Information and Software Technology 76 (2016) 65–80

Contents lists available at ScienceDirect

Information and Software Technology

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

B-Refactoring: Automatic test code refactoring to improve dynamic analysis

Jifeng Xuan[a,*], Benoit Cornu[b,c], Matias Martinez[b,c], Benoit Baudry[c], Lionel Seinturier[b,c], Martin Monperrus[b,c]

[a] State Key Lab of Software Engineering, Wuhan University, China
[b] University of Lille, France
[c] INRIA, France

ARTICLE INFO

Article history:
Received 28 May 2015
Revised 6 April 2016
Accepted 28 April 2016
Available online 29 April 2016

ABSTRACT

Context: Developers design test suites to verify that software meets its expected behaviors. Many dynamic analysis techniques are performed on the exploitation of execution traces from test cases. In practice, one test case may imply various behaviors. However, the execution of a test case only yields one trace, which can hide the others.

Objective: In this article, we propose a new technique of test code refactoring, called B-Refactoring. The idea behind B-Refactoring is to split a test case into small test fragments, which cover a simpler part of the control flow to provide better support for dynamic analysis.

Method: For a given dynamic analysis technique, B-Refactoring monitors the execution of test cases and constructs small test cases without loss of the testability. We apply B-Refactoring to assist two existing analysis tasks: automatic repair of `if`-condition bugs and automatic analysis of exception contracts.

Results: Experimental results show that B-Refactoring can effectively improve the execution traces of the test suite. Real-world bugs that could not be previously fixed with the original test suites are fixed after applying B-Refactoring; meanwhile, exception contracts are better verified via applying B-Refactoring to original test suites.

Conclusions: We conclude that applying B-Refactoring improves the execution traces of test cases for dynamic analysis. This improvement can enhance existing dynamic analysis tasks.

1. Introduction

Developers design and write test suites to automatically verify that software meets its expected behaviors. For instance, in regression testing, the role of a test suite is to catch new bugs – the regressions – after changes [40]. Test suites are used in a wide range of dynamic analysis techniques: in fault localization, a test suite is executed for inferring the location of bugs by reasoning on code coverage [19]; in invariant discovery, input points in a test suite are used to infer likely program invariants [10]; in software repair, a test suite is employed to verify the behavior of synthesized patches [23]. Many dynamic analysis techniques are based on the exploitation of execution traces obtained by each test case [5,10,40].

Different types of dynamic analysis techniques require different types of traces. The accuracy of dynamic analysis depends on the structure of those traces, such as length, diversity, redundancy, etc. For example, several traces that cover the same paths with different input values are very useful for discovering program invariants [10]; fault localization benefits from traces that cover different execution paths [5] and that are triggered by assertions in different test cases [54]. However, in practice, one manually-written test case results in one single trace during test suite execution; test suite execution traces can be optimal with respect to test suite comprehension (from the human viewpoint by authors of the test suite) but might be suboptimal with respect to other criteria (from the viewpoint of dynamic analysis techniques).

Test code refactoring is a family of methods, which improve test code via program transformation without changing behaviors of the test code [49]. In this article, we propose a new kind of test code refactoring, which focuses on the design of test cases, directly for improving dynamic analysis techniques. Instead of having a

* Corresponding author. Tel.: +8618674053457.
E-mail address: jxuan@whu.edu.cn (J. Xuan).

http://dx.doi.org/10.1016/j.infsof.2016.04.016
0950-5849/

single test suite used for many analysis tasks, *our hypothesis is that a system can automatically optimize the design of a test suite with respect to the requirements of a given dynamic analysis technique*. For instance, given an original test suite, developers can have an optimized version with respect to fault localization as well as another optimized version with respect to automatic software repair. This optimization can be made on demand for a specific type of dynamic analysis. The optimized test suite is used as the input of dynamic analysis without manual checking by developers.

In this paper, we propose a novel automated test code refactoring system dedicated to dynamic analysis, called B-Refactoring,[1] detects and splits impure test cases. In our work, an *impure test case* is a test case, which executes an unprocessable path in one dynamic analysis technique. The idea behind B-Refactoring is to split a test case into small "test fragments", where *each fragment is a completely valid test case and covers a simple part of the control flow*; test fragments after splitting provide better support for dynamic analysis. A *purified* test suite after applying B-Refactoring does not change the test behaviors of the original one: it triggers exactly the same set of behaviors as the original test suite and detects exactly the same bugs. However, it produces a different set of execution traces. This set of traces suits better for the targeted dynamic program analysis. Note that our definition of purity is specific to test cases and is completely different from the one used in the programming language literature (e.g., [50]).

