the binomial test for each subject.[7]

As the data shows, for all twelve subjects, there is a statistically significant difference in the likelihood that a mutant is killed depending on how the mutant is covered by the test suite. Moreover, the percentage change in mutation score ranges from $5.5\,\%$ to $41.7\,\%$. This means that not only is the effect of how a mutant is covered significant, the size of the effect can be large as well. These results support our assumption that indirectly covered code is less effectively tested than directly covered code and should be brought to the attention of testers.

D. Distribution

The purpose of our third research question is to learn about the distribution of indirectly covered statements in each of our applications. By learning how indirectly covered statements are distributed, we can determine the appropriate level at which to report insufficiently covered code to developers. If indirectly covered statements are evenly distributed, then reporting them individually is the only option. However, if indirectly covered statements are clustered, reporting indirectly covered code at a higher granularity can be more useful.

To investigate how indirectly covered statements are distributed, we computed the indirect coverage scores of each method in each subject application. The results of this calculation are shown in Figure 2. This figure shows several plots, one for each subject, and one, *(all)*, for all applications together. Each individual plot presents a histogram that shows the distribution of indirectly covered statements in the application. The y-axis shows the percentage of indirect statement coverage grouped into three bins: $0\,\%$ indirect coverage, $1\,\%$ and $99\,\%$ indirect coverage, and $100\,\%$ indirect coverage. The y-axis shows the percentage of methods whose indirect coverage score falls within the corresponding bin. For example, for apache-xml-security, approximately $20\,\%$ of its methods are completely directly covered, approximately $3\,\%$ of its methods have indirect coverage scores between $1\,\%$ and $99\,\%$, and approximately $77\,\%$ of its methods are completely indirectly covered.

As the data shows, for all the applications, the distribution of the methods is primarily binary. In general, methods are either completely directly covered or completely indirectly covered. Across all applications, less than $3\,\%$ of the methods have indirect coverage scores between $1\,\%$ and $99\,\%$. This suggests that reporting indirectly covered statements at a method-level will be effective at helping guide testers to indirectly covered code.

E. Categorization

The purpose of our fourth research question is to determine the potential causes of indirectly covered code. Understanding why indirectly covered code occurs in our applications can help testers preemptively address why the code is indirectly covered and improve their test suites.

To determine the causes of the indirectly covered code, we manually investigated all of the methods in our subjects with at least some percentage of indirectly covered code. As a result of this investigation, we classified the methods into three groups: Overloading, Inheritance, and Other. The Overloading group contains methods that appear to be indirectly covered due to overloading among methods in the same class. For example, if a class contains more than one method with the same name and return type and at least one of the overloaded methods is directly covered, we consider the cause of any indirectly covered overloaded methods to be Overloading. Similarly, the Inheritance group contains methods that appear to be indirectly covered due to inheritance among subclasses and super classes.

[7] 2.2×10^{-16} is the minimum value can be shown by R in the binomial test.

