Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study

Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, *Member, IEEE*, Baowen Xu, Hareton Leung, *Member, IEEE*, and Zhenyu Zhang, *Member, IEEE*

Abstract—*Background*. Slice-based cohesion metrics leverage program slices with respect to the output variables of a module to quantify the strength of functional relatedness of the elements within the module. Although slice-based cohesion metrics have been proposed for many years, few empirical studies have been conducted to examine their actual usefulness in predicting fault-proneness. *Objective*. We aim to provide an in-depth understanding of the ability of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction, i.e. their effectiveness in helping practitioners find post-release faults when taking into account the effort needed to test or inspect the code. *Method*. We use the most commonly used code and process metrics, including size, structural complexity, Halstead's software science, and code churn metrics, as the baseline metrics. First, we employ principal component analysis to analyze the relationships between slice-based cohesion metrics and the baseline metrics. Then, we use univariate prediction models to investigate the correlations between slice-based cohesion metrics and post-release fault-proneness. Finally, we build multivariate prediction models to examine the effectiveness of slice-based cohesion metrics in effort-aware post-release fault-proneness prediction when used alone or used together with the baseline code and process metrics. *Results*. Based on open-source software systems, our results show that: 1) slice-based cohesion metrics are not redundant with respect to the baseline code and process metrics; 2) most slice-based cohesion metrics are significantly negatively related to post-release fault-proneness; 3) slice-based cohesion metrics in general do not outperform the baseline metrics when predicting post-release fault-proneness; and 4) when used with the baseline metrics together, however, slice-based cohesion metrics can produce a statistically significant and practically important improvement of the effectiveness in effort-aware post-release fault-proneness prediction. *Conclusion*. Slice-based cohesion metrics are complementary to the most commonly used code and process metrics and are of practical value in the context of effort-aware post-release fault-proneness prediction.

Index Terms—Cohesion, metrics, slice-based, fault-proneness, prediction, effort-aware

1 Introduction

Cohesion refers to the relatedness of the elements within a module [1], [2]. A highly cohesive module is one in which all elements work together towards a single function. Highly cohesive modules are desirable in a system as they are easier to develop, maintain, and reuse, and hence are less fault-prone [1], [2]. For software developers, it is expected to automatically identify low cohesive modules targeted for software quality enhancement. However, cohesion is a subjective concept and hence is difficult to use in practice [14]. In order to attack this problem, program slicing is applied to develop quantitative cohesion metrics, as it provides a means of accurately quantifying the interactions between the elements within a module [12]. In the last three decades, many slice-based cohesion metrics have been developed to quantify the degree of cohesion in a module at the function level of granularity [3], [4], [5], [6], [7], [8], [9], [10]. For a given function, the computation of a slice-based cohesion metric consists of the following two steps. At the first step, a program reduction technology called program slicing is employed to obtain the set of program statements (i.e. program slice) that may affect each output variable of the function [9], [11]. The output variables include the function return value, modified global variables, modified reference parameters, and variables printed or other outputs by the function [12]. At the second step, cohesion is computed by leveraging the commonality among the slices with respect to different output variables. Previous studies showed that slice-based cohesion metrics provided an excellent quantitative measure of cohesion [3], [13], [14]. Hence, there is a reason to believe that they should be useful predictors for fault-proneness. However, few empirical studies have so far been conducted to examine the actual usefulness of slice-based cohesion metrics for predicting fault-proneness, especially compared with

- *Y. Yang, Y. Zhou, H. Lu, L. Chen, and B. Xu are with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China. E-mail: yangyibiao@smail.nju.edu.cn, {zhouyuming, hmlu, lchen, bwxu}@nju.edu.cn.*
- *Z. Chen is with the State Key Laboratory for Novel Software Technology, School of Software, Nanjing University, Nanjing 210023, China. E-mail: zychen@software.nju.edu.cn.*
- *H. Leung is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong, China. E-mail: cshleung@inet.polyu.edu.hk.*
- *Z. Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: zhangzy@ios.ac.cn.*

Manuscript received 16 Feb. 2013; revised 24 Oct. 2014; accepted 29 Oct. 2014. Date of publication 11 Nov. 2014; date of current version 17 Apr. 2015. Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2370048

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing

Yibiao Yang*, Yuming Zhou*, Hao Sun†, Zhendong Su‡§, Zhiqiang Zuo*, Lei Xu*, and Baowen Xu*
*State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China
†Unaffiliated
‡Department of Computer Science, ETH Zurich, Switzerland
§Computer Science Department, UC Davis, USA

