



Measuring and Mitigating Gaps in Structural Testing

Soneya Binta Hossain, Matthew Dwyer, Sebastian Elbaum, Anh Nguyen-Tuong



Apache Commons CSV

Used by 29.4k

+ 29,404

Contributors 38

+ 27 contributors

Languages

Java 99.8% Shell 0.2%

README.md

Apache Commons CSV

Java CI passing codecov 98% maven central 1.10.0 javadoc 1.10.0 CodeQL passing openssf scorecard 8.6

The Apache Commons CSV library provides a simple interface for reading and writing CSV files of various types.

Documentation

More information can be found on the [Apache Commons CSV homepage](#). The [Javadoc](#) can be browsed. Questions related to the usage of Apache Commons CSV should be posted to the [\[user mailing list\]\[ml\]](#).

Where can I get the latest release?

FasterXML/jackson-dataformat-xml

The screenshot shows the GitHub README.md page for the project. At the top, there's a "codecov" badge indicating 61% coverage. A large orange arrow points from this badge to a detailed view of the contributors section on the right.

Contributors 30

+ 19 contributors

Type	Status
Build (CI)	Build and Deploy Snapshot passing
Artifact	maven central 2.15.0-rc2
OSS Sponsorship	lifted!
Javadocs	javadoc 2.15.0-rc2
Code coverage (2.15)	codecov 61%
Fuzzing	oss-fuzz fuzzing

Based on statement coverage CSV seems to be better tested than jackson-dataformat.

Would we say the same thing if we considered the quality of the test oracles in addition to coverage?

Code Coverage, Test Oracle and Fault-detection

❖ Code coverage is *essential* but *insufficient*

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

Parveen Singh Kochhar, Ferdinand Thung, and David Lo
Singapore Management University, Singapore
(kochharp.2012, ferdust.2013, dlo@smu.edu.sg)

Abstract—During activity to ensure that the code is killed by the adequate software to kill mutants. Code coverage is often used to measure quality. It often measures the ability to kill mutants. However, previous studies use small programs and apply it to real systems. In this paper, we measure the relations killing real bugs from test cases. We compare levels of coverage and mutants killed. In our study, we have performed experiments on Java. Our experiment finds strengths and weaknesses of strength of test suites. For HTTP server, we find that strong coverage is strong for both static and dynamic bugs.

Keywords—Code coverage, test suite effectiveness, real bugs, Java.

Testing is an integral part of software development. Code coverage is often used to measure the amount of test coverage of a test suite to understand the quality of test suite. Although this is not general answer to the problem above, we propose a procedure to investigate whether any test coverage criterion has a general impact on the quality of test suite. This procedure can be applied to any test coverage criterion which relates to the impact of just running additional test cases. This procedure is applicable in typical testing conditions where the test suite is run on a real system and the test cases are collected where coverage measures are collected as well as defect data. We then test the procedure on published data and conclude that the results do not support the hypothesis that outcomes do not support the assumption of a causal dependency between test coverage and defect coverage, a result for which several plausible explanations are presented.

Keywords—software test, test coverage, defect coverage, test intensity, simulation

1. Introduction

Testing is one of the most effort-intensive activities during software development [2]. A great deal of research is directed towards development of new, improved test coverage measures for control

methods. One way to better control test resource allocation is referred to as test coverage) of the detected during testing (referred to as test coverage). We have performed experiments on Java. Our experiment finds strengths and weaknesses of strength of test suites. For HTTP server, we find that strong coverage is strong for both static and dynamic bugs.

Keywords—software test, test coverage, defect coverage, test intensity, simulation

1. Introduction

Testing is one of the most effort-intensive activities during software development [2]. A great deal of research is directed towards development of new, improved test coverage measures for control

methods. One way to better control test resource allocation is referred to as test coverage) of the detected during testing (referred to as test coverage). We have performed experiments on Java. Our experiment finds strengths and weaknesses of strength of test suites. For HTTP server, we find that strong coverage is strong for both static and dynamic bugs.

