Measuring and Mitigating Gaps in Structural Testing

Soneya Binta Hossain
Department of Computer Science
University of Virginia
Charlottesville, USA
sh7hv@virginia.edu

Matthew B. Dwyer
Department of Computer Science
University of Virginia
Charlottesville, USA
matthewbdwyer@virginia.edu

Sebastian Elbaum
Department of Computer Science
University of Virginia
selbaum@virginia.edu

Anh Nguyen-Tuong
Department of Computer Science
University of Virginia
an7s@virginia.edu

***Abstract*—Structural code coverage is a popular test adequacy metric that measures the percentage of program structure (e.g., statement, branch, decision) executed by a test suite. While structural coverage has several benefits, previous studies showed that code coverage is not a good indicator of a test suite's fault-detection effectiveness as code coverage does not consider test assertion quality. In this research, we formally define the *coverage gap* in structural testing as the percentage of program structure that is executed but not observed by any test oracle. Our large-scale empirical study of 13 Java applications, 16K test cases, and 51.6K test assertions shows that even for mature test suites, the gap can be as high as 51 percentage points (*pp*) and is 34 *pp* on average. Our study shows that the coverage gap strongly and negatively correlates with a test suite's fault-detecting effectiveness. We propose a lightweight static analysis of program dependencies related to the gap to produce a ranked recommendation of test focus methods that can improve test suite quality. When considering 34.8K assertions in the test suite as ground truth, the recommender suggests two-thirds of the focus methods written by testers within the top 5 recommendations.**

***Index Terms*—code coverage, test oracles, fault-detection effectiveness, checked coverage, mutation testing**

I. Introduction

Structural code coverage is a popular test adequacy metric that reports the percentage of a program's structural code elements (e.g., statement, branch, condition, decision) executed by a test suite. Among its strengths, code coverage is easy to interpret, can be integrated into build processes, incurs only modest runtime overhead, and many tools support its consumption by software engineers [4], [25]. For these reasons, code coverage tools are used by millions of developers in thousands of organizations on a daily basis, e.g., [8], [36].

Despite its popularity, code coverage has well-understood limitations. More than three decades ago, research on software fault modeling, e.g., [46], established that for a program execution to reveal a faulty statement, four conditions must be met: (**C1**) the statement must be executed; (**C2**) the execution of the statement must create an error state; (**C3**) the error state must propagate to program output; and (**C4**) the erroneous output must be observed and be judged to be incorrect. Covering a faulty statement is essential, but it only addresses the first of these conditions, which likely contributes to research findings demonstrating that coverage alone is inadequate for explaining fault-detection effectiveness [5], [24].

Research has clearly demonstrated the importance of **C4** by showing that the number and strength of assertions in test oracles influences fault-detection effectiveness [51]. Moreover, Schuler and Zeller developed *checked coverage* (CC) which, at least partially, accounts for conditions **C1**, **C2**, and **C4** by considering whether a statement can influence the oracle through a chain of dependencies [40]. In this way, it identifies statements that were *covered* by the test suite *but unobserved* by a test oracle. Covered but unobserved statements are incapable of revealing faults relative to an explicit oracle and their inclusion in an adequacy metric *overestimates* the quality of a test suite.

In this paper, we adapt the checked coverage framework to support a richer coverage criterion – object branch coverage [12] – and to make it directly comparable to traditional structural coverage measures. This allows for computation of the *checked coverage gap* – the portion of the coverage domain that is covered by a test suite, but unobserved by a test oracle. We leverage these changes to show through a controlled experiment using 13 real-world Java programs with substantial test suites that the size of the coverage gap ranges from 19% to 51% – 34% on average. We characterize the consequences of the coverage gap through a study using mutation testing that reveals that the size of the coverage gap is strongly and negatively related to fault detection effectiveness. Finally, to mitigate the negative consequences of the coverage gap, we propose a method that analyzes control and data dependencies to recommend functions to include as the focus method in a test oracle. Conceptually, this recommender suggests focus methods that can establish missing links in establishing conditions **C3** and **C4** from the description above. By experimentally removing assertions in existing test suites, we show that the original focus methods that close the coverage gaps are recommended 50%, 67%, and 73% times within top-1, top-5, and top-10 recommendations, respectively.

The primary contributions of this paper are:

(1) Confirming the findings of Schuler and Zeller's checked

coverage paper [40], [41].

(2) Demonstrating that for real-world Java systems traditional coverage criteria significantly overestimate test suite quality by quantifying the gap between traditional statement and object-branch criteria and their checked variants.

(3) Demonstrating that the size of the checked coverage gap is strongly and negatively related to fault-detection effectiveness.

(4) Proposing a method to recommend test oracle focus methods that can close the coverage gap and demonstrating its effectiveness.

II. Background and Related Work

This section reviews program slicing, structural coverage, and methods for assessing test suite effectiveness. It concludes with a discussion of the most closely related research.

A. Program Slicing

Program slicing analyzes a program's data flow and control flow dependencies and computes a set of statements (program slice) that may influence a designated set of values at some point of interest in the program – known as the slicing criterion [48]. Slicing can be divided along two axes: static vs. dynamic and forward vs. backward [45]. Forward slicing computes statements that the criterion affects and backward slicing computes statements that affect the criterion. Static backward slicing computes all statements that *may* affect the slicing criterion on some program execution, whereas, dynamic backward slicing computes statements that *actually* affect the slicing criterion on a specific program execution [1]. Slicing can easily be adapted to different representations of a program, e.g., source, bytecode, or assembler. In this work we use dynamic backward slicing on Java programs represented in bytecode.

B. Testing and Structural Coverage

A test consists of the definition of a set of *test inputs* and a predicate, called the *test oracle*, which is true if the test is consistent with program requirements [3]. Oracles are commonly expressed as `assert` statements whose arguments express those predicates. A test suite is a set of tests.

A key question in judging a test suite is the extent to which the set of test inputs force the execution of different parts of the software. Coverage criteria are used to make such judgements. Many different structural coverage criteria have been developed that vary in the definition of their *coverage domain*, e.g., [38]. The coverage domain defines a set of *elements* which capture the structure of a program. For example, coverage elements may be functions, statements, branch outcomes, pairs of definitions and uses, or program paths. A widely used structural criterion is *statement coverage* (SC) which is efficient to compute and whose results are easy to interpret, e.g., [36]. Coverage criteria naturally define an *adequacy* requirement which states that all feasible elements of the coverage domain must be executed by a test suite. One coverage criterion, c, is said to be *stronger* than another, c', if adequacy for c implies adequacy for c'. For inadequate test suites, coverage is usually expressed as the percentage of the coverage domain that was executed by a test suite. For inadequate tests, one must take care in comparing coverage percentages, because the strength relation does not apply, and changes in the coverage domain can shift percentages significantly, e.g., expressing coverage for statements versus basic blocks.