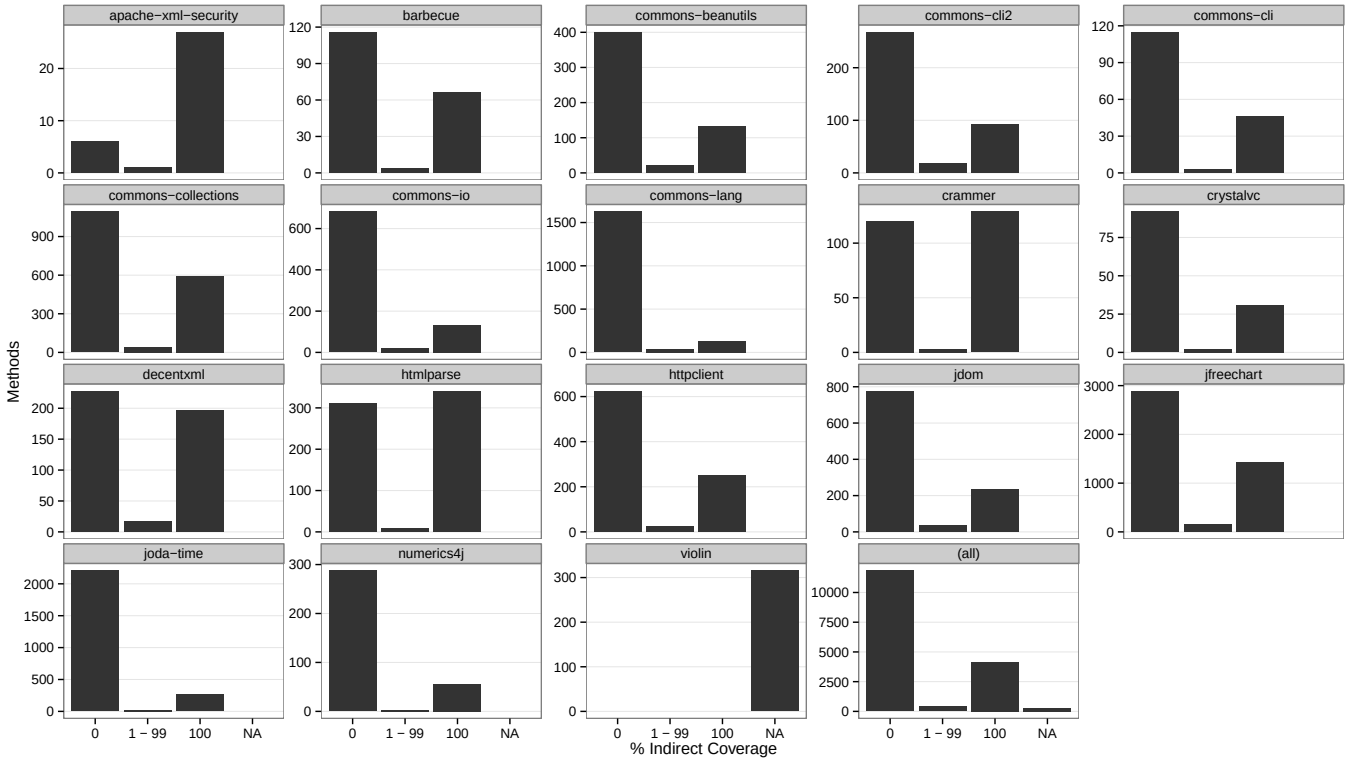

Fig. 2: The distribution of indirectly covered statements of each application.

For example, if super class contains a method that is indirectly covered, while a subclass's implementation of the method is directly covered, we consider the cause of the indirectly covered method to be Inheritance. Finally, the last group Other contains methods that are indirectly covered for another reason (e.g., a tester may simply have forgotten to directly test the method).

The results of this classification are shown in Tables IV and V. Table IV shows the results for methods whose percentage of indirectly covered statements is between $1\,\%$ and $99\,\%$ while Table V shows the results for methods that are completely indirectly covered. In each table, the first column, *Subject* shows the name of each subject. The second column, *Total*, shows the number of indirectly covered methods. The third and fourth columns, *Overloading*, show the total number of methods in the Overloading category (*# Methods*) and the total number of unique method names (*# Unique*). For example, for Commons Beanutils, there are 14 methods in the Overloading category, but two of them have the same name as another indirectly covered method. The fifth and sixth columns, *Inheritance* show the same information for the methods in the Inheritance group. Finally, the last column, *Other*, shows the number of methods in the Other category.

As the data shows, for methods with partial indirect coverage, $53\,\%$ are caused by either overloading or inheritance and for methods with $100\,\%$ indirect coverage, $40\,\%$ are caused by either overloading or inheritance. This suggests that both overloading and inheritance are common causes of indirect coverage and should be given special attention in the testing process.

```
// 0% Indirect coverage
public static FileOutputStream openOutputStream(File
file) throws IOException {
  return openOutputStream(file, false);
}

// 100% Indirect coverage
public static FileOutputStream openOutputStream(File
file, boolean append) throws IOException {
  return new FileOutputStream(file, append);
}
```

Fig. 3: An example for an overloading group where one method is completely indirectly covered.