Keywords—software test, test coverage, defect coverage, test intensity, simulation

Copyright © 2012 by the author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2756-5/12/06 \$15.00 © 2012 ACM

http://doi.org/10.1145/2348813.2348771

The Effect of Code Coverage on Fault Detection under Different Testing Profiles

Xia Cai
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
xcai@cse.cuhk.edu.hk

Michael R. Lyu
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
lyu@cse.cuhk.edu.hk

ABSTRACT

Software testing is a key procedure to ensure high quality and reliability of software programs. The key issue in software testing is the selection and evaluation of efficient test cases. Code coverage is often used as a metric to evaluate test effectiveness, but it remains a controversial which lacks of support from empirical data. In this paper, we empirically study the effect of test coverage on testing effectiveness under different testing profiles. To improve the test resource allocation, we propose to evaluate the performance of code coverage, we employ test and mutation testing to compare experimental results to judge the performance improvement code coverage and fault

1. INTRODUCTION

As the main fault removal technique, software testing is one of the most effort-intensive activities during software development [1]. The key issue in software testing is test case selection and evaluation of efficient test cases. Code coverage is often used as a metric to evaluate test effectiveness, but it remains a controversial which lacks of support from empirical data. In this paper, we empirically study the effect of test coverage on testing effectiveness under different testing profiles. To improve the test resource allocation, we propose to evaluate the performance of code coverage, we employ test and mutation testing to compare experimental results to judge the performance improvement code coverage and fault

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
(linozemtseva,rholmes)@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the relationship between test suite size and fault detection effectiveness have failed to reach a consensus about the number of test cases required to achieve maximum fault detection effectiveness. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether the same conclusions hold for large programs. This paper studies the effect of the confounding influence of test suite size on the fault detection effectiveness of test suites with adequate suites, which are rare in practice, so the results may not generalize to typical test suites.

We conducted a study to investigate the relationship between test suite size, coverage, and effectiveness for large programs. We analyzed 13 test suites from the literature: 11 [1] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29]. The basic assumption is that the fault detection effectiveness of a test suite often depends on its coverage as much as the number of test cases. However, since both test coverage and test suite size are correlated with fault detection effectiveness, it is not clear how to separate the causal effect between test coverage and test suite size.

However, since both test coverage and test suite size are correlated with fault detection effectiveness, it is not clear how to separate the causal effect between test coverage and test suite size.

We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases is small. For example, for the test suite with the strongest forms of coverage do not provide greater insight into the effectiveness of the test suite. The reason is that the coverage, while useful for identifying under-tested parts of a program, should also be used as a quality target because it is not always possible to identify all faults in a program.

We conclude that the test suite effectiveness is not strongly correlated with its statement coverage, decision coverage, and/or modified condition coverage when the number of test cases is small.

This paper presents a new study of the relationship between test suite size and fault detection effectiveness, which is the following research questions for large Java programs:

RESEARCH QUESTION 1. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage, and/or modified condition coverage when the number of test cases is small?*

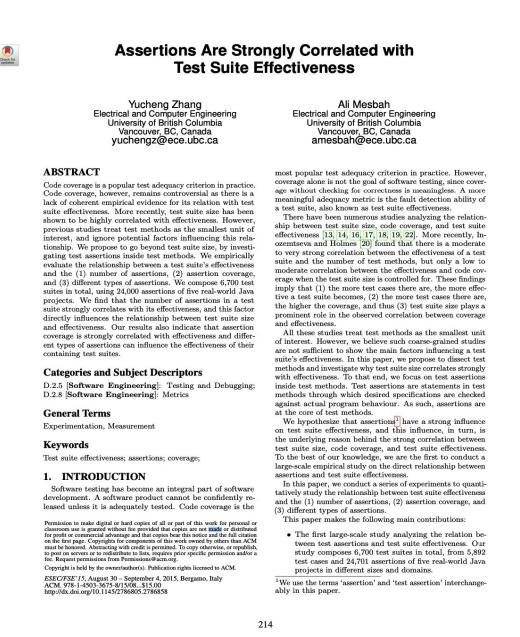
RESEARCH QUESTION 3. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage, and/or modified condition coverage when the number of test cases is large?*

The paper makes the following contributions:

• A comprehensive survey of previous studies that investigated the relationship between coverage and effectiveness (Section 2 and accompanying online material).