For critical systems, a number of stronger structural criteria have been developed, e.g., MC/DC [6]. In this paper, we choose *object branch coverage* (OBC) as a representative of such criteria [12]. OBC defines a coverage domain consisting of the outcomes of all object code branch instructions. For a simple decision, source branch and object branch coverage requirements are the same. However, for compound conditions in languages with short-circuit evaluation of logical expressions, OBC is much stronger. It has been shown that OBC is stronger than source branch, condition, and condition/decision coverage, and for simple and commonly met restrictions on programs, it is stronger than MC/DC [12].

C. Test Effectiveness

Test effectiveness refers to the fault-detection effectiveness of a test suite. A widely used approach to measure effectiveness is *mutation testing* which injects artificial faults in a program through minor modifications, such as changing conditional boundary or altering arithmetic operators. A modified program (*mutant*) is killed if any test detects it. The *mutation score* measures the percentage of mutants killed by a test suite. Mutation testing has proven to be a good indicator of test suite fault-detection ability [2], [33]. In our research, we use the PIT mutation tool for the Java programs [11].

D. Test Oracles

Oracles play a fundamental role in testing - without them, a test is incapable of making a judgement about the exercised program behavior [3], [43]. Because of their centrality, there is a long history of research on how to derive test oracles from requirements specifications, e.g., [34], [39], [44], and the interest in generating oracle assertions continues with recent work proposing methods that learn to generate test assertions from examples, e.g., [47]. Despite this progress, most test oracles in use today are either implicit, i.e., checks imposed by the runtime system, or written by developers.

Developer effort in writing tests, and encoding oracles, is valuable, and researchers have studied a variety of methods for leveraging existing test oracles to support different software testing sub-problems: test data selection [50], test case prioritization [42], and test coverage [40], [49] – which we discuss in detail below.

A primary motivation in our work is the growing body of research demonstrating the strong link between oracle quality and fault-detection effectiveness. Of the recent work in this area, Zhang and Mesbah's study of the relationship between assertions and test effectiveness is noteworthy [51]. They found that the number of assertions is strongly correlated with

test suite effectiveness and that checked coverage, discussed below, is more strongly correlated with test suite effectiveness than traditional coverage metrics. They also found that developer-written assertions are more effective than automated assertions.

E. Oracle-Based Coverage

In 2007, Ken Koster and David Kao proposed a structural test adequacy metric called *state coverage* that measures the percentage of output-defining and side effect variables checked by test oracles [35]. To the best of our knowledge, state coverage is the first paper that introduced a structural test adequacy metric that considers the use of test oracles in a test suite. By doing so, state coverage focuses more on checking the behavior of the programs than merely executing them. The authors conducted a small experiment on the `Apache Jakarta Commons Lang`, a utility package of Java. They randomly reduced the number of checks in a test suite to vary the total number of checks and compute varied levels of state coverage. Their experimental study demonstrated some correlation between total checks and the mutation scores and between checks and state coverage.

Schuler and Zeller proposed *checked coverage* (CC) as a metric to assess test oracle quality [40] [23]. CC is computed by intersecting recorded coverage sets with dynamic backward slices using test assertions as the slice criterion. Their study showed that CC is more sensitive to assertions in test suite than statement coverage. Their study only considered statement coverage criterion and showed that CC is always less than statement coverage and it is common for mature test suites to miss assertions in the test case. In our study, we adapt CC to support an additional coverage domain, OBC, and to compute coverage with respect to that underlying domain which allows their direct comparison – what we term the coverage gap. Additionally, we propose a recommender that recommends actionable suggestion to improve CC and thus improving test suite fault-detection effectiveness.

While CC and state coverage are limited to statement-based coverage, the oracle-based concept of *observability* was proposed for the MC/DC coverage criterion to support the testing of mission critical systems [49]. Observability ensures a masking-free path from the coverage element to a test oracle and in this way it can guide test generation. This idea was later extended to other Boolean expression-based criteria [37]. Existing implementations of observable coverage focus on test generation for data flow languages and are not applicable to computing test suite coverage for imperative languages. Consequently, we do not include them in our evaluation. Like this work, our extension of CC to support other host criteria, like OBC, allows us to explore whether checked coverage varies with the host criteria.

III. Approach

We adapt CC to different host coverage criteria, define the coverage gap, and how the gap can be analyzed to recommend test suite enhancements that reduce it.

A. Host Coverage (h)

We generalize CC to allow it to be applied to an underlying *host* coverage criterion.

Definition 1 (Coverage Elements and Domain): A coverage element represents the execution of a fragment of a program (e.g., statement, branch, condition, decision, def-use pair). A coverage domain, H, is the set of coverage elements associated with a given program.

Definition 2 (Test Coverage): Test coverage for a host criterion for an individual test t is defined by a function, $h(t) \subseteq H$, that records the set of coverage elements executed by the test.

Test coverage naturally extends to test suites, TS, $h(TS) = \bigcup_{t \in TS} h(t)$ and coverage is typically reported as the ratio $|h(TS)|/|H|$. When it is clear from the context we refer to this ratio as h.

B. Checked Host Coverage (hcc)

We adapt the definition of Schuler and Zeller so that CC computes the percentage of coverage elements defined by a chosen host criterion that are executed by a test suite and are observed by at least one test oracle. A coverage element is *observed* if there is a chain of dynamic data or control dependencies associated with a test execution that begins with the element and ends with the argument of a test oracle assertion.

Definition 3 (Checked Coverage): The set of observed coverage elements for domain H and a test t with oracle o is:

$$hcc(t) = \text{MAP}(H, \text{SLICE}(o, \text{TRACE}(t)))$$

where TRACE records an object code trace of a test run, SLICE computes a backwards dynamic slice of its second argument (TRACE(t)) using the first argument (O) as a slicing criterion, and MAP converts a sliced trace to a set of coverage elements in H.

Functions TRACE and SLICE are the same as what was defined for CC [40]. The MAP function is maps to the selected coverage domain H. In CC, line number metadata was used to map object trace elements back to source lines, but to generalize to other domains additional processing is required. For example, analysis of branch instructions and their outcomes is needed to record OBC and for coverage domains whose elements cannot be associated with a single object code instruction, e.g., def-use coverage, MAP must ensure that the entire coverage element is present in the sliced trace, e.g., a bytecode writing to a field and a subsequent bytecode reading from that field.

As with traditional coverage, CC extends naturally to test suites and coverage can be reported as the ratio $|hcc(TS)|/|H|$. When it is clear from the context we refer to this ratio as HCC, with H replaced by the appropriate coverage criterion, e.g., SCC and OBCC for statement and object-branch checked coverage, respectively.