***Abstract*—Reliable code coverage tools are critically important as it is heavily used to facilitate many quality assurance activities, such as software testing, fuzzing, and debugging. However, little attention has been devoted to assessing the reliability of code coverage tools. In this study, we propose a randomized differential testing approach to hunting for bugs in the most widely used C code coverage tools. Specifically, by generating random input programs, our approach seeks for inconsistencies in code coverage reports produced by different code coverage tools, and then identifies inconsistencies as potential code coverage bugs. To effectively report code coverage bugs, we addressed three specific challenges: (1) How to filter out duplicate test programs as many of them triggering the same bugs in code coverage tools; (2) how to automatically reduce large test programs to much smaller ones that have the same properties; and (3) how to determine which code coverage tools have bugs? The extensive evaluations validate the effectiveness of our approach, resulting in 42 and 28 confirmed/fixed bugs for gcov and llvm-cov, respectively. This case study indicates that code coverage tools are not as reliable as it might have been envisaged. It not only demonstrates the effectiveness of our approach, but also highlights the need to continue improving the reliability of code coverage tools. This work opens up a new direction in code coverage validation which calls for more attention in this area.**

***Index Terms*—Code Coverage; Differential Testing; Coverage Tools; Bug Detection.**

I. Introduction

Code coverage [1] refers to which code in the program and how many times each code is executed when running on the particular test cases. The code coverage information produced by code coverage tools is widely used to facilitate many quality assurance activities, such as software testing, fuzzing, and debugging [1]–[15]. For example, researchers recently introduced an EMI ("Equivalence Modulo Inputs") based compiler testing technique [7]. The equivalent programs are obtained by stochastically pruning the unexecuted code of a given program according to the code coverage information given by the code coverage tools (e.g., llvm-cov, gcov). Therefore, the correctness of "equivalence" relies on the reliability of code coverage tools.

In spite of the prevalent adoption in practice and extensive testing of code coverage tools, a variety of defects still remain. Fig. 1(a) shows a buggy code coverage report produced by llvm-cov [16], a C code coverage tool of Clang [17]. Note that all the test cases have been reformatted for presentation in this study. The coverage report is an annotated version of source code, where the first and second column list the line number and the execution frequency, respectively. We can see that the code at line 5 is marked incorrectly as unexecuted by

```
1 |   | #include <stdio.h>
2 |   | int main()
3 | 1 | {
4 | 1 |   int g=0, v=1;
5 | 0 |   g = v || !v;
6 | 1 |   printf("%d\n", g);
7 | 1 | }
```

(a)

```
#include <stdio.h>
int main()
{
  int g=0, v=1;
  // g = v || !v;
  printf("%d\n", g);
}
```

(b)

Fig. 1. (a) Bug #33465 of llvm-cov and (b) The "equivalent" program obtained by pruning the unexecuted code (Line #4) of the program in (a)

llvm-cov. Given the program *p* and its corresponding code coverage as shown in Fig. 1(a), EMI compiler testing [7] generates its "equivalent" program *p'* as shown in Fig. 1(b) by removing unexecuted code (statement 5). The program *p* and *p'* will be compiled by a compiler under testing and then executed to obtain two different outputs, i.e. 1 and 0, resulting in a bug reported by the EMI approach. However, this is obviously not a real compiler bug. The incorrect code coverage report leads to the false positive in compiler testing. As the code coverage tools offer the fundamental information needed during the whole process of software development, it is essential to validate the correctness of code coverage. Unfortunately, to our best knowledge, little attention has been devoted to assessing the reliability of code coverage tools.

This work makes the first attempt in this direction. We devise a practical randomized differential testing approach to discovering bugs in code coverage tools. Our approach firstly leverages programs generated by a random generator to seek for inconsistencies of code coverage reports produced by different code coverage tools, and identifies inconsistencies as potential code coverage bugs. Secondly, due to the existence of too many inconsistency-triggering test programs reported and a large portion of irrelevant code within these test programs, reporting these inconsistency-triggering tests directly is hardly beneficial to debugging. Before reporting them, the reduction to those test programs is required [18]. However, it is usually very costly and thus unrealistic to reduce every and each of the test programs [18]. We observe that many test programs trigger the same coverage bugs. Thus, we can filter out many duplicate test programs. Note that 'duplicate test programs' in this study indicates multiple test programs triggering the same code coverage bug. Overall, to effectively report coverage bugs, we need to address the following key challenges:

Challenge 1: Filtering Out Test Programs. To filter out potential test programs triggering the same code coverage bugs, the most intuitive way is to calculate similarities between

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00061

Automatic Self-Validation for Code Coverage Profilers

Yibiao Yang*†, Yanyan Jiang*, Zhiqiang Zuo*, Yang Wang*,
Hao Sun*, Hongmin Lu*, Yuming Zhou*, and Baowen Xu*
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
†School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
{yangyibiao, jyy, zqzuo}@nju.edu,cn, dz1933028@smail.nju.edu.cn
shqking@gmail.com, {hmlu, zhouyuming, bwxu}@nju.edu.cn

***Abstract*—Code coverage as the primitive dynamic program behavior information, is widely adopted to facilitate a rich spectrum of software engineering tasks, such as testing, fuzzing, debugging, fault detection, reverse engineering, and program understanding. Thanks to the widespread applications, it is crucial to ensure the reliability of the code coverage profilers.**

Unfortunately, due to the lack of research attention and the existence of testing oracle problem, coverage profilers are far away from being tested sufficiently. Bugs are still regularly seen in the widely deployed profilers, like gcov and llvm-cov, along with gcc and llvm, respectively.