In the remainder of this section, we will provide specific examples of methods in the overloading and inheritance categories and explain why they are likely to be insufficiently tested.

Figure 3 shows an excerpt of an overloading group where one method is directly covered and another is completely indirectly covered. The method `openOutputStream` has two variants, one that accepts a File and a boolean as input (`openOutputStream(File, boolean)`) and one that accepts only a File as input (`openOutputStream(File)`). The single argument variant delegates to the multiple argument variant by supplying a default boolean parameter. While the single argument variant is directly covered by the associated

TABLE IV: Methods with partial indirect covereage.

Subject	Total	Overloading		Inheritance		Other
		# Methods	# Unique	# Methods	# Unique	
Apache XML Security	2	0	0	0	0	2
Barbecue	5	1	1	1	1	3
Commons Beanutils	23	14	12	3	3	6
Commons CLI	4	2	2	0	0	2
Commons CLI2	20	5	5	4	4	11
Commons Collections	43	11	9	7	7	25
Commons IO	22	16	15	1	1	5
Commons Language	37	30	26	6	5	1
Crammer	3	2	2	0	0	1
Crystal VC	2	0	0	0	0	2
Decent XML	18	9	9	2	2	7
HTML Parser	9	4	4	0	0	5
HttpClient	25	3	3	4	4	18
JDom	36	14	10	1	1	21
JFreeChart	126	32	30	31	15	63
Joda Time	24	9	8	2	2	13
Numerics4j	4	1	1	0	0	3
Total	403	153	137	62	45	188

TABLE V: Methods with full indirect covereage.

Subject	Total	Overloading		Inheritance		Other
		# Methods	# Unique	# Methods	# Unique	
Apache XML Security	27	0	0	2	1	25
Barbecue	116	12	8	33	11	71
Commons Beanutils	132	80	47	22	14	30
Commons CLI	46	16	10	6	2	24
Commons CLI2	93	27	17	10	8	56
Commons Collections	597	128	79	101	53	368
Commons IO	133	73	47	5	4	55
Commons Language	133	46	20	6	5	81
Crammer	129	24	18	6	2	99
Crystal VC	32	2	1	1	1	29
Decent XML	228	56	47	16	6	156
HTML Parser	313	79	50	54	35	180
HttpClient	252	62	43	21	15	169
JDom	223	79	32	15	10	129
JFreeChart	1,443	347	254	139	80	957
Joda Time	263	93	51	55	28	115
Numerics4j	56	40	39	3	3	13
Total	4,216	1,164	756	495	276	2,557

test suite, the two argument variant is never directly covered. Its indirect coverage score is $100\,\%$. In this case, the tester cannot access the boolean parameter and any failures that depend on the parameter being true can not be found by the test suite.

Figure 4 shows an excerpt of an overloading group where one method is directly covered and the other is partially indirectly covered. The method `toInt` has two variants, one that accepts a String and an int as parameters and one that accepts only a String and delegates to the two argument variant by supplying a default int value. Unlike the previous example, the application's test suite does directly cover some of the statements in the two argument variant by calling it directly in `testToIntStringI`. However, because the calls to `toInt` in this test never pass in a null value for the String parameter, the method never returns the provided default value. Again, any failures related to this code path can not be detected by the test suite.

Figure 5 shows an excerpt of an inheritance group where one method is directly covered and the other is completely indirectly covered. The implementation of `widthInBars()` declared in the Module class, a concrete and public class, is never tested directly by the test suite while the implementation declared in the `CompositeModule` subclass is directly covered.

