HAL
archives-ouvertes.fr

B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis

Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, Martin Monperrus

▶ **To cite this version:**

Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, et al.. B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis. Information and Software Technology, Elsevier, 2016, 76, pp.65-80. <10.1016/j.infsof.2016.04.016>. <hal-01309004>

HAL Id: hal-01309004
https://hal.archives-ouvertes.fr/hal-01309004

Submitted on 6 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

STRUCTURED 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.

I. 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 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

[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.

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 V-B. 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 VI-B 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 II, we introduce the background and motivation of B-Refactoring. In Section III, we define the problem of refactoring test code for dynamic analysis and propose our approach B-Refactoring. In Section IV, we evaluate our approach on five open-source projects; in Section V, we apply the approach to automatic repair and exception contract analysis. Section VI details discussions and threats to the validity. Section VII lists the related work and Section VIII concludes our work. Section Appendix describes two case studies of repairing real-world bugs.

II. 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 V-B.

A. 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]

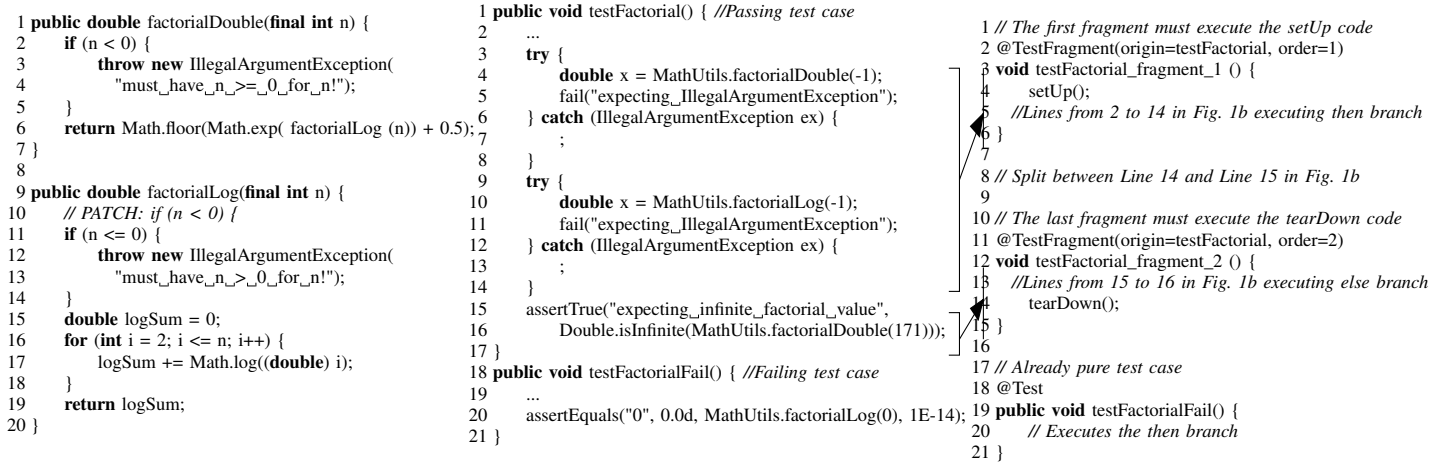
Fig. 1 shows a code snippet of this project. It consists of a bug in an `if` and two related test cases.[3] The program in Fig. 1a is designed to calculate the factorial, including two methods: `factorialDouble` for the factorial of a real number and `factorialLog` for calculating the natural logarithm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

Fig. 1b 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`

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

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



```
1 public double factorialDouble(final int n) {
2     if (n < 0) {
3         throw new IllegalArgumentException(
4           "must have n >= 0 for n!");
5     }
6     return Math.floor(Math.exp( factorialLog (n)) + 0.5);
7 }
8
9 public double factorialLog(final int n) {
10    // PATCH: if (n < 0) {
11    if (n <= 0) {
12        throw new IllegalArgumentException(
13          "must have n > 0 for n!");
14    }
15    double logSum = 0;
16    for (int i = 2; i <= n; i++) {
17        logSum += Math.log((double) i);
18    }
19    return logSum;
20 }
```