This paper proposes `Cod`, an automated self-validator for effectively uncovering bugs in the coverage profilers. Starting from a test program (either from a compiler's test suite or generated randomly), `Cod` detects profiler bugs with zero false positive using a metamorphic relation in which the coverage statistics of that program and a mutated variant are bridged.

We evaluated `Cod` over two of the most well-known code coverage profilers, namely gcov and llvm-cov. Within a four-month testing period, a total of 196 potential bugs (123 for gcov, 73 for llvm-cov) are found, among which 23 are confirmed by the developers.

***Index Terms*—Code coverage, Metamorphic testing, Coverage profilers, Bug detection.**

I. Introduction

Profiling code coverage data [1] (e.g., executed branches, paths, functions, etc.) of the instrumented subject programs is the cornerstone of a rich spectrum of software engineering practices, such as testing [2], fuzzing [3], debugging [4]–[6], specification mining [7], [8], fault detection [9], reverse engineering, and program understanding [10]. Incorrect coverage information would severely mislead developers in their software engineering practices.

Unfortunately, coverage profilers themselves (e.g., gcov and llvm-cov) are prone to errors. Even a simple randomized differential testing technique exposed more than 70 bugs in coverage profilers [11]. The reasons are two-fold. Firstly, neither the application-end developers nor academic researchers paid sufficient attention to the testing of code coverage profilers. Secondly, automatic testing of coverage profilers is still challenging due to the lack of test oracles. During the code coverage testing, the oracle is supposed to constitute the rich execution information, e.g., the execution frequency of each code statement in the program under a given particular test case. Different from the functional oracle which usually can be obtained via the given specification, achieving the complete code coverage oracles turns out to be extremely challenging. Even though the programming experts can specify the oracle precisely, it requires enormous human intervention, making it impractical.

A simple differential testing approach `C2V` tried to uncover coverage bugs by comparing the coverage profiling results of the same input program over two different profiler implementations (e.g., gcov and llvm-cov) [11]. For instance, if gcov and llvm-cov provide different coverage information for the same statement of the profiled program, a bug is reported. Due to the inconsistency of coverage semantics defined by different profiler implementations, it is rather common that independently implemented coverage profilers exhibit different opinions on the code-line based statistics (e.g., the case in Figure 1) — this essentially contradicts the fundamental assumption of differential testing that distinct coverage profilers should output identical coverage statistics for the same input program.

Approach To tackle the flaws of the existing approach, this paper presents `Cod`, a fully automated *self-validator* of coverage profilers, based on the metamorphic testing formulation [12]. Instead of comparing outputs from two independent profilers, `Cod` takes a single profiler and a program $\mathcal{P}$ (either from a compiler's test suite or generated randomly) as input and uncovers the bugs by identifying the inconsistency of coverage results from $\mathcal{P}$ and its equivalent mutated variants whose coverage statistics are expected to be *identical*. The equivalent program variants are generated based on the assumption that *modifying unexecuted code blocks should not affect the coverage statistics of executed blocks under the identical profiler*, which should generally hold in a non-optimized setting[1]. This idea originates from EMI [2], a metamorphic testing approach which is targeted at compiler optimization bugs.

Specifically, assuming that the compiler is correct[2] and given a deterministic program $\mathcal{P}$ under profiling (either from a compiler's test suite or generated randomly) and fixate its input, `Cod` obtains a reference program $\mathcal{P}'$ by removing the unexecuted statements in $\mathcal{P}$. $\mathcal{P}'$ should strictly follow the same execution path as long as the coverage profiling data of $\mathcal{P}$ is correct. Therefore, `Cod` asserts that the coverage statistics should be exactly the same over all unchanged statements

[1]According to the developers [13], coverage statistics are only stable under zero optimization level.

[2]We assume this because mis-compilations are rare.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00018

Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models

Yibiao Yang[1], Yuming Zhou[1*], Jinping Liu[1], Yangyang Zhao[1], Hongmin Lu[1], Lei Xu[1], Baowen Xu[1], and Hareton Leung[2]

[1]Department of Computer Science and Technology, Nanjing University, China

[2]Department of Computing, Hong Kong Polytechnic University, Hong Kong, China

ABSTRACT

Unsupervised models do not require the defect data to build the prediction models and hence incur a low building cost and gain a wide application range. Consequently, it would be more desirable for practitioners to apply unsupervised models in effort-aware just-in-time (JIT) defect prediction if they can predict defect-inducing changes well. However, little is currently known on their prediction effectiveness in this context. We aim to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction, especially compared with the state-of-the-art supervised models in the recent literature. We first use the most commonly used change metrics to build simple unsupervised models. Then, we compare these unsupervised models with the state-of-the-art supervised models under cross-validation, time-wise-cross-validation, and across-project prediction settings to determine whether they are of practical value. The experimental results, from open-source software systems, show that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware JIT defect prediction.