Code Coverage, Test Oracle and Fault-detection

- ❖ Code coverage is *essential* but *insufficient*
- ❖ Test oracles and fault-detection are *strongly correlated*



Assertions Are Strongly Correlated with Test Suite Effectiveness

Yucheng Zhang
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
yuchengz@ece.ubc.ca

Ali Mesbah
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT
Code coverage is a popular test adequacy criterion in practice. Code coverage, however, remains controversial as there is a lack of coherent empirical evidence for its relation with test suite effectiveness. More recently, test suite size has been shown to have a strong correlation with test suite effectiveness, and ignore potential factors influencing this relationship. In this paper, we go beyond suit size and investigate test assertions inside test methods. We empirically evaluate the relationship between test suite size, code coverage, and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. We compose 5,703 test suites in three different domains and conduct experiments on five projects. We find that the number of assertions in a test suite strongly correlates with its effectiveness, and this factor directly impacts the overall effectiveness of a test suite and effectiveness. Our results also indicate that assertion coverage is strongly correlated with effectiveness and different types of assertions can influence the effectiveness of their containing test suites.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging;
D.2.8 [Software Engineering]: Metrics

General Terms
Experimentation, Measurement

Keywords
Test suite effectiveness; assertions; coverage

1. INTRODUCTION
Software testing is an integral part of software development. A software product cannot be confidently released unless it is adequately tested. Code coverage is the permission to make digital or hard copies of all or part of this work for personal classes we as is granted without prior written consent are not made or distributed outside your organization, without prior written permission. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior written permission and/or specific permission from the copyright holder. Copyright © 2015, Association for Computing Machinery (ACM).
ESEC/FSE '15, August 30–September 4, 2015, Bergamo, Italy.
ACM 978-1-4503-3675-8/15/08...\$15.00
<http://doi.acm.org/10.1145/2796898>

more popular test adequacy criterion in practice. However, coverage alone is not the goal of software testing, since coverage without checking for correctness is meaningless. A more meaningful adequacy metric is the fault detection ability of a test suite, which is called test suite effectiveness. There have been numerous studies analyzing the relationship between test suite size, code coverage, and test suite effectiveness. For example, Gherghina et al. [22] and Holzmann, heimannova and Holmes [20] found that there is a moderate to very strong correlation between the effectiveness of a test suite and its size. In contrast, Bhandarkar et al. [10] found no moderate correlation between the effectiveness and code coverage. These studies, however, have some limitations. They argue that (1) the more test cases there are, the more effective a test suite becomes, (2) the more test cases there are, the more effective a test suite becomes, and (3) the more assertions there are, the more effective a test suite becomes. All these studies assume that assertions play a prominent role in the observed correlation between coverage and effectiveness.

We hypothesize that assertions have a strong influence on test suite effectiveness and this influence is the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness. To this end, we focus on test assertions and their impact on test suite effectiveness. We study test methods through which desired specifications are checked against observed behaviour. As such, assertions are at the core of test methods.

We hypothesize that assertions have a strong influence on test suite effectiveness and this influence is the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness. To this end, we focus on test assertions and their impact on test suite effectiveness. We study test methods through which desired specifications are checked against observed behaviour. As such, assertions are at the core of test methods.

In this paper, we conduct a series of experiments to quantitatively study the relationship between test suite effectiveness and (1) the number of assertions, (2) assertion coverage, and (3) different types of assertions.

This paper makes the following main contributions:

- The first large-scale study analyzing the relation between two test assertions and test suite effectiveness. Our study contains 6,703 test suites in total, from 399 test cases and 24,701 assertions of five real-world Java projects in different sizes and domains.