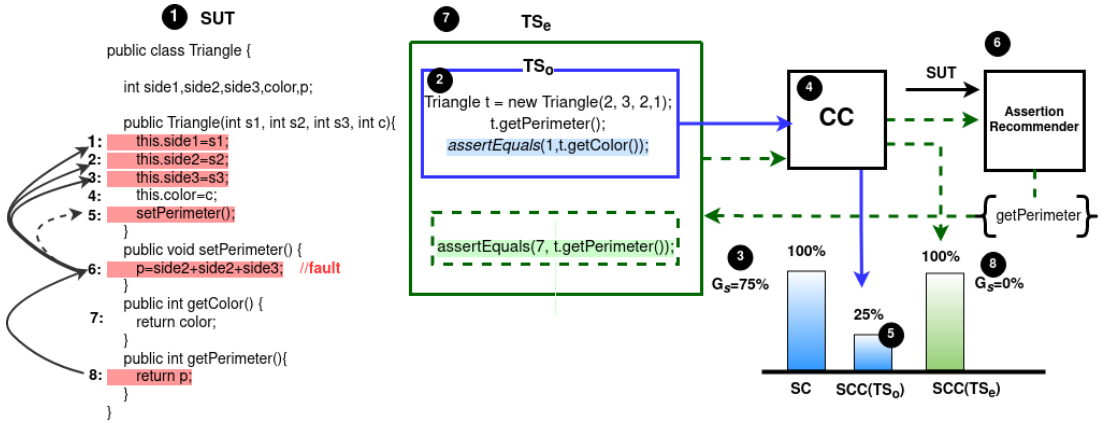

Fig. 1. Overview of the HCC framework on a simple SUT. TS_o is the original test suite (blue) and TS_e is enriched test suite (green); their statement (SC) and statement checked coverage (SCC) are shown to the lower right along with the statement checked coverage gap (G_s).

C. Coverage Gap (G)

Since h and hcc are defined over the same domain, H, we can compute the gap between them. As we discuss below the set of coverage elements that constitute the gap, $G_h(TS) = h(TS) \setminus hcc(TS)$, can be useful in recommending test improvements. The size of the gap is simply the ratio of the size of that set to the size of the domain, $|G_h(TS)|/|H|$. When it is clear from the context we refer to the gap as G_h.

D. Assertion Recommender

As we show in §V the size of the G_h is negatively related to fault-detection effectiveness. One approach to closing the gap, and thereby improve a test suite, is to add assertions, but what assertions should be added?

Our key insight is that the gap exists because the chain of dependencies from an element of the coverage domain to an oracle does not exist - either **C3** or **C4** are not met. Consequently a covered but unobserved element of the coverage domain can be eliminated from the gap by adding a focus method to a test assertion that is the sink of a chain of dependencies beginning with that element. We propose light-weight static analysis to compute a candidate set of non-void focus methods in the System Under Test (SUT) based on G_h – the elements in the gap. More specifically, a non-void function is *recommended* if: (1) it reads a field and the G_h contains a write to that field; (2) it reads a field and the G_h contains a call to a function that writes that field; or (3) it contains an element in G_h that reads/writes a field. A static intra-procedural flow-insensitive analysis can efficiently compute a set of candidate focus methods based on these constraints. This analysis may, however, yield many recommendations. When a recommended method calls another it has the potential to subsume the observations made by the second. We codify this heuristic by computing a breadth-first ordering of the call graph of recommended methods. The top-K recommendations are the first K methods in this ordering. In §V, we show this approach is quite effective. In future work, we plan to explore accuracy improvements to the analysis and the ranking heuristics.

This process requires that the test engineer formulate an oracle assert statement whose argument invokes the recommended focus method. The specifics of arguments to the recommended function and how its results are tested are left to the tester, but future work could build on methods that automatically generate assertions [47]. In §V, we study the efficacy of recommendations by removing test assertions from the test suite and measuring whether the focus methods used in those assertions are recommended.

E. Motivating Example

Figure 1, shows a motivating example that illustrates how faults can hide in the gap between host coverage and its CC variant and how the recommender can help close the gap and detect the faults. (1) shows a simple Triangle class with eight statements. (2) shows it's original test suite, TS_o, that achieves 100% statement coverage, SC. SCC ((4)) is computed using the test assertions, shown with a blue background, as the slicing criterion in TS_o. This calculation reveals that only 25% ((5)) of the statements are covered and observed, which leaves a significant gap, $G_s = 75\%$((3)).

The unobserved statements are shown with red background in (1). If faults are present inside `setPerimeter` method, such as "p=side2+side2+side3", the test suite will not be able to detect them. Therefore, despite achieving 100% statement coverage, the original test suite is not as effective at detecting faults. One well-understood strategy for increasing fault detection effectiveness is to strengthen a test oracle [51]. The checked coverage gap provides guidance on how oracles can be strengthened.

To illustrate, we show data and control dependencies among the unobserved statements in (1). Line 1, 2, 3 write the values of the `side1`, `side2`, and `side3`, line 6 uses these values to define the value of `p` which is returned by line 8. The assertion recommender in (6) analyzes such read/write relationships to suggest focus methods for enriched test assertions that can reduce the coverage gap. In this example, the `getPerimeter` method is recommended and adding a test assertion using it

yields an enhanced test suite TS_e (7) that achieves 100% SCC (8) reducing the gap to 0%.

It can be challenging to navigate dependency chains from the unobserved statements and add the appropriate assertions in real programs with thousands of statements and hundreds of methods. HCC automates the first part of this process. Then, the recommender utilizes the HCC report and efficiently analyzes the SUT to recommend valuable methods for assertions.

IV. Implementation

We use two industrial standard tools to record host coverage: for statement coverage *Atlassian Clover* [7] and for object branch coverage JaCoCo [27]. We use the maven-clover-plugin(4.0.6) and jacoco-maven-plugin(0.8.2).

A. CC computation

We use JavaSlicer, a backward dynamic slicing tool for Java [22], to implement the SLICE component of CC. This tool implements a TRACE module that instruments Java bytecode and inserts additional instructions to record execution logs with source file line numbers, variable references, instruction type, and control flow jumps.

JavaSlicer is capable of supporting large and complex slicing criterion and either individual or multiple assertions to be included in a single slice. We use the Java parser [28] to extract test assertions from JUnit tests and formulate a slicing criterion.

For the statement coverage domain, we implemented the MAP function (discussed in §III-B) using the source file annotations included in the bytecode slices. For the object branch coverage domain, we implemented the MAP function by extending JavaSlicer. JavaSlicer does not provide information regarding which branch was taken for conditional instruction. We extend the JavaSlicer so that when a conditional instruction is found, we check whether the execution jumps out of the current BB or not. This information is used to map slices to the object branch coverage domain.

These implementations of TRACE, SLICE, and MAP for both statement and object branch coverage establish a solid basis for supporting a wider range of criteria; we leave that to future work.

B. Recommender

The recommender takes two inputs: unobserved bytecodes from G_h and the SUT. Since the recommender analyses are flow-insensitive we use the ASM library [18] to efficiently scan the SUT bytecode files to detect read/write relationships on fields that appear in unobserved bytecodes as outlined in §III.

The following listing illustrates an examples of the recommendations computed for statement coverage gaps in the Joda Time software.