Figure 6 shows an excerpt of an inheritance group with partial indirect coverage. The class `ParentImpl` is the base of many subclasses whose method `canProcess` has many variants. The `ParentImpl` has not been explicitly constructed, however, `ParentImpl.canProcess` has partial direct coverage. Unlike `Module` in the previous example, there are some subclasses that do not override `canProcess`. So when these subclasses get tested in the test suite, part of the method got directly covered.

```java
// 0% Indirect coverage
public static int toInt(String str) {
  return toInt(str, 0);
}

// 20% Indirect coverage
public static int toInt(String str, int defaultValue) {
  if(str == null) {
    return defaultValue;
  }
  try {
    return Integer.parseInt(str);
  } catch (NumberFormatException nfe) {
    return defaultValue;
  }
}
```

(a) Application under test.

```java
public void testToIntString() {
  assertTrue(NumberUtils.toInt("12345") == 12345);
  assertTrue(NumberUtils.toInt("abc") == 0);
  assertTrue(NumberUtils.toInt("") == 0);
  assertTrue(NumberUtils.toInt(null) == 0);
}

public void testToIntStringI() {
  assertTrue(NumberUtils.toInt("12345", 5) == 12345);
  assertTrue(NumberUtils.toInt("1234.5", 5) == 5);
}
```

(b) Corresponding test suite.

Fig. 4: An example for an overloading group where one method is partially indirectly covered.

We can see from the later two examples for inheritance groups that it is hard for test programers clearify which siblings/ancestors/offsprings have not yet been used as the test inputs because of the various builders and dynamic binding for the big family of classes.

IV. Related Work

This sections presents existing work that is related to ours and organized into categories.

A. Defect Prediction

The first group of related work is defect prediction approaches that attempt to model various software features in order to predict where defects are located before the defects lead to failures. A wide range of prediction models based on various features have been proposed including size and complexity models (e.g., [2]), development quality data models (e.g., [9, 14, 18]), history defect models (e.g., [15, 21]), multilinear regression models based on multiple metrics (e.g., [20, 28]). The defect prediction approaches that are most closely related to our approach are the ones proposed by Miller et al. and Voas and Miller.

Miller et al. proposed an approach to estimate the probability of a fault even when testing reveals no failures [26]. The authors provided a probability density function to predict the true probability of failure. With this function, Voas and Miller proposed *testability* which can be measured statically

```java
// Module
// 100% Indirect coverage
public int widthInBars() {
  int sum = 0;
  for (int i = 0; i < bars.length; i++) {
    sum += bars[i];
  }
  return sum;
}

// CompositeModule extends Module
// 0% Indirect coverage
public int widthInBars() {
  int width = 0;
  for (Iterator iterator = modules.iterator();
       iterator.hasNext();) {
    Module module = (Module) iterator.next();
    width += module.widthInBars();
  }
  return width;
}
```

(a) Application under test.

```java
protected Module getPreAmble() {
  CompositeModule module = new CompositeModule();
  if(drawingQuietSection) {
    module.add(QUIET_SECTION);
  }
  module.add(START[mode]);
  return module;
}

public void testQuietZoneWidthIsAtLeast10BarWidths() {
  assertTrue(barcode.getPreAmble().widthInBars() > 10);
}
```

(b) Corresponding test suite.

Fig. 5: An example for an inheritance group where one method is completely indirectly covered.

```java
//class ParentImpl
public boolean canProcess(WriteableCommandLine cl,
                          String arg) {
  final Set triggers = getTriggers();

  if (argument != null) {
    char separator = argument.getInitialSeparator();
    // if there is a valid separator character
    if (separator != NUL) {
      final int initialIndex = arg.indexOf(separator);
      // if there is a separator present
      if (initialIndex > 0) {
        return triggers.contains(arg.substring(0,
                                 initialIndex));
      }
    }
  }
  return triggers.contains(arg);
}

// DefaultOption extends ParentImpl
public boolean canProcess(WriteableCommandLine cl,
                          String argument) {
  return (argument != null) &&
         (super.canProcess(cl, argument) ||
         ((argument.length() >= burstLength) &&
         burstAliases.contains(argument.substring(0,
                               burstLength))));
}
```

Fig. 6: An example for an inheritance group where one method is partially indirectly covered.

even before testing has taken place to predict the presence or the absence of defects [39]. The authors stated that different software systems have different sensitivities towards testing due to their structures that can be modeled by two kinds of information loss: implicit information loss and explicit information loss. Implicit information loss occurs when two or more different inputs to a function produce the same output. The authors defined the domain/range ratio (DRR), as the ratio between the cardinality of the domain of the specification and the cardinality of the range of the specification. For example, while a function that checks whether a integer is odd can accept any integer as input, it only has two possible outputs so its DRR is $\infty : 2$. Conversely, a function that negates a boolean value has a DRR of $2 : 2$. The assumption is that function with higher DRR values are more likely to contain undetected failures.