¹We use the terms ‘assertion’ and ‘test assertion’ interchangeably in this paper.

Coverage Based on Test Oracles

- ❖ Considers program execution and test oracles
 - Support statement criterion
 - Only *evaluate* test suites

- ❖ We build on Checked Coverage by Schuler and Zeller
 - We support stronger criterion
 - We introduce and study the concept of *Coverage Gap*

State Coverage: Software Validation Metrics beyond Code Coverage

Dries Vanoverberghen^{1,*}, Jonathan de Halleux², Nikolai Tillmann², and Frank Piessens¹

¹ Katholieke Universiteit Leuven, Leuven, Belgium
(dries.vanoverberghen, frank.piessens)@cs.kuleuven.be

² Saarland University – Computer Science
Saarbrücken, Germany
ds@cs.uni-saarland.de

2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

State Coverage: A Structural Test Adequacy Criterion for Behavior Checking

Koen Koster
kenn@agitar.com

David Kee
dk@agitar.com

Agitar Software Laboratories
450 National Avenue
Mountain View, California 94043

ABSTRACT

We propose a new language-independent, structural test adequacy criterion called state coverage. State coverage measures which unit-level tests check the objects and side effects of a program.

State coverage differs in its focus from existing test adequacy metrics such as code coverage and mutation analysis. Unlike mutation-based criteria, state coverage measures the extent of check of program behavior. And unlike testing metrics such as statement coverage, state coverage has been designed to be readily understood and to present users with easily understood test inadequacy reports.

An experiment showed strong positive correlations between the number of behavior checks and both state coverage and mutation adequacy.

Catagories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; D.2.8 [Software Engineering]: Metrics

General Terms: Measurement, Reliability

Keywords: Coverage, fault-based, mutation testing, state coverage, structural testing, test adequacy criteria, unit testing

1. INTRODUCTION

How thoroughly is a program? This is the main question that test adequacy criteria try to answer. Early research assumed that the best way to answer it was to measure *program size* and how many of its statements are evaluated by a test. For instance, branch coverage measures a program's ability to evaluate all possible outcomes when each of those branches were not taken during test execution. *State coverage*, the test adequacy criterion we propose in this paper, evaluates programs by measuring their behavior. It does so by checking whether all objects and side effects and output variables and judges tests according to what they do in the program.

The `testIsFaulty` test below, `testIsNotFaulty` does not check program behavior automatically. Humans have to review the generated test cases and manually verify the correctness of the behavior of the program. The implementation of `abc` could be replaced by any compiling compiler (as is the case with `abc`).

2. MOST EXISTING TEST ADQUA

Most of the existing test adequacy criteria are important but not always appropriate for our needs. We found that few development projects even use the most basic of these metrics. For example, in the Java project we analyzed, no test cases checked for non-existent [8, 12, 14]. Although major factor preventing adoption is the lack of tool support, this is not the only reason why these metrics are not taken during test execution.

State coverage, the test adequacy criterion we propose in this paper, evaluates programs by measuring their behavior. It does so by checking whether all objects and side effects and output variables and judges tests according to what they do in the program.

The `testIsFaulty` test below, `testIsNotFaulty` does not check program behavior automatically. Humans have to review the generated test cases and manually verify the correctness of the behavior of the program. The implementation of `abc` could be replaced by any compiling compiler (as is the case with `abc`).

3. CHECKED COVERAGE: AN INDICATOR FOR ORACLE QUALITY

David Schuler^{*} and Andreas Zeller[†]

[†] Department of Computer Science, Saarland University, Saarbrücken, Germany
ds@cs.uni-saarland.de

[‡] Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

Assessing Oracle Quality with Checked Coverage

Checked coverage: an indicator for oracle quality

SUMMARY

A known problem of traditional coverage metrics is that they do not assess *oracle quality*—that is, whether the computation result is actually checked against expectations. In this paper, we introduce the concept of *checked coverage*—the dynamic slice of covered statements that actually influence an oracle. Our experiments on seven open-source projects show that checked coverage is a sure indicator for oracle quality and even more sensitive than mutation testing. Copyright © 2013 John Wiley & Sons, Ltd.