Listing 1. Unobserved statement calls a method that writes field

```
final class GJDayOfWeekDateTimeField extends PreciseDurationDateTimeField {

    GJDayOfWeekDateTimeField(BasicChronology chronology, DurationField days) {
46      super(DateTimeFieldType.dayOfWeek(), days); /* writes iUnitMillis field */
        iChronology = chronology;
    }

Recommendation: org/Joda/time/field/PreciseDurationDateTimeField.getUnitMillis
    /* method from super class that reads iUnitMillis */
    public final long getUnitMillis() {
        return iUnitMillis;
    }
```

In Listing 1, line 46 is an unobserved statement, which calls the constructor of the super-class `PreciseDurationDateTimeField`, which writes the `iUnitField` and `iUnitMillis` fields. After the execution of line 46, calling and adding an assertion with the `getUnitMillis` method can bring line 46 into the dependence chain to be observed by a test assertion.

The recommender does not automatically add assertions. It is up to the developer to incorporate the recommended methods into meaningful assertions. New inputs are not required for most cases since unobserved statements are already executed. However, a method can also be called transitively by other methods making it difficult for the oracle to check its result. In such a scenario, a tester may need to generate new inputs to call the recommended method directly.

V. Experimental Study

We answer the following research questions to evaluate our study:

RQ1 What are the coverage differences between host coverage and HCC?

RQ2 To what extent do faults hide in the gap between host coverage and HCC?

RQ3 Are the recommended focus methods representative of the assertions in the original test suite?

RQ4 Does adding recommended assertions improve fault detection effectiveness?

A. Artifacts

We evaluate our research on 13 large-scale open-source Java systems as shown in Table I. To generalize the findings, we have selected subjects from the checked coverage paper [41], Defects4J [19], and some artifacts from GitHub. We select these artifacts because they have mature and well-developed test suites, a few artifacts are currently part of the Java Development Kit (JDK) or used as a Java utility package, and they have a large-scale codebase. In aggregate, our study considers 248 thousand source lines of codes (SLOC) and 237 thousand lines of test code spread across 16 thousand test cases with more than 51 thousand assertions. Our study includes the largest artifacts in the original checked coverage study, but uses more recent versions of those code bases. We replace artifacts from that study, that are very dated and use old and deprecated Java features, with more modern software systems from different sources. The selection of these artifacts allows us to confirm the general findings of Schuler and Zeller [40] while enabling a broad evaluation of our research questions.

B. RQ1: Host Coverage versus HCC.

RQ1 investigates the gap in structural testing in terms of the difference between code coverage and host checked coverage,

TABLE I
DESCRIPTION OF ARTIFACTS

Artifact (version)	Description	Program Size(SLOC)[1]	Test Size(SLOC)[1]	Tests(#)[2]	Assertions(#)[2]
Commons-Cli (1.4) [13]	Command line option parsing	2,699	3,932	372	573
Commons-Codec (1.2) [14]	Common encodings	8,352	12,182	887	1,793
Commons-Csv (1.5) [15]	CSV utilities	1,615	4,467	296	934
Commons-Lang (3.6) [16]	Java helper utilities	27,265	48,172	2,908	15,424
Commons-Validator (1.6) [17]	Data validation	7,409	8,352	536	2,486
Gson (2.8.0) [21]	JSON support	7,815	13,762	1,014	1,780
Jackson-Dataformat-Xml (2.9.10) [26]	XML processing	4,945	5,728	185	556
Jaxen (1.2.0) [29]	XPath engine	10,760	8,042	716	587
JFreeChart (1.5.0) [30]	2D Charts	97,350	39,348	2,174	5,506
Joda-Time (2.10.11) [31]	Date and time library	28,811	55,849	4,238	17,973
Jsoup (1.10.1) [32]	HTML parsing	10,785	5,499	510	1,645
Plexus-Utils (3.1.0) [9]	Utility classes	18,496	6,337	304	799
XStream (1.14.15) [10]	XML serialization	21,741	25,518	1,830	1,554
	Total:	248K	237K	16K	51.6K

[1] Source lines of code (SLOC) are non-comment, non-blank lines reported by IntelliJ statistic plugin.
[2] Tests are JUnit test cases annotated with @Test, Assertions are JUnit assertions

TABLE II
HOST COVERAGE, HCC AND COVERAGE GAP FOR STATEMENT AND OBJECT BRANCH CRITERIA

Artifact	Test(#)	Assertion(#)	SC(%)	SCC(%)	Gap_s (%p)	OBC(%)	OBCC(%)	Gap_{ob} (%p)
Commons-Cli	137	405	83	55	28	74	44	30
Commons-Codec	563	1,030	75	32	43	77	32	45
Commons-Csv	278	898	92	49	43	88	41	47
Commons-Lang	2,534	14,153	82	54	28	81	52	29
Commons-Validator	442	2,276	77	51	26	76	46	30
Gson	1,014	1,723	86	48	38	79	46	33
Jackson-Dataformat-Xml	185	530	68	47	21	60	41	19
Jaxen	581	567	67	38	29	56	25	31
JFreeChart	2,174	5,420	57	21	36	47	17	30
Joda-Time	4,193	17,589	89	55	34	77	41	36
Jsoup	510	1,645	73	36	37	73	35	38
Plexus-Utils	277	780	48	26	22	37	18	19
XStream	1,697	1,238	74	25	49	72	21	51
Total/Average:	14.6K	48.3K	75	41	34	69	35	34

presented as a percentage point (*pp*), for statement and object branch criteria.

Experimental Procedure. To compute checked statement coverage (SCC) and object branch coverage (OBCC), first, we identify all JUnit assertions in the test suite and construct the slicing criteria automatically using our implemented tool. Next, we record the execution trace of the test class using the JavaSlicer tracer module and compute the dynamic slices using the slicer module. We repeat this process for all test classes in each artifact. Occasionally, we need to exclude test cases/classes due to the limitations of JavaSlicer. We also exclude those tests from host coverage computation to avoid overestimating the gap. Our study considers 91% of the total test cases and 94% of the total assertions from the original test suites across the 13 artifacts. The total number of tests and assertions are shown in columns II and III of Table II. Once all slices are computed, we count the total unique statements and object branches across each subject. Statement coverage gap (G_s) and object branch coverage gap (G_{ob}) are then calculated by subtracting SCC and OBCC from the statement and object branch coverage and represented as percentage points.

Analysis and Findings: Table II summarizes the results, where each row represents coverage values for each artifact. Each row includes the total number of tests, assertions, statement coverage (SC), statement checked coverage (SCC), statement coverage gap (G_s), object branch coverage (OBC), object branch checked coverage (OBCC), and object branch gap (Gap_{ob}). In total, we have studied 14.6 thousand tests and 48.3 thousand assertions. For the statement criterion, the minimum and maximum gaps are 21 *pp* and 49 *pp*, and the average statement gap is 34 *pp*. Note that gaps for two different subjects are not comparable, as it depends on each subject's host coverage. Similar to the statement coverage, the object branch coverage gap varies from 19 *pp* to 51 *pp*. We observe a 34 *pp* average object branch coverage gap across 13 subjects.