B. Coverage Criteria

Also related to our approach are the various coverage criteria that have been well studied [17, 29, 35, 40]. Late in the 70's, Goodenough and Gerhart proposed *reliability* and *validity* to guarantee the correctness of the software [12, 13]. Reliability requires tests to consistently produce the same outputs. Validity requires tests to reveal errors if there are. However, neither of the two requirements can be shown practically applicable [16]. So the research community keeps searching for new test metrics and also new ways to interpret coverage metrics.

Schuler and Zeller proposed checked coverage as an approach for assessing oracle quality [34]. The checked coverage of a test or test suite is the ratio of executed statements that compute values that are checked by the test to the total number of executed statements. A low checked coverage score suggests that a test is likely to be missing assertions. State coverage, originally proposed by Koster [22, 23] and extended by Vanoverberghe et al. [37] is similar to checked coverage. The primary difference is that state coverage is the ratio of executed output defining statements, the statements that define a variable that can be checked by the test suite. Both of the two approaches try to find *missing assertions* based on, however, the inputs that have already been in the original tests. They both assume that the additional assertions would be placed on the expected entity to be tested which means they are only searching locally within the test with existing inputs. In contrast, our technique is looking for potential new tests (i.e., both inputs and assertions) globally in the whole software. In our terminology, they focus on the direct coverage of the test suite. Moreover, both of the coverage coverage and the state coverage involve slicing which is quite expensive.

C. Automatic Test Generation

As our approach can give suggestions for what to test next, automated test generation approaches can be used to actually generated the missing test cases. The family contains different techniques focusing on different facets on automatic test generation, including (1) generating test inputs effectively and efficiently, (e.g., [38]), (2) generating test oracles effectively and efficiently, (e.g., [1]), (3) using mutation to expose the vulnerability of the system, (e.g., [7, 10]), and (4) generating tests directed by coverage information, (e.g., [11, 33, 41]). The techniques that are guided by coverage information may benefit most from our technique. Although we do not provide an automatic approach about how to generate new tests, e.g., what the test inputs and expected values will be, the information about direct coverage can be used to guide the generation process.

D. Relative Coverage

Finally, other ways of interpreting coverage information have been proposed. Relative coverage, proposed by Bartolini et al. gives another way of interpreting the adequacy criteria [4]. It suggests, that instead of measuring the covered entities throughout the test suite, it is more appropriate to count the entities covered for a certain purpose. The target entities will be specified and the coverage is the ratio of covered targeted entities among the targeted entities. For example, the measurement for newly added functionalities in regression testing [27]. In this case, the targeted entities are the newly added operations.

V. Conclusions and Future Work

In this paper we presented a new approach for interpreting coverage information to identify insufficiently tested methods. The technique is based on partitioning the set of covered entities into entities that are directly covered and entities that are indirectly covered. We also presented the results of an empirical study of 17 applications that demonstrates: (1) real test suites indirectly cover large portions of their corresponding applications, (2) faults located in code that is indirectly covered are significantly less likely to be detected than faults that are located in code that is directly covered, (3) the majority of methods are either completely directly covered or completely indirectly covered, and (4) a significant portion of indirectly covered methods are likely due to testers improperly considering inheritance or method overloading relations. As a result, we believe that identifying indirectly covered methods can be an effective approach for helping testers improve the quality of their test suites by directing them to insufficiently tested code.