Received 11 September 2011; Revised 22 November 2012; Accepted 13 March 2013

KEY WORDS: test suite quality, coverage metrics, dynamic slicing, mutation testing

4. INTRODUCTION

How can I ensure that my programme is well-tested? The most widespread metric to assess test quality is *coverage*. Test coverage measures the percentage of code features such as statements or code blocks that are executed during a test. The higher the coverage, the better the chances of catching a code feature that causes a failure—a rationale that is easy to explain, but which relies on an important assumption. This assumption is that it actually are able to detect the failure. If it does not suffice to cover the error, we also need a means to detect it.

As it comes to detecting arbitrary errors, it does not make a difference whether an error is detected by the test (e.g., a mismatch between the expected and actual result), by the programme itself (e.g., a segmentation fault or a maximum system value), or by the user. To make computation results, however, we need tools in the test and in the programme code—checks furthermore summarized as *oracles*. A high coverage does not tell anything about oracle quality. It is perfectly possible to achieve a 100% coverage and still not have any result checked by even a single oracle. The fact that the run-time system did not receive errors indicates robustness, but does not tell anything about future properties.

As a consequence of this, the notion of *checked coverage* is introduced. It is based on the `PatternParser` class from the JAXEN XPATH library, shown in Figure 1. This test case passes a set of paths through the parser. Interestingly enough, it never checks any of the parser results. The parser return complex nonsense, and this test would never notice it.

Why are the parser results uncheckable? Actually, the purpose of this test is not to test the parser, but the individual paths. For example, if any of the paths in a system under test would return an exception, if the test passes, the test is fine. Running this test, however, also achieves a statement coverage of 83% in the parser, and one may thus assume the parser is well-tested.

5. CHECKED COVERAGE AS AN INDICATOR FOR ORACLE QUALITY

5.1. TestValidPaths()

```
function SAXPathHandleException {
    if (0 <= path.length() < path.length() - 1) {
        var pathString = path.substring(0, path.length() - 1);
        var p = PatternParser.parse(pathString);
    }
}
```

run L: A test without outcome checks.

rule quickly, a frequently proposed approach

rule. Mutation testing sends artificial errors to the code and assesses whether the test suite low score of coverage statements implies low quality is not exceeded. In other words, low oracle effect was not checked). Unfortunately, is costly; even with recent advancements, usual work involved to weed out equivalent

rule, we introduce an alternative, cost-efficient oracle quality. Using dynamic slicing, we checked coverage—statements that are the result of a test. In Figure 1, the checked coverage is one of the easiest ways to find a run-time error in the parser. In the following, we argue that a simple assertion that the result is increases the checked coverage to 65%; adding runs on the properties of the result further increases the checked coverage to 75%. As checked coverage as a metric rather than regular significant advantages:

sufficient clauses immediately result in a low coverage, giving a more realistic assessment

that are executed, but whose outcomes are checked would be considered unexecuted. As a consequence, it would improve the oracle to actually execute them.

in focusing on executing as much code as possible, the oracle would focus on checking as many possible conditions more often. In this case, checked coverage, one only needs to run

© 2013 The Authors. Journal compilation © 2013 Association for Computing Machinery. Inc.