> **RQ1:** On average, approximately 34% of the code elements exercised by test inputs are not checked by test oracles.

C. RQ2: Coverage Gap versus Fault-detection Effectiveness

In RQ1, we notice a substantial gap between host coverage and HCC. As the gap represents the percentage of coverage elements executed but not observed by any oracle, faults residing within the gap are more likely to go undetected. Therefore, in RQ2, we investigate the extent to which faults may hide in the gap and how the coverage gap may affect the fault-detection effectiveness of a test suite. To this end, we conduct two experiments operating at different granularities.

The first experiment investigates the relationship between the gap and test effectiveness at the application level, and the second experiment investigates the relationship between the gap and test effectiveness at the package level.

1) Application Level Analysis: **Experimental Procedure.** In this experiment, we manipulate the gap size for 12 of the applications in RQ1 by randomly removing assertions from their test suites[1]. To remove an assertion without affecting the host coverage, we transform selected JUnit assertions with a **no-op**. This preserves host coverage, since the assertion arguments are still evaluated, but it may affect HCC and thus impact the gap. For each application, we derive 15 different test suites from the original test suite by enabling the following percentage of assertions from test suites: 0% (all assertions disabled), 1%, 2%, 5%, 7%, 10%, 15%, 20%, 25%, 30%, 45%, 55%, 65%, 75%, and 100%. The selection of assertions is random. We compute mutation scores for all test suites and use them as a proxy for fault detection effectiveness. We mutate the source code using PIT's **STRONG** mutation operators (the rest of the PIT parameters are set to their default values), which injects between 794 and 34,608 mutations across the applications; with a total of 95,393 mutations for the entire study. We compute the kill scores for an application by running the 15 generated test suites on the mutated source.

Analysis and Findings. Figure 2 shows the statement coverage gap and mutation score plots for 12 applications used in RQ1. The gap size is on the X-axis and the mutation score is on the Y-axis. We also compute a linear regression with R^2 value and slope for each artifact. The R^2 value indicates the goodness of the fit of the linear regression model, and it measures the proportion of the variance in the dependent variable (mutation score) predictable from the independent variable (Gap). The slope indicates the steepness of each regression line which measures the change in mutation score as the coverage gap increases 1 unit. The higher the slope, the higher the change rate. We also compute Kendall's correlation coefficient (τ) to measure the strength and direction of the relationship between coverage gap and mutation score.

We see high R^2 values (0.76 to 0.98), indicating that the gap can predict a high proportion of the variance in the mutation scores. High correlation coefficients (τ) (-.88 to -.1) also indicate that gap and test effectiveness are strongly and negatively correlated. Across the study the p-values computed for the correlation coefficients lie in the range $[2.14e^{-12}, 1.33e^{-04}]$, indicating that the strong negative correlation holds with significance.

The negative slope of the regression line measures the drop in mutation score as the coverage gap increases. A high negative slope also indicates the strength/effectiveness of the oracles. Commons-codec has the highest slopes (-1.37), indicating that test effectiveness drops quickly as the gap increases. A lower slope, like Jaxen's -0.13, indicates that the mutation score decreases by .13% as the statement coverage gap increases by 1%. We found that this was caused by the large number of mutants (40%) seeded in Jaxen killed by the runtime system (implicit oracles). When all 581 assertions are enabled, Jaxen achieves a 49% mutation score, meaning that explicit assertions kill only 9% of mutants (around 480 mutants). Further analysis showed that the Jaxen test suite contains many exception oracles that check exceptional behavior inside the `catch` block, and STRONG mutators do not mutate exception cause, types or message. As a result, enabling/disabling those assertions does not affect the mutation scores much.

2) Package Level Analysis: Testing typically operates at a finer granularity than the entire application, e.g., at the class or package level. We explored whether the trends observed at the application also appear at the package level along with extending the study to consider the OBC host criterion and its checked variant OBCC.

Experimental Procedure. To mitigate costs in conducting this study, we did not consider all packages across all applications. Instead, we selected three applications from which we selected three packages each. We selected the two applications from RQ1 with the largest test suites, Commons-lang and Joda-time. Both have relatively high degrees of statement coverage, above 80%, so we chose an application with less than 70% coverage, Jaxen, for variety.

To select the packages, we first sort the packages based on the statement count and then sort again based on statement coverage and select the top three packages per application. We make this choice to maintain variation in the package size and coverage. From Joda-Time, we have selected convert (433 statements, 98% SC), time (4320 statements, 96% SC), and chrono (2445 statements and 83% SC). From Commons-Lang, we have selected mutable (320 statement and 100% SC), math (699 statement and 98% SC), and lang3 (6011 statement and 97% SC) packages. From Jaxen, we selected, expr (891 statement, 88% SC), function (446 statement, 99% SC) and Jaxen (413 statement, 91% SC) packages. These package sizes vary from 320-6011 statement count, and coverage varies from 83 to 100%. We manipulate the coverage gap using the same approach as in the application-level study. For each package, we generate 15 test suites to explore the SC gap and 15 test suites to explore the OBC gap. We mutate the target package source code (using PIT's targetClasses option), run the newly generated test suites on the mutated source code, and record their mutation scores.

Analysis and Findings. Figure 3, shows three rows of subplots for the Joda-Time, Commons-Lang, and Jaxen packages, respectively. Each subplot consists of two datasets; blue data points show statement coverage gap and mutation scores, and orange data points show object branch coverage gap and mutation score.

A package's coverage gap depends on its host coverage and CC score, and we see a wider range of gaps across packages than was observed at the application level. The highest gap (when all assertions are disabled) is the same as host coverage. On the other hand, the lowest gap is achieved when all asser-

[1]XStream was not included in RQ2 because it triggers a bug in the PIT mutation testing tool used in this study.