(a) Buggy program

```
1 public void testFactorial() { //Passing test case
2     ...
3     try {
4         double x = MathUtils.factorialDouble(-1);
5         fail("expecting IllegalArgumentException");
6     } catch (IllegalArgumentException ex) {
7         ;
8     }
9     try {
10        double x = MathUtils.factorialLog(-1);
11        fail("expecting IllegalArgumentException");
12    } catch (IllegalArgumentException ex) {
13        ;
14    }
15    assertTrue("expecting infinite factorial value",
16        Double.isInfinite(MathUtils.factorialDouble(171)));
17 }
18 public void testFactorialFail() { //Failing test case
19     ...
20     assertEquals("0", 0.0d, MathUtils.factorialLog(0), 1E-14);
21 }
```

(b) Two original test cases

```
1 // The first fragment must execute the setUp code
2 @TestFragment(origin=testFactorial, order=1)
3 void testFactorial_fragment_1 () {
4     setUp();
5   //Lines from 2 to 14 in Fig. 1b executing then branch
6 }
7
8 // Split between Line 14 and Line 15 in Fig. 1b
9
10 // The last fragment must execute the tearDown code
11 @TestFragment(origin=testFactorial, order=2)
12 void testFactorial_fragment_2 () {
13   //Lines from 15 to 16 in Fig. 1b executing else branch
14     tearDown();
15 }
16
17 // Already pure test case
18 @Test
19 public void testFactorialFail() {
20     // Executes the then branch
21 }
```

(c) Three test cases after refactoring

Fig. 1: Example of refactoring a test suite. The buggy program and test cases are extracted from Apache Commons Math. The buggy `if` is at Line 11 of Fig. 1a. A test case `testFactorial` in Fig. 1b executes both `then` (at Line 10 of Fig. 1b) and `else` (at Line 15 of Fig. 1b) branches of the `if` (at Line 11 of Fig. 1a). Fig. 1c shows the test cases after the splitting (between Lines 14 and 15) according to the execution on branches.

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 II-B.

B. Automatic Software Repair with Nopol

In test suite based repair, a test suite drives the patch generation. We consider an existing test-suite based repair approach called Nopol [52]. Nopol focuses on fixing bugs in `if` conditions. To generate a patch for an `if` condition, Nopol *requires test cases have to cover either the `then` branch or the `else` branch, exclusively*.

However in practice, there are cases where one test case covers both `then` and `else` branches together. This fact results in an ambiguous behavior with respect the repair algorithm of Nopol. In the best case, the repair approach discards this test case and continues the repair process with the remaining test cases; in the worst case, the repair approach cannot fix the bug because discarding the test case leads to a lack of test cases.

Let us consider again, the example of Fig. 1b. Is there any way to split this test case into two parts according to the execution of branches? Fig. 1c shows two test cases after splitting the test case `testFactorial` between Lines 14 and 15.[4] Based on the test cases after splitting, Nopol works well and is able to generate a correct patch as expected. This necessary test case splitting motivates our work: we aim refining a test case to cover simpler parts of the control flow during program execution.

[4]Note that in Fig. 1c, the first two test cases after splitting have extra annotations like `@TestFragment` at Line 2 as well as extra code like `setUp` at Line 4 and `tearDown` at Line 14. We add these lines to facilitate the test execution, which will be introduced in Section III-C.

In this article, we propose to refactor the test suite to fix bugs that are unfixed because of the structure of the test suite. Consider the example in Fig. 1. We apply our test code refactoring technique to obtain simple test cases, as shown in Fig. 1b. The test suite after refactoring in Fig. 1c can make Nopol generate a correct patch that fixes the bug.

III. B-Refactoring: A Test Code Refactoring Technique

In this section, we present the basic concepts of B-Refactoring, and important technical aspects.

A. Basic Concepts

Definition 1. (program element, test constituent) In this article, a *program element* denotes an entity in the code of a program, in opposition to a *test constituent*, which denotes an entity in the code of a test case.

We use the terms *element* and *constituent* for sake of being always clear whether we refer to the application program or its test suite. Any node in an Abstract Syntax Tree (AST) of the program (resp. the test suite) can be considered as a program element (resp. a test constituent). For example, an *`if` element* and a *`try` element* denote an `if` statement and a `try` statement in Java programs, respectively.[5] We consider a test case t as a sequence of test constituents, i.e., $t = \langle c_1, c_2, \ldots, c_n \rangle$.