CCS Concepts

•**Software and its engineering** → **Risk management;** Software development process management;

Keywords

Defect, prediction, changes, just-in-time, effort-aware

1. INTRODUCTION

Recent years have seen an increasing interest in just-in-time (JIT) defect prediction, as it enables developers to identify defect-inducing changes at check-in time [7, 13]. A defect-inducing change is a software change (i.e. a single or several consecutive commits in a given period of time) that introduce one or several defects into the source code in a software system [37]. Compared with traditional defect prediction at module (e.g. package, file, or class) level, JIT defect prediction is a fine granularity defect prediction. As stated by Kamei et al. [13], it allows developers to inspect an order of magnitude smaller number of SLOC (source lines of code) to find latent defects. This could provide large savings in effort over traditional coarser granularity defect predictions. In particular, JIT defect prediction can be performed at check-in time [13]. This allows developers to inspect the code changes for finding the latent defects when the change details are still fresh in their minds. As a result, it is possible to find the latent defects faster. Furthermore, compared with conventional non-effort-aware defect prediction, effort-aware JIT defect prediction takes into account the effort required to inspect the modified code for a change [13]. Consequently, effort-aware JIT defect prediction would be more practical for practitioners, as it enables them to find more latent defects per unit code inspection effort. Currently, there is a significant strand of interest in developing effective effort-aware JIT defect prediction models [7, 13].

Kamei et al. [13] leveraged supervised method (i.e. the linear regression method) to build an effort-aware JIT defect prediction model. To the best of our knowledge, this is the first time to introduce effort-aware concept into JIT defect prediction. Their results showed that the proposed supervised model was effective in effort-aware performance evaluation compared with the random model. This work is significant, as it could help find more defect-inducing changes per unit code inspection effort. In practice, however, it is often time-consuming and expensive to collect the defect data (used as the dependent variable) to build supervised models. Furthermore, for many new projects, the defect data are unavailable, in which supervised models are not applicable. Different from supervised models, unsupervised models do not need the defect data to build the defect prediction models. Therefore, for practitioners, it would be more desirable to apply unsupervised models if they can predict defects well. According to recent studies [16, 17, 18, 19, 26, 42], simple unsupervised models, such as the ManualUp model in which modules are prioritized in ascending order according to code size, are effective in the context of effort-aware defect prediction at coarser granularity. Up till now, however, little is known on the practical value of simple unsupervised models in the context of effort-aware JIT defect prediction.

The main contributions of this paper are as follows:

*Corresponding author: zhouyuming@nju.edu.cn.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950353

An Empirical Study on Dependence Clusters for Effort-Aware Fault-Proneness Prediction

Yibiao Yang[1], Mark Harman[2], Jens Krinke[2], Syed Islam[3], David Binkley[4], Yuming Zhou[1*], and Baowen Xu[1]

[1]Department of Computer Science and Technology, Nanjing University, China
[2]Department of Computer Science, University College London, UK
[3]School of Architecture, Computing and Engineering, University of East London, UK
[4]Department of Computer Science, Loyola University Maryland, USA

ABSTRACT

A dependence cluster is a set of mutually inter-dependent program elements. Prior studies have found that large dependence clusters are prevalent in software systems. It has been suggested that dependence clusters have potentially harmful effects on software quality. However, little empirical evidence has been provided to support this claim. The study presented in this paper investigates the relationship between dependence clusters and software quality at the function-level with a focus on effort-aware fault-proneness prediction. The investigation first analyzes whether or not larger dependence clusters tend to be more fault-prone. Second, it investigates whether the proportion of faulty functions inside dependence clusters is significantly different from the proportion of faulty functions outside dependence clusters. Third, it examines whether or not functions inside dependence clusters playing a more important role than others are more fault-prone. Finally, based on two groups of functions (i.e., functions inside and outside dependence clusters), the investigation considers a segmented fault-proneness prediction model. Our experimental results, based on five well-known open-source systems, show that (1) larger dependence clusters tend to be more fault-prone; (2) the proportion of faulty functions inside dependence clusters is significantly larger than the proportion of faulty functions outside dependence clusters; (3) functions inside dependence clusters that play more important roles are more fault-prone; (4) our segmented prediction model can significantly improve the effectiveness of effort-aware fault-proneness prediction in both ranking and classification scenarios. These findings help us better understand how dependence clusters influence software quality.