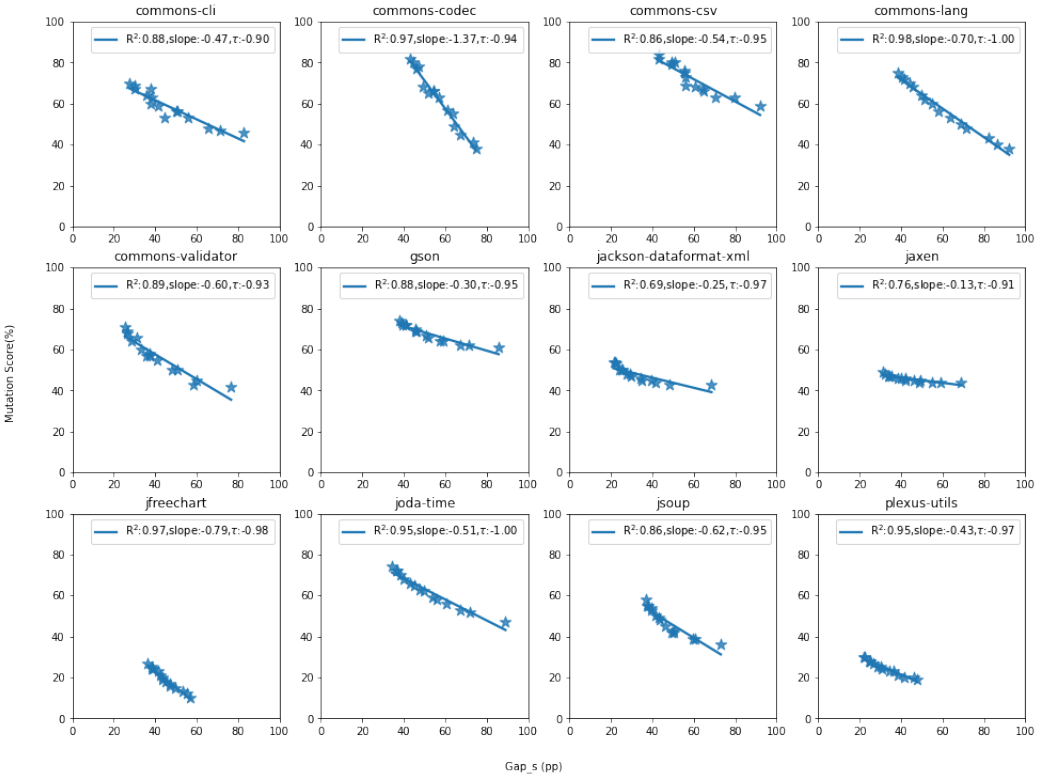

Fig. 2. Relationship between statement coverage gap (G_s) (X-axis) and mutation kill score (Y-axis) for 12 applications (one panel per application). Data points plot gap between SC and SCC versus mutation kill score for 15 test suites with varying oracle strength. Slope of regression line and statistical tests for each application shown in the box in the top of each panel.

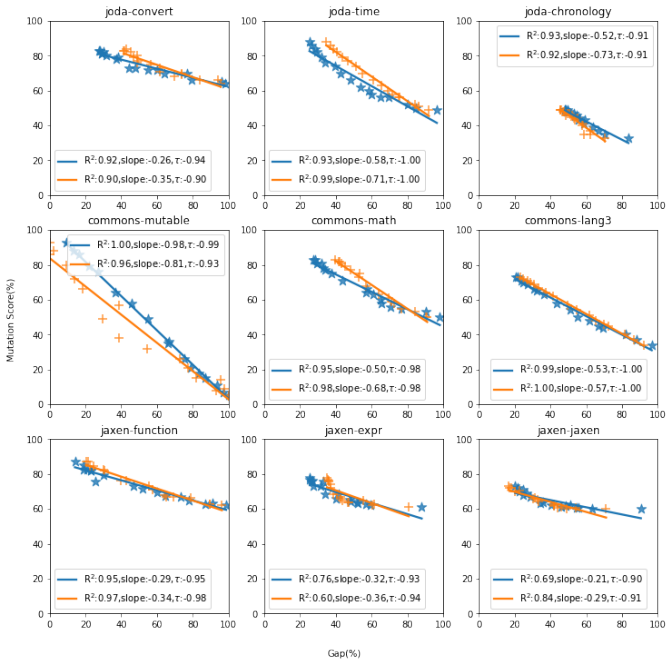

Fig. 3. Mutation score vs statement coverage gap (blue data points) and object branch gap (orange data points) at the package level. Each dataset is computed from 15 test suites (30 test suites per package) with varying gap and their mutation scores.

tions are enabled, which is the difference between host coverage and HCC. For example, the commons-mutable package's statement coverage gap varies from 9-100 *pp* (SC=100% and SCC=91%), whereas the joda-chronology package's statement coverage gap varies from 48-83*pp* (SC=83% and SCC=35%).

Similarly, mutation score also varies across a broader range at the package level than was observed at the application level. The lowest mutation score indicates the kill score when gap is maximum (all assertions as removed). For example, the commons-mutable package has the lowest mutation score of 5% (G_s=100*pp*) even when SC is 100%, which indicates that 95% faults can hide in the gap when host coverage yields a high score. Commons-math has a low mutation score of 50% (G_s=98*pp*) when statement coverage is 98%, indicating that 50% of faults can go undetected even with a perfect host coverage. As the data suggests, this low mutation score varies from package to package depending on where and what type of mutants are injected. However, the variation of implicitly killed mutants does not affect the relationship between the gap and kill score; instead, it just shifts the regression lines along the Y-axis.

Looking across host criteria, we see similar trends for statement and object branch coverage. These trends are similar to those observed at the application level. We observe high correlation coefficient (τ) (ranging from -.9 to -1) values for all packages and criteria, indicating that test suite effectiveness is strongly negatively correlated to the gap. Across the nine packages and two host criteria, the p-values computed for the correlation coefficients lie in the range $[1.02e^{-21}, 2.51e^{-04}]$, so we can say that the strong negative correlation holds with significance

We also get high R^2 values for all packages and criteria, indicating that the gap variation can explain a high proportion of variance in test effectiveness.

RQ2: Faults can hide in the coverage gap and there is a strong negative and statistically-significant correlation between coverage gap size and fault-detection effectiveness.

VI. RQ3: Recommendation Evaluation

In RQ1, we notice gaps between host coverage and HCC of up to 49%, indicating that almost half of the executed statements do not influence any oracles in the test suite. RQ2 shows that test effectiveness decreases as the coverage gap increases, and a strong negative correlation exists between them. Therefore, a tester should aim to lower this coverage gap by enriching the oracle. To this end, we implement the recommender discussed in Section IV-B to provide developers actionable feedback to reduce the gap by adding more test oracles. In this research question, we evaluate the accuracy of recommendations.

Assessing Recommendation Quality. A *focus method* is one whose result is checked by an assertion. To evaluate the quality of the recommendations, from each test suite in each of the artifacts, we remove a single assertion at a time, record the set of statements that become unchecked, and then run the recommender to recommend focus methods. If the focus method in the removed assertion is among the top-k recommendations, then we have a top-k match. We analyze the 34,894 assertions, from the 14.6k test cases of the 13 artifacts, that include a focus method in the applications' code. A possible future extension may include assertions having no method calls in the assertion body, as we can easily find the focus methods by analyzing the bytecodes.

Table III summarizes the results. Each row represents an artifact. Column I shows the artifact name, and II represents the number of assertions for evaluating the recommendations. Columns III, IV, and IV present the top-1, top-5, and top-10 scores. The top-K score measures the percentage of focus methods within the top-K recommendations.

On average, 46%, 67%, and 73% of the assertion focus methods are within the top 1, top 5, and top 10 recommendations. Furthermore, for artifacts like Common-Codec, Commons-Lang, and JFreeChart, the recommender achieves more than 80% top 1 score. The recommender recommends all possible ways to check a set of unchecked statements; however, the ranking algorithm works well when there is a single dominator method in the call graph of the recommended methods.