Focus of Our Paper

- ❖ Measuring the gap between code that is executed and code that is checked by test oracles – we call this the coverage gap
- ❖ Evaluating the impact of the coverage gap on fault-detection
- ❖ Mitigating coverage gaps by enhancing test suites to achieve better fault detection

Measuring Gaps

E ENC G

- | | E | ENC | G |
|----|---|-----|---|
| 1: | ✓ | | ✓ |
| 2: | ✓ | | ✓ |
| 3: | ✓ | | ✓ |
| 4: | ✓ | ✓ | |
| 5: | ✓ | | ✓ |
| 6: | ✓ | | ✓ |
| 7: | ✓ | | ✓ |
| 8: | ✓ | ✓ | |

```
public class Triangle {  
  
    int s1, s2, s3, P, C;  
    Triangle(int a1, int a2, int a3, int c) {  
        1:   s1 = a1;  
        2:   s2 = a2;  
        3:   s3 = a3;  
        4:   C = c; ← dashed arrow  
        5:   setPerimeter();  
    }  
  
    private void setPerimeter() {  
        6:   P = s1 + s2 + s3; ← red dot  
    }  
  
    public int getPerimeter() {  
        7:       return P;  
    }  
  
    public int getColor() {  
        8:       return C; ← dashed arrow  
    }  
}
```

```
@Test  
public void testColor() {  
    Triangle t = new Triangle(2,3,2,1);  
    t.getPerimeter();  
    assertEquals(1, t.getColor());  
}
```

Covered: 100%
Checked: 25%
In Gap: 75%

Mitigating Gaps With Recommender

```
public class Triangle {  
  
    int s1, s2, s3, P, C;  
    Triangle(int a1, int a2, int a3, int c) {  
        1:   s1 = a1; }  
        2:   s2 = a2; }  
        3:   s3 = a3; }  
        4:   C = c;  
        5:   setPerimeter();  
    }  
  
    private void setPerimeter() {  
        6:       P = s1 + s2 + s3;  
    }  
  
    public int getPerimeter() {  
        7:       return P;  
    }  
  
    public int getColor() {  
        8:       return C;  
    }  
}
```

field write: s1, s2, s3

field read: s1, s2, s3
write: P

field read: P

Recommendation

getPerimeter()

```
@Test  
public void testColor() {  
    Triangle t = new Triangle(2,3,2,1);  
    assertEquals(1, t.getColor());  
    assertEquals(7,t.getPerimeter());  
}
```

Evaluation: Artifacts

- ❖ 13 Java Applications
- ❖ 16K tests
- ❖ 51.6K test assertions

TABLE I
DESCRIPTION OF ARTIFACTS

Artifact (version)	Description	Program Size(SLOC) ¹	Test Size(SLOC) ¹	Tests(#) ²	Assertions(#) ²
Commons-Cli (1.4) [14]	Command line option parsing	2,699	3,932	372	573
Commons-Codec (1.2) [15]	Common encodings	8,352	12,182	887	1,793
Commons-Csv (1.5) [16]	CSV utilities	1,615	4,467	296	934
Commons-Lang (3.6) [17]	Java helper utilities	27,265	48,172	2,908	15,424
Commons-Validator (1.6) [18]	Data validation	7,409	8,352	536	2,486
Gson (2.8.0) [23]	JSON support	7,815	13,762	1,014	1,780
Jackson-Dataformat-Xml (2.9.10) [27]	XML processing	4,945	5,728	185	556
Jaxen (1.2.0) [30]	XPath engine	10,760	8,042	716	587
JFreeChart (1.5.0) [31]	2D Charts	97,350	39,348	2,174	5,506
Joda-Time (2.10.11) [32]	Date and time library	28,811	55,849	4,238	17,973
Jsoup (1.10.1) [33]	HTML parsing	10,785	5,499	510	1,645
Plexus-Utils (3.1.0) [10]	Utility classes	18,496	6,337	304	799
XStream (1.14.15) [11]	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 the IntelliJ statistic plugin.

² Tests are JUnit test cases annotated with @Test, and assertions are JUnit assertions.