*Corresponding author: zhouyuming@nju.edu.cn.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970353

CCS Concepts

•**Software and its engineering → Abstraction, modeling and modularity;** Software development process management;

Keywords

Dependence clusters, fault-proneness, fault prediction, network analysis

1. INTRODUCTION

A dependence cluster is a set of program elements that all directly or transitively depend upon one another [8, 18]. Prior empirical studies found that large dependence clusters are highly prevalent in software systems and further complicate many software activities such as software maintenance, testing, and comprehension [8, 18]. In the presence of a (large) dependence cluster, an issue or a code change in one element likely has significant ripple effects involving the other elements of the cluster [8, 18]. Hence, there is a reason to believe that dependence clusters have potentially harmful effects on software quality. This suggests that the elements inside dependence clusters have relatively lower quality when compared to elements outside any dependence cluster. Given this observation, dependence clusters should be useful in fault-prediction. However, few empirical studies have investigated the effect of dependence clusters on fault-proneness prediction.

This paper presents an empirical study of the relationships between dependence clusters and fault-proneness. The concept of a dependence cluster was originally introduced by Binkley and Harman [8]. They treat program statements as basic units, however, they note that dependence clusters can be also defined at coarser granularities, such as at the function-level [7]. For a given program, the identification of function-level dependence clusters consists of two steps. The first step generates a function-level System Dependence Graph for all functions of the program. In general, these graphs involve two types of dependencies between functions: call dependency (i.e., one function calls another function) and data dependency (e.g., a global variable defined in one function is used in another function). In the System Dependence Graphs used in our study, nodes denote functions and directed edges denote the dependencies between these functions. In the second step, a clustering algorithm is used

Information and Software Technology 75 (2016) 90–104

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An empirical investigation into the effect of slice types on slice-based cohesion metrics

Yibiao Yang, Yangyang Zhao, Changsong Liu, Hongmin Lu, Yuming Zhou*, Baowen Xu

State Key Laboratory for Novel Software Technology, Nanjing University, China

ARTICLE INFO

Article history:
Received 2 June 2015
Revised 28 February 2016
Accepted 1 April 2016
Available online 12 April 2016

Keywords:
Cohesion
End slice
Metric slice
Metrics

ABSTRACT

Context: There is a debate about whether end slice or metric slice is preferable for computing slice-based cohesion metrics. However, up till now, there is no consensus about this issue.
Objective: We aim to investigate the relationship between end-slice-based and metric-slice-based cohesion metrics and then determine which type of slice is preferable for computing slice-based cohesion metrics.
Method: We used forty widely used open-source software systems to conduct the study. First, we compute the baseline values for end-slice-based and metric-slice-based cohesion metrics. Then, we investigate their relationships with module size. Finally, we employ correlation analysis and principal component analysis to analyze the relationships between end-slice-based and metric-slice-based cohesion metrics.
Results: End-slice-based and metric-slice-based cohesion metrics have similar baseline metric values. Furthermore, they exhibit a similar negative correlation with module size. In particular, the results from correlation analysis and principal component analysis reveal that they essentially measure the same cohesion dimensions.
Conclusion: From the viewpoint of metric values, there is little difference between end-slice-based and metric-slice-based cohesion metrics. We hence recommend choosing end slice for computing slice-based cohesion metrics in practice, as extra cost involved in data collection could be avoided.

1. Introduction

Cohesion is an important software quality attribute which refers to the tightness of elements within a module [1,2]. Highly cohesive modules are desirable in a system as they are easier to develop, maintain, and reuse, and hence are less fault-prone [1,2]. In the last three decades, researchers developed many slice-based cohesion metrics to quantify the cohesion of a module at the function level of granularity [3–10]. In [11], Meyers and Binkley found that slice-based cohesion metrics could be used to quantify the deterioration that accompanies software evolution. More recently, our study shows that slice-based cohesion metrics are not redundant with respect to the most commonly used code/process metrics and are of practically important value in the context of fault-proneness prediction [12]. Therefore, slice-based cohesion metrics can be used as an important quality indicator for practitioners. In other words, practitioners could use these metrics to identify potentially faulty modules for quality enhancement.

For a given function, the computation of a slice-based cohesion metric consists of the following three steps. The first step is to identify the output variables of the function, including function return values, modified global variables, printed variables, and modified reference parameters [13]. The second step is to employ program slicing techniques to obtain one slice with respect to each output variable of the function [9,14]. The third step is to use the resulting slices to compute the slice-based cohesion metric. Of these three steps, the second step is the most crucial, as it is the basis for the computation of slice-based cohesion metrics. In the literature, there is a debate about whether end slice or metric slice should be used for computing slice-based cohesion metrics [11,12,15]. An end slice with respect to variable v is the *backward slice* with respect to v at the end point of the module [9,16]. A metric slice with respect to variable v is the union of the *backward slice* with respect to v at the end point of the module and the *forward slice* computed from the top of the backward slice [7]. Consequently, a metric slice is more time-consuming to compute compared with an end slice. In most previous studies, end slice was used to compute slice-based cohesion metrics [5–9,11,15–18]. However, Ott and her colleagues argued that metric slice can result in more accurate cohesion metrics [10]. Up till now, there is

* Corresponding author. Tel.: +86 25 89682450.
E-mail address: cs.zhou.yuming@gmail.com (Y. Zhou).