For other artifacts like Jaxen, Commons-Cli's, and Joda-Time, the scores are lower for various reasons. For Commons-Cli's the reason is that the test cases are less diverse. Among 332 assertion focus methods, only 46 focus methods are unique. As a result, when the recommender misses a few focus methods in the top-1, the overall top-1 score goes down quickly.

TABLE III
Percentage of assertion focus methods recommended within the top-k recommendations across all 13 artifacts

Artifacts	Assert(#)	Top 1(%)	Top 5(%)	Top 10(%)
Commons-Cli	332	16	51	70
Commons-Codec	532	84	96	97
Commons-Csv	602	69	84	90
Commons-Lang	9843	80	96	98
Commons-Validator	1441	50	77	89
Jackson-Dataformat-Xml	83	33	43	63
Jaxen	134	11	30	37
JFreeChart	3240	82	93	97
Joda-Time	15775	17	43	53
Jsoup	1098	21	31	38
Gson	871	53	82	87
Plexus-Utils	365	55	75	78
XStream	578	38	56	59
Summary	**Total 34894**	**Average 46**	**Average 67**	**Average 73**

For Jaxen, we find that from its 134 assertions, 83 checks for exception causes, types, and messages. This is problematic because JavaSlicer cannot construct the slice properly when an exception is thrown. As a result, those statements are not in the unchecked statements due to assertion removal, and the recommender misses valuable information, which affects the ranking process.

For Joda-Time, we find that it has a complex hierarchy of sub and super-class, specially, in the Chronology package, where fields are read and written by multiple methods. Since any of these methods could be a focus method, the recommender chances of producing the one used in the code are reduced.

RQ3: On average, 67% of the focus methods in the original test suites are suggested within the top 5 recommendations. Restricting to the top 1 recommendation, nearly half of the developer-written focus methods are present.

VII. RQ4: Assertion Addition and Gap Reduction.

RQ3 shows that the recommender can accurately suggest focus methods, and nearly 50% of the focus methods are suggested within the top 1 recommendation. In RQ4, we perform a smaller scale study on the classes in the `org.joda.time.chrono` package's test suite to investigate whether adding assertions reduces its gap and improves fault-detection effectiveness. To this end, we manually added assertions using simple nullness or equality checks of the recommended methods' return values. For example, `GJEraDateTimeField` class has 13 *pp* gap and our recommender suggested to check the return value of `getMaximumTextLength` and `set` methods. We added the assertions shown in Listing 2, resulting in a 10 *pp* gap reduction and 7 *pp* fault-detection improvement.

Listing 2. Additional assertions based on the recommendations

```
GJEraDateTimeField gf = new GJEraDateTimeField(IslamicChronology.getInstance());
assertNotEquals(0, gf.set(222222,1));
assertNotEquals(0,gf.getMaximumTextLength(Locale.UK));
```

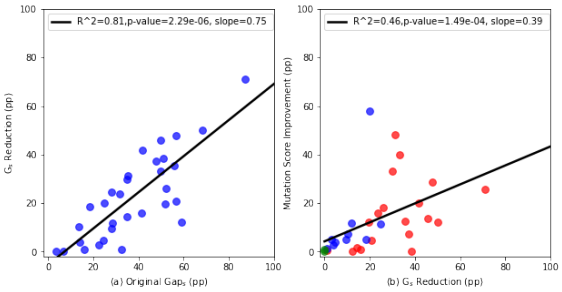

Fig. 4. Gap reduction and mutation score improvement due to additional assertions in the Joda-Time chronology package

We compute the coverage gap for the original and enriched versions of the chronology package's test suite, and report gap reduction as percentage point differences. Similarly, we perform mutation testing on the original and enriched test suite and report the change in score as percentage point differences.

Figure 4(a) shows a data point per class with 31 in total. Each data point represents the original gap (x-axis) and gap reduction (y-axis) due to additional assertions for a class. Generally, when the original gap is large, there is more room for improvement; thus, the gap reduction also tends to be high. For example, the `BasicFixedMonthChronology` class has an 87 *pp* original gap, and additional assertions reduced the gap by 70 *pp*. Conversely, when the gap is small, the chances of reducing the gap diminishes. For example, the `AssembledChronology` class has only 6 *pp* gap, and its gap reduction is zero. We see a strong positive correlation (.81) between the original gap and gap reduction due to additional assertions, demonstrating that recommendations are valuable in improving checked coverage.

Figure 4(b) shows the gap reduction on the x-axis and mutation score improvement on the y-axis. We use color to further delineate relationships: green points have original gaps below 10 *pp*, blue points have a gap between 10-30 *pp*, and red points have gaps of more than 30 *pp*. We observe variability in this relation based on the original gap. When the gap is large, red points, there is a potential for more significant mutation score improvement, but this reduces with gap – see the blue points and the green points. We note that additional confounding factors, such as class sizes, number of mutants per class, and strength of inserted assertion are likely impacting the data. Despite these confounds, the data do show a correlation between increased mutation scores and coverage gap reduction. This suggests that gap reduction is necessary for improving fault-detection effectiveness.

These findings suggest the potential benefit of the recommender to increase fault detection effectiveness, even when using the simplest assertions. In future work we plan to integrate the recommender with automated assertion generation tools (e.g., EvoSuite [20]) to generate high-quality automated assertions that check the recommended focus method.

RQ4: Additional assertions improved fault-detection effectiveness by as much as 58 *pp*, with an average improvement of 13 *pp*

A. Threats to Validity

External Validity. We study 13 open-source applications from different domains and organizations. However, the findings may not generalize to other projects following different coding and V&V practices, or having fault types that require other infrastructure to be detected. Similarly, our evaluation targets two common host criteria but further studies are needed to confirm that the findings generalize to other criteria.

Internal Validity. We employ several tools in the studies. These tools and our manipulation and integration of their outputs may have faults. We mitigated this threat by using publicly available and broadly used tools such as JavaSlicer, PIT, Clover, JaCoCo, and the ASM library to limit this threat. We also conducted extensive testing of our pipeline which led us to identify other limitations. For example, JavaSlicer cannot handle well some integral Java classes (Java.lang.String, java.lang.System, java.lang.Object). As a result, dependencies through method calls of these classes cannot be computed, leading to the overestimation of the coverage gap. To handle such threats, we documented and excluded classes and tests that the tools could not process consistently. To manipulate coverage gap, we remove assertions from test suite. Although systematic, this approach does not precisely control that removing a percentage of random assertions may lead to different gap sizes. However, as we only care about how the gap affects the effectiveness, we do not consider this as a threat to validity.