In future work, we plan to investigate the insufficiently tested methods that our tool identified in more detail in order to expand the categorization of these methods. In addition, we will implement an automated tool for generating recommendations for possibly missing test inputs or oracles of the identified methods. Finally, we will extend our empirical evaluation to consider additional coverage metrics (e.g., branch coverage).

VI. Acknowledgments

This work is supported in part by National Science Foundation Grant No. 1527093.

VII. References

[1] K. Aggarwal, Y. Singh, A. Kaur, and O. Sangwan. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–6, 2004.

[2] F. Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[4] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti. Whitening SOA testing. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 161–170, 2009.

[5] B. Benware, C. Schuermyer, S. Ranganathan, R. Madge, P. Krishnamurthy, N. Tamarapalli, K.-H. Tsai, and J. Rajski. Impact of multiple-detect test patterns on product quality. In *null*, page 1031. IEEE, 2003.

[6] S. Chakravarty, Y. Chang, H. Hoang, S. Jayaraman, S. Picano, C. Prunty, E. W. Savage, R. Sheikh, E. N. Tran, and K. Wee. Experimental evaluation of bridge patterns for a high performance microprocessor. In *null*, pages 337–342. IEEE, 2005.

[7] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[8] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *Proceedings of the 2014 International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[9] M. Dyer. *The cleanroom approach to quality software development*. John Wiley & Sons, Inc., 1992.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, 2012.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[12] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *Software Engineering, IEEE Transactions on*, (2):156–173, 1975.

[13] J. B. Goodenough and S. L. Gerhart. Toward a theory of testing: Data selection criteria. *Current trends in programming methodology*, 2(2): 44–79, 1977.

[14] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, 2009.

[15] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, 2005.

[16] W. E. Howden. Reliability of the path analysis testing strategy. *Software Engineering, IEEE Transactions on*, (3):208–215, 1976.

[17] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7 (3):113–128, Sept. 1975.

[18] C. Jones. The pragmatics of software process improvements. *Software Process Newsletter*, 3(5), 1996.

[19] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[20] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[21] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007.

[22] K. Koster. A state coverage tool for JUnit. In *Companion of the 30th International Conference on Software Engineering*, pages 965–966, 2008.

[23] K. Koster and D. C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 541–544, 2007.

[24] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *Software Engineering, IEEE Transactions on*, (3):347–354, 1983.

[25] E. J. Mccluskey and C.-W. Tseng. Stuck-fault tests vs. actual defects. In *Test Conference, 2000. Proceedings. International*, pages 336–342. IEEE, 2000.

[26] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas. Estimating the probability of failure when testing reveals no failures. *Software Engineering, IEEE Transactions on*, 18(1):33–43, 1992.

[27] B. Miranda. A proposal for revisiting coverage testing metrics. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 899–902, 2014.

[28] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[29] J. P. Myers, Jr. Adaptive approaches to structural software testing (abstract only). In *Proceedings of the 15th Annual Conference on Computer Science*, pages 432–, 1987.

[30] S. C. Ntafos. On required element testing. *Software Engineering, IEEE Transactions on*, (6):795–803, 1984.

[31] W. Qiu, J. Wang, D. Walker, D. Reddy, X. Lu, Z. Li, W. Shi, and H. Balachandran. K longest paths per gate (klpg) test generation for scan-based sequential circuits. In *Test Conference, 2004. Proceedings. ITC 2004. International*, pages 223–231. IEEE, 2004.

[32] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, (4):367–375, 1985.

[33] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[34] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

[35] A. Sen. Concurrency-oriented verification and coverage of system-level designs. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):37:1–37:25, Oct. 2011.

[36] C.-W. Tseng, S. Mitra, S. Davidson, and E. J. McCluskey. An evaluation of pseudo random testing for detecting real defects. In *VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001*, pages 404–409. IEEE, 2001.

[37] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, pages 542–553, 2012.

[38] W. Visser, C. S. Pâsâreanu, and S. Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4): 97–107, 2004.

[39] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE software*, (3):17–28, 1995.

[40] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 1–10, 1991.

[41] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing-EDCC 5*, pages 281–292. 2005.