http://dx.doi.org/10.1016/j.infsof.2016.04.001
0950-5849/

How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction

YUMING ZHOU, YIBIAO YANG, HONGMIN LU, LIN CHEN, YANHUI LI, and YANGYANG ZHAO, Nanjing University
JUNYAN QIAN, Guilin University of Electronic Technology
BAOWEN XU, Nanjing University

Background. Recent years have seen an increasing interest in cross-project defect prediction (CPDP), which aims to apply defect prediction models built on source projects to a target project. Currently, a variety of (complex) CPDP models have been proposed with a promising prediction performance.
Problem. Most, if not all, of the existing CPDP models are not compared against those simple module size models that are easy to implement and have shown a good performance in defect prediction in the literature.
Objective. We aim to investigate how far we have really progressed in the journey by comparing the performance in defect prediction between the existing CPDP models and simple module size models.
Method. We first use module size in the target project to build two simple defect prediction models, ManualDown and ManualUp, which do not require any training data from source projects. ManualDown considers a larger module as more defect-prone, while ManualUp considers a smaller module as more defect-prone. Then, we take the following measures to ensure a fair comparison on the performance in defect prediction between the existing CPDP models and the simple module size models: using the same publicly available data sets, using the same performance indicators, and using the prediction performance reported in the original cross-project defect prediction studies.
Result. The simple module size models have a prediction performance comparable or even superior to most of the existing CPDP models in the literature, including many newly proposed models.
Conclusion. The results caution us that, if the prediction performance is the goal, the real progress in CPDP is not being achieved as it might have been envisaged. We hence recommend that future studies should include ManualDown/ManualUp as the baseline models for comparison when developing new CPDP models to predict defects in a complete target project.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software**;

Additional Key Words and Phrases: Defect prediction, cross-project, supervised, unsupervised, model

This work is partially supported by the National Key Basic Research and Development Program of China (2014CB340702) and the National Natural Science Foundation of China (61432001, 61772259, 61472175, 61472178, 61702256, 61562015, 61403187).

Authors' addresses: Y. Zhou (corresponding author), Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, and B. Xu (corresponding author), State Key Laboratory for Novel Software Technology, Nanjing University, No. 163, Xianlin Road, Nanjing, 210023, Jiangsu Province, P.R. China; emails: {zhouyuming, yangyibiao, hmlu, lchen, yanhuili, bwxu}@nju.edu.cn, csurjzhyy@163.com; Y. Qian, Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, 541004, Guangxi Province, P.R. China; email: qjy2000@gmail.com.

https://doi.org/10.1145/3183339

2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00065

Isolating Compiler Optimization Faults via Differentiating Finer-grained Options

Jing Yang*†, Yibiao Yang‡§, Maolin Sun*†, Ming Wen*†, Yuming Zhou‡§, Hai Jin†¶
*Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering
Huazhong University of Science and Technology, Wuhan, China
†National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab
Huazhong University of Science and Technology, Wuhan, China
‡State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
§Department of Computer Science and Technology, Nanjing University, China
¶Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, China
{yjing, merlinsun, mwenaa, hjin}@hust.edu.cn, {yangyibiao, zhouyuming}@nju.edu.cn

***Abstract*—Code optimization is an essential feature for compilers and almost all software products are released by compiler optimizations. Consequently, bugs in code optimization will inevitably cast significant impact on the correctness of software systems. Locating optimization bugs in compilers is challenging as compilers typically support a large amount of optimization configurations. Although prior studies have proposed to locate compiler bugs via generating witness test programs, they are still time-consuming and not effective enough. To address such limitations, we propose an automatic bug localization approach, `ODFL`, for locating compiler optimization bugs via differentiating finer-grained options in this study. Specifically, we first disable the fine-grained options that are enabled by default under the bug-triggering optimization levels independently to obtain bug-free and bug-related fine-grained options. We then configure several effective passing and failing optimization sequences based on such fine-grained options to obtain multiple failing and passing compiler coverage. Finally, such generated coverage information can be utilized via Spectrum-Based Fault Localization formulae to rank the suspicious compiler files. We run `ODFL` on 60 buggy GCC compilers from an existing benchmark. The experimental results show that `ODFL` significantly outperforms the state-of-the-art compiler bug isolation approach RecBi in terms of all the evaluated metrics, demonstrating the effectiveness of `ODFL`. In addition, `ODFL` is much more efficient than RecBi as it can save more than 88% of the time for locating bugs on average.**

***Index Terms*—Compiler, Bug Isolation, Fault Localization, Finer-grained, Optimization Option**

I. Introduction

Compilers are the most fundamental infrastructures in software development. Bugs hidden in compilers will inevitably cast significant impacts on the correctness of the compiled software systems. The correctness of compilers is thus of critical importance. However, being complex software systems, compilers themselves are prone to errors. Consequently, many compiler bugs are being reported every day. Among those bugs, compiler optimization bugs account for the largest proportion [1]. Thus, one of the most important tasks for developers is to locate and fix optimization bugs in compilers. Nevertheless, locating compiler bugs is challenging as compilers are one of the largest software systems. Driven by this, it is of significant importance to advance the techniques for locating bugs of compiler optimizations.