A purified test suite after applying B-Refactoring can be employed to temporarily replace the original test suite in a given dynamic analysis technique. Based on such replacement, performance of dynamic analysis can be enhanced. To evaluate our approach B-Refactoring, we consider two dynamic analysis techniques, one in the domain of automatic software repair [9,52] and the other in the context of dynamic verification of exception contracts [8]. We briefly present the case of software repair here and present in details the dynamic verification of exception contracts in Section 5.2.2. For software repair, we consider Nopol [52], an automatic repair system for bugs in `if` conditions. Nopol employs a dynamic analysis technique that is sensitive to the design of test suites. The efficiency of Nopol depends on whether the same test case executes both `then` and `else` branches of an `if`. This forms a refactoring criterion that is given as input to B-Refactoring. In our dataset, we show that B-Refactoring improves the test execution on `if`s and *unlocks new bugs which are able to be fixed by purified test suites*.

Prior work. Our work [54] shows that traces by an original test suite are suboptimal with respect to *fault localization*. The original test suite is updated to enhance the usage of *assertions* in fault localization. In the current article, the goal and technique are different, B-Refactoring refactors the whole *test suite* according to a *given dynamic analysis technique*. Section 6.2 explain the differences between the proposed technique in this article and our prior work.

This article makes the following major contributions:

- We formulate the problem of automatic test code refactoring for dynamic analysis. The concept of pure and impure test cases is generalized to any type of program element.
- We propose B-Refactoring, an approach to automatically refactoring test code according to a specific criterion. This approach detects and refactors impure test cases based on analyzing execution traces. The test suite after refactoring consists of smaller test cases that do not reduce the potential of bug detection.
- We apply B-Refactoring to assist two existing dynamic analysis tasks from the literature: automatic repair of `if`-condition bugs and automatic analysis of exception contracts. Three real-world bugs that could not be fixed with original test suites are empirically evaluated after B-Refactoring; exception contracts are better verified by applying B-Refactoring to original test suites.

The remainder of this article is organized as follows. In Section 2, we introduce the background and motivation of B-Refactoring. In Section 3, we define the problem of refactoring test code for dynamic analysis and propose our approach B-Refactoring. In Section 4.2, we evaluate our approach on five open-source projects; in Section 5, we apply the approach to automatic repair and exception contract analysis. Section 6 details discussions and threats to the validity. Section 7 lists the related work and Section 8 concludes our work. Section Appendix describes two case studies of repairing real-world bugs.

2. Background and motivation

In this section, we present one scenario where test code refactoring improves the automatic repair of `if`-condition bugs. However, test code refactoring is a generic concept and can be applied prior to other dynamic analysis techniques beyond software repair. Another application scenario in the realm of exception handling can be found in Section 5.2.2.

2.1. Real-world example in automatic repair: Apache commons math 141473

In test suite based repair, a repair method generates a patch for potentially buggy statements according to a given test suite [[23,33,52]. The research community of test suite based repair has developed fruitful results, such as GenProg by Le Goues et al. [23], Par by Kim et al. [21], and SemFix by Nguyen et al. [33]. In this article, we automatically refactor the test suite to improve the ability of constructing a patch.

We start this section with a real-world bug in open source project, Apache Commons Math, to illustrate the motivation of our work. *Apache Commons Math* is a Java library of mathematics and statistics components.[2]

Fig. 1 shows a code snippet of this project. It consists of a bug in an `if` and two related test cases.[3] The program in Fig. 1a is designed to calculate the factorial, including two methods: `factorialDouble` for the factorial of a real number and `factorialLog` for calculating the natural logarithm of the factorial. The bug, at Line 11, is that the `if` condition `n<=0` should actually be `n<0`.

Fig. 1 b displays two test cases that execute the buggy `if` condition: a passing one and a failing one. The failing test case detects that a bug exists in the program while the passing test case validates the existing correct behavior. To generate a patch, a repair method needs to analyze the executed branches of an `if` by each test case. Note that an `if` statement with only a `then` branch, such as Lines 11 to 14 in Fig. 1a, can be viewed as an `if` with a `then` branch and an empty `else` branch.

As shown in Fig. 1b, we can observe that test code before Line 14 in test case `testFactorial` executes the `then` branch while test code after Line 15 executes the `else` branch. The fact that a single test case executes several branches is a problem for certain automatic repair algorithms such as Nopol [52] described in Section 2.2.

[1] B-Refactoring is short for Banana-Refactoring. We name our approach with *Banana* because we split a test case as splitting a banana in the ice cream named Banana Split.

[2] Apache Commons Math, http://commons.apache.org/math/.

[3] See https://fisheye6.atlassian.com/changelog/commons?cs=141473.