Research Questions

- ❖ RQ1: Gaps in studied artifacts

Finding: Gaps range from 19-51 percentage points (pp), with an average of 35pp

- ❖ RQ2: Impact of gaps on fault detection

- ❖ RQ3: Recommender performance

- ❖ RQ4: Recommended assertions and fault detection effectiveness

Finding: Fault detection improved as much as 57pp and on avg. 13pp

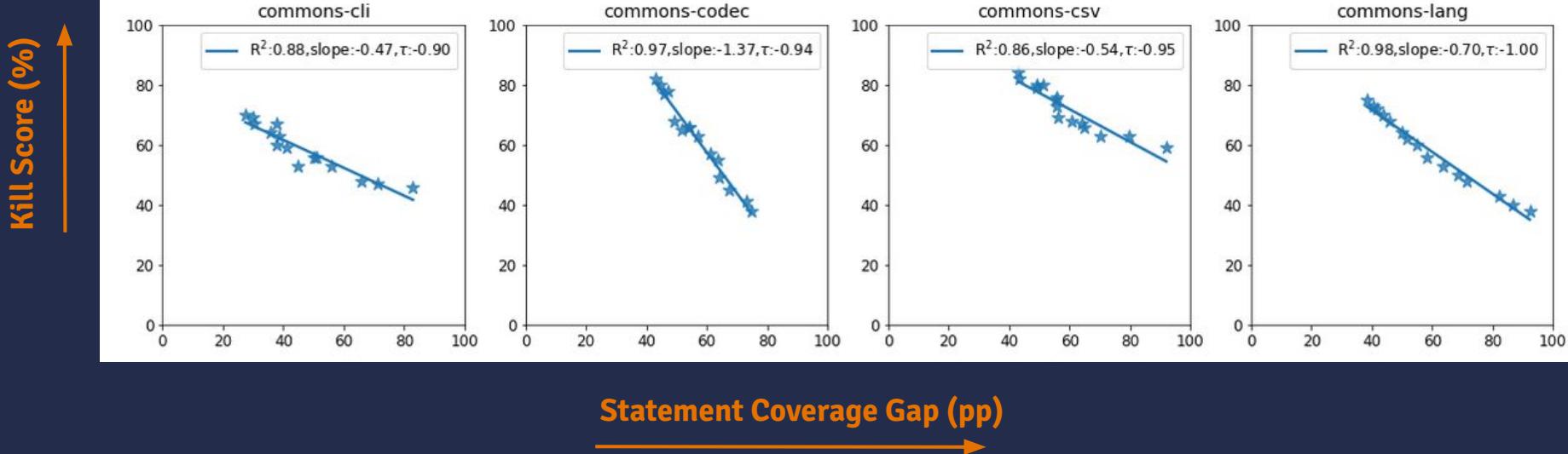
RQ2: Impact of Gaps on Fault Detection

Study Design:

- ❖ Generate 180 test suites by manipulating the gap size
- ❖ Generated 96K mutants to evaluate fault detection effectiveness
- ❖ Measure the correlation between gaps and kill scores

RQ2: Impact of Gaps on Fault Detection

Granularity: Application, Package
Criteria: Statement, Object branch



RQ2: Impact of Gaps on Fault Detection

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

RQ3: Recommender Performance

Study design:

- ❖ Remove developer written assertions from test suites
- ❖ Compute the resulting gap
- ❖ Analyze the SUT and the gap to recommend focus methods
- ❖ Compare recommended focus methods to focus methods in removed assertions

RQ3: Recommender Performance

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

RQ3: Recommender Performance

TABLE III
PERCENTAGE OF ASSERTION FOCUS METHODS RECOMMENDED
WITHIN THE TOP-K RECOMMENDATIONS ACROSS ALL 13 ARTIFACTS

Finding: 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.

Artifact	Total	Average	Average	Average
Commons-Lang	9843	80	96	98
Apache-Commons-CSV	1441	50	77	89
Jackson-Databind	92	22	42	62
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	34894	46	67	73

What next?

- ❖ Reduce the cost of gap analysis
- ❖ Automate the insertion of recommended assertions

Artifact: <https://github.com/soneyahossain/hcc-gap-recommender>



Acknowledgement: DARPA ARCOS FA8750-20-C-0507, Air Force Office of Scientific Research FA9550-21-0164, and Lockheed Martin Advanced Technology Laboratories