Recently, Chen et al. proposed DiWi [2] and RecBi [3] to facilitate compiler bug isolation. They introduced a novel concept of witness programs, which are mutated from the bug-triggering test programs of the compiler. A mutated test program is considered as a witness program when it does not trigger any bugs of the compiler. The witness test programs can be viewed as the passing test programs in the **S**pectrum-**B**ased **F**ault **L**ocalization (SBFL) techniques [4], [5]. Specifically, DiWi utilizes traditional local mutation operators and Metropolis-Hastings (MH) [6] algorithm to find effective witness programs, while RecBi mutates the structure of programs and utilizes the reinforcement learning [7] algorithm to select programs. Then, both of them leverage the SBFL formula, Ochiai [8], to locate compiler bugs by comparing the coverage information between the bug-triggering program and the generated passing programs.

Although these witness test programs based approaches are effective for compiler bug localization, they still encounter inevitable limitations. First, the goal of the existing approaches is to generate witness test program (i.e., passing test program) which will not produce multiple failing coverage of compilers. Second, generating passing test programs is time-consuming as it is hard to obtain a witness test program without a large amount of attempts in program mutation. Third, the localization results are not reliable due to the randomness of the generated programs. To mitigate this influence, the experiment must be repeated multiple times to obtain relatively reliable results. For example, in GCC bug 58570, we found that the position of the buggy file located by RecBi floated from 21st to 47th among five different experiments. Forth, diverse mutated programs and coverage information is hard to obtained for effective localization when the given failing test program has a simple program structure. Besides, there is little difference between the results of RecBi and DiWi (see more details in Section V), indicating that due to the limited diversity of the coverage information of the mutation

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00065

CBUA: A Probabilistic, Predictive, and Practical Approach for Evaluating Test Suite Effectiveness

Peng Zhang, Yanhui Li, Wanwangying Ma, Yibiao Yang, Lin Chen, Hongmin Lu, Yuming Zhou, and Baowen Xu, *Member, IEEE*

Abstract—Knowing the effectiveness of a test suite is essential for many activities such as assessing the test adequacy of code and guiding the generation of new test cases. Mutation testing is a commonly used defect injection technique for evaluating the effectiveness of a test suite. However, it is usually computationally expensive, as a large number of mutants (buggy versions) are needed to be generated from a production code under test and executed against the test suite. In order to reduce the expensive testing cost, recent studies proposed to use supervised models to predict the effectiveness of a test suite without executing the test suite against the mutants. Nonetheless, the training of such a supervised model requires labeled data, which still depends on the costly mutant execution. Furthermore, existing models are based on traditional supervised learning techniques, which assume that the training and testing data come from the same distribution. But, in practice, software systems are subject to considerable concept drifts, i.e., the same distribution assumption usually does not hold. This can lead to inaccurate predictions of a learned supervised model on the target code as time progresses. To tackle these problems, in this paper, we propose a Coverage-Based Unsupervised Approach (CBUA) for evaluating the effectiveness of a test suite. Given a production code under test, the corresponding mutants, and a test suite, CBUA first collects the coverage information of the mutated statements in the target production code under the execution of the test suite. Then, CBUA employs coverage to estimate the probability of each mutant being alive. As such, a mutation score is computed to evaluate the test suite effectiveness and the predicted labels (i.e., killed or alive) are obtained. The whole process only requires a one-time execution of the test suite against the target production code, without involving any mutant execution and any training data. CBUA can ensure the score monotonicity property (i.e., adding test cases to a test suite does not decrease its mutation score), which may be violated by a supervised approach. The experimental results show that CBUA is very competitive with the state-of-the-art supervised approaches in prediction accuracy. In particular, CBUA is shown to be more effective in finding mutants that are covered but not killed by a test suite, which is helpful in identifying the weaknesses in the current test suite and generating new test cases accordingly. Since CBUA is an easy-to-implement approach with a low cost, we suggest that it should be used as a baseline approach for comparison when any novel prediction approach is proposed in future studies.

Index Terms—Effectiveness, test suites, coverage, unsupervised model, mutation testing

1 Introduction

1.1 Motivation

In software testing, test suits are widely used to validate the correctness of software. Knowing the effectiveness (i.e., the fault detecting potential) of a test suite is essential for many activities such as assessing the test adequacy of code and guiding the generation of new test cases. Consequently, an immense amount of research effort so far has been devoted to developing an effective approach to evaluating the effectiveness of test suites [1], [2], [3], [4]. Mutation testing is widely considered as one of the most effective approaches t at finding faults [5], [6], [7], [8]. In mutation testing, a set of mutants (i.e., faulty variants) are generated by modifying a production code under test (CUT) in small ways (i.e., by applying mutation operators). A mutant is regarded as killed by a test suite if at least one test case in the test suite leads to the behavior of CUT to differ from the mutant. Otherwise, the mutant is regarded as survived (i.e., alive) in the test suite, which means that the test suite cannot detect bugs in the mutated source code. The ratio of killed mutants to all the non-equivalent mutants is referred as mutation score, which is used to assess test effectiveness. The higher the mutation score, the more effective the test suite is considered. In the literature [9], [46], mutation score is a commonly used indicator to evaluate the effectiveness of a given test suite.