Construct Validity. We have established a connection between the coverage gap and test effectiveness through the use of mutation testing. This connection depends on the extent to which the generated mutations corresponds to real faults. Future work could address this threat by examining whether faults in previous versions of the systems reside in the coverage gaps. Also, we note that the depicted relationship between gap size and mutation scores is confounded by the presence of mutations that are killed by implicit oracles (those detected by the system, not the test assertions). Our construct does not differentiate among faults that require an explicit versus an implicit oracle, but the findings make it clear that the more assertions that are required to expose a fault, the stronger the relationship between gap and test effectiveness.

VIII. Conclusion and Future Directions

Structural coverage is widely used as a test adequacy metric, but its inability to relate coverage to what is observed by a test oracle limits its fault-detection effectiveness. Our work continues a line of research, beginning with Schuler and Zeller [40], that clearly demonstrates that incorporating the oracle into coverage metrics improves their ability to measure test suite quality. In particular, we find that there is a substantial gap between the portions of a program that are covered and those

that may influence a test oracle outcome. This gap can hide faults, but it also represents a source of information that can be leveraged to improve the fault detection effectiveness of test suites by enhancing their test oracles with assertions that call well-chosen focus methods.

We did not consider the cost of checked coverage in this work, but we note that in our experiments it yields an order of magnitude increase in the overhead of test coverage. Depending on the development context, this may mean that checked coverage should only be run periodically, e.g., when the source code of tests themselves are changed, rather than in a typical continuous integration workflow. Such an adaptive approach to configuring different forms of test coverage is consistent with other research on how to adjust coverage methods to best suit developer needs [25].

IX. Data Availability

Artifacts, tools, and results are available for review at https://github.com/icse2023anon/cc-gap-artifacts and will be made publicly available upon acceptance.

X. Acknowledgements

This material is based in part upon work supported by the DARPA ARCOS program under contract FA8750-20-C-0507, by The Air Force Office of Scientific Research under award number FA9550-21-0164, and by Lockheed Martin Advanced Technology Laboratories.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, jun 1990.

[2] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411, 2005.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[4] S. Berner, R. Weber, and R. K. Keller. Enhancing software testing by judicious use of code coverage information. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 612–620, USA, 2007. IEEE Computer Society.

[5] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 237–249, 2020.

[6] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[7] Clover. Clover – code coverage and testing tool for java and groovy, 2015. https://github.com/openclover/clover-maven-plugin/, Last accessed on 2022-03-17.

[8] Codecov. Codecov - the leading code coverage solution, 2022. http://about.codecov.io/, Last accessed on 2022-03-08.

[9] Codehaus. Plexus-utils – a package of utility classes, 2017. https://github.com/codehaus-plexus/plexus-utils/, Last accessed on 2022-08-28.

[10] Codehaus. Xstream – a package of xml serialization classes, 2020. https://github.com/x-stream/xstream, Last accessed on 2022-08-28.

[11] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 449–452, 2016.

[12] C. Comar, J. Guitton, O. Hainque, and T. Quinot. Formalization and comparison of mcdc and object branch coverage criteria. In *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[13] Commons-Cli. Commons-cli – command line processing classes, 2015. https://github.com/apache/commons-cli/, Last accessed on 2022-08-28.

[14] Commons-Codec. Commons-codec – encoder/decoder classes, 2019. https://github.com/apache/commons-codec/, Last accessed on 2022-08-28.

[15] Commons-Csv. Commons-csv – csv file format classes, 2018. https://github.com/apache/commons-csv/, Last accessed on 2022-08-28.

[16] Commons-Lang. Commons-lang – a package of java utility classes, 2017. https://github.com/apache/commons-lang/, Last accessed on 2022-08-28.

[17] Commons-Validator. Commons-validator – a package of data validation classes, 2017. https://github.com/apache/commons-validator/, Last accessed on 2022-08-28.

[18] J. R. Davis et al. *ASM specialty handbook: tool materials*. ASM international, 1995.

[19] Defects4J. A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research, 2022. https://github.com/rjust/defects4j, Last accessed on 2022-07-07.

[20] 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*, pages 416–419, 2011.

[21] Google. Gson – a package of json manipulation classes, 2016. https://github.com/google/gson/, Last accessed on 2022-08-28.

[22] C. Hammacher. Design and implementation of an efficient dynamic slicer for java. *Bachelor's Thesis*, 2008.

[23] S. B. Hossain and M. B. Dwyer. A brief survey on oracle-based test adequacy metrics. *arXiv preprint arXiv:2212.06118*, 2022.

[24] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[25] M. Ivanković, G. Petrović, R. Just, and G. Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[26] Jackson-Dataformat-Xml. Jackson-dataformat-xml – xml processing, 2019. https://github.com/FasterXML/jackson-dataformat-xml, Last accessed on 2022-08-28.

[27] JaCoCo. Jacoco – java code coverage library, 2014. https://github.com/jacoco/jacoco/, Last accessed on 2022-03-17.

[28] JavaParser. Javaparser – parser and abstract syntax tree for java, 2022. https://github.com/javaparser/javaparser/, Last accessed on 2022-03-17.

[29] Jaxen. Jaxen – xpath engine for java, 2019. https://github.com/jaxen-xpath/jaxen/, Last accessed on 2022-03-17.

[30] JFreeChart. Jfreechart – a package of chart classes, 2017. https://github.com/jfree/jfreechart/, Last accessed on 2022-08-28.

[31] Joda-Time. Joda-time – date and time library for java, 2014. https://www.joda.org/joda-time/, Last accessed on 2022-03-17.

[32] Jonathan Hedley. Jsoup – a package of html classes, 2016. https://github.com/jhy/jsoup/, Last accessed on 2022-08-28.

[33] 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 Symposium on the Foundations of Software Engineering (FSE)*, pages 654–665, Hong Kong, November 18–20 2014.

[34] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434, 2004.

[35] K. Koster and D. C. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *Proceedings of the 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.

[36] LCOV. Lcov – ltp gcov extension, 2022. https://github.com/linux-test-project/lcov, Last accessed on 2022-03-13.

[37] Y. Meng, G. Gay, and M. Whalen. Ensuring the observability of structural test obligations. *IEEE Transactions on Software Engineering*, 46(7):748–772, 2018.

[38] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on software engineering*, 14(6):868–874, 1988.

[39] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Proceedings of the ACM SIGSOFT'89 third symposium on Software testing, analysis, and verification*, pages 86–96, 1989.

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

[41] D. Schuler and A. Zeller. Checked coverage: an indicator for oracle quality. *Software testing, verification and reliability*, 23(7):531–551, 2013.

[42] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 311–320. IEEE, 2012.

[43] M. Staats, M. W. Whalen, and M. P. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *2011 33rd international conference on software engineering (ICSE)*, pages 391–400. IEEE, 2011.

[44] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, 1996.

[45] F. Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[46] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on software Engineering*, 18(8):717, 1992.

[47] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409, 2020.

[48] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[49] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 102–111. IEEE, 2013.

[50] T. Yu, X. Qu, M. Acharya, and G. Rothermel. Oracle-based regression test selection. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 292–301. IEEE, 2013.

[51] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 214–224, New York, NY, USA, 2015. Association for Computing Machinery.