Definition 2. (execution domain) Let E be a set of program elements in the same type of AST nodes. The *execution domain* D of a program element $e \in E$ is a set of code that characterizes one execution of e.

[5]We follow existing work on Java program analysis [17] and call `if` and `try` statements.

For instance, for an `if` element, the execution domain can be defined as

$$D_{\texttt{if}} = \{\texttt{then-branch}, \texttt{else-branch}\}$$

where `then-branch` and `else-branch` are the execution of the `then` branch and the `else` branch, respectively.

The execution domain is a generic concept. Besides `if`, let us give three examples of other execution domains as follows: the execution domain of a method invocation $func(var_a, var_b, ...)$ is $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ where $\mathbf{x}_i$ is a vector of actual arguments in a method invocation (potentially infinite); the execution domain of `switch-case` is $\{case_1, case_2, \ldots, case_n\}$ where $case_i$ is a `case` in the `switch`.

For `try` elements, we define the execution as follows

$$D_{\texttt{try}} = \{\texttt{no-exception}, \texttt{exception-caught}, \texttt{exception-not-caught}\}$$

where `no-exception`, `exception-caught`, and `exception-not-caught` are the execution results of `try` element: no exception is thrown, one exception is caught by the catch block, and one exception is thrown in the catch block but not caught, respectively. The execution domain of `try` will be used in dynamic verification of exception handling in Section V-B.

Values in an execution domain D are mutually exclusive: a single execution of a program element is uniquely classified in D. During the execution of a test case, a program element $e \in E$ may be executed multiple times.

Definition 3. (execution signature) We refer to an execution result of a program element as an *execution signature*.

A *pure execution signature* denotes the execution of a program element, which yields a single value in an execution domain D, e.g., only the `then` branch of `if` is executed by one given test case t. An *impure execution signature* denotes the execution of a program element with multiple values in D, e.g., both `then` and `else` branches are executed by t. Given an execution domain, let $D_0 = \{\text{impure}\}$ be the set of impure execution signatures.

The execution signature with respect to an element e by a test case t is the aggregation of each value as follows. Let T be the test suite, the set of all test cases, we define

$$f : E \times T \rightarrow D \cup D^0 \cup \{\bot\}$$

where $\bot$ (usually called "bottom") denotes that the test case t does not execute the program element e. For example, $f(e,t) \in D_0$ indicates both `then` and `else` branches of an `if` element e are executed by a test case t. If a test case executes the same element always in the same way (e.g., the test case always executes `then` in an `if` element), we call it *pure*. Note that for the simplest case, a set of program elements may consist of only one program element.

Let C denote the set of test constituent c_i $(1 \leq i \leq n)$. Then the above function $f(e,t)$ can be refined for the execution of a test constituent $c \in C$. A function g gives the purity of a program element according to a test constituent:

$$g : E \times C \rightarrow D \cup D^0 \cup \{\bot\}$$

A test constituent c is pure on E if and only if $(\forall e \in E)\ g(e,c) \in D \cup \{\bot\}$; c is impure on E if and only if $(\exists e \in E)\ g(e,c) \in D^0$.

For example, consider a statement in a test case as a test constituent; then a method call (also a statement) that executes both `then` and `else` branches, is an impure constituent for `if` elements. If we consider a top-level statement in a test case as a test constituent, then a `while` loop that executes both `then` and `else`, is an impure constituent for `if` elements.

Definition 4. (test impurity) Given a set E of program elements and a test case $t \in T$, let us define the impurity indicator function $\delta : \mathcal{E} \times T$, where $\mathcal{E}$ is a set of all the candidate sets of program elements. In details, $\delta(E,t) = 0$ if and only if the test case t is *pure* (on the set E of program elements) while $\delta(E,t) = 1$ if and only if t is *impure*. Formally,

$$\delta(E,t) = \begin{cases} 0 & \text{pure, iff } (\forall e \in E)\ f(e,t) \in D \cup \{\bot\} \\ 1 & \text{impure, iff } (\exists e \in E)\ f(e,t) \in D^0 \end{cases}$$

At the test constituent level, the above definition of purity and impurity of a test case can be stated as follows. A test case t is pure if the following holds

$$(\exists x \in D)\ (\forall e \in E)\ (\forall c \in C)\ g(e,c) \in \{x\} \cup \{\bot\}$$

A test case t is impure if t contains either at least one impure constituent or at least two different execution signatures on