Although mutation score is a good metric for measuring the effectiveness of a test suite in detecting defects, its computation can be expensive, especially for a large CUT [9], [46]. The reason is that a large number of mutants (buggy versions) are needed to be generated and executed against the test suite. In order to reduce the computation cost, recent studies used supervised models to predict the mutation score without executing the test suite against the mutants. Jalbert and Bradbury [10], [11] used a number of features extracted from both CUT and test code to build support vector machine (SVM) models to predict whether the resulting mutation score was "high", "medium", or "low". These features included static features (structural metrics collected from both production and test code) and dynamic features (code coverage metrics collected from production code when

• *The authors are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: {dz1833034, wwyma}@smail.nju.edu.cn,{yanhuili,yangyibiao,lchen,zhouyuming,bwxu}@nju.edu.cn, 32493172@qq.com.*

Manuscript received 7 Aug. 2019; revised 19 May 2020; accepted 13 July 2020. Date of publication 20 July 2020; date of current version 15 Mar. 2022. (Corresponding authors: Yanhui Li, Lin Chen, Yuming Zhou, and Baowen Xu) Recommended for acceptance by Raymond K. Sen. Digital Object Identifier no. 10.1109/TSE.2020.3010361

Mutant Reduction Evaluation: What is There and What is Missing?

PENG ZHANG, YANG WANG, XUTONG LIU, YANHUI LI, and YIBIAO YANG,
Nanjing University
ZIYUAN WANG, Nanjing University of Posts and Telecommunications
XIAOYU ZHOU, Southeast University
LIN CHEN and YUMING ZHOU, Nanjing University

Background. Mutation testing is a commonly used defect injection technique for evaluating the effectiveness of a test suite. However, it is usually computationally expensive. Therefore, many mutation reduction strategies, which aim to reduce the number of mutants, have been proposed.

Problem. It is important to measure the ability of a mutation reduction strategy to maintain test suite effectiveness evaluation. However, existing evaluation indicators are unable to measure the "order-preserving ability", i.e., to what extent the mutation score order among test suites is maintained before and after mutation reduction. As a result, misleading conclusions can be achieved when using existing indicators to evaluate the reduction effectiveness.

Objective. We aim to propose evaluation indicators to measure the "order-preserving ability" of a mutation reduction strategy, which is important but missing in our community.

Method. Given a test suite on a **Software Under Test (SUT)** with a set of original mutants, we leverage the test suite to generate a group of test suites that have a partial order relationship in defect detecting ability. When evaluating a reduction strategy, we first construct two partial order relationships among the generated test suites in terms of mutation score, one with the original mutants and another with the reduced mutants. Then, we measure the extent to which the partial order under the original mutants remains unchanged in the partial order under the reduced mutants. The more partial order is unchanged, the stronger the **Order Preservation (*OP*)** of the mutation reduction strategy is, and the more effective the reduction strategy is. Furthermore, we propose **Effort-aware Relative Order Preservation (*EROP*)** to measure how much gain a mutation reduction strategy can provide compared with a random reduction strategy.

Result. The experimental results show that OP and EROP are able to efficiently measure the "order-preserving ability" of a mutation reduction strategy. As a result, they have a better ability to distinguish various mutation reduction strategies compared with the existing evaluation indicators. In addition, we find

This work is partially supported by the National Natural Science Foundation of China (61772259, 62172205, 61872177, 62072194, 62172202).

Authors' addresses: P. Zhang, Y. Wang, X. Liu, Y. Li (corresponding author), Y. Yang, L. Chen (corresponding author), and Y. Zhou (corresponding author), Department of Computer Science and Techonology Nanjing University 163 Xianlin Avenue, Qixia District Nanjing, Jiangsu Province, China, 210023; emails: {dz1833034, njuwy, xryu}@smail.nju.edu.cn, {yanhuili, yangyibiao, lchen, zhouyuming}@nju.edu.cn; Z. Wang, School of Computer Science Nanjing University of Posts and Telecommunications 9 Wenyuan Road, Qixia District Nanjing, Jiangsu Province, China, 210023; email: wangziyuan@njupt.edu.cn; X. Zhou, School of Computer Science and Engineering Souteast University 2 Souteast University, Jiangning District Nanjing, Jiangsu Province, China, 211189; email: zhouxy@seu.edu.cn.

1049-331X/2022/07-ART69 $15.00
https://doi.org/10.1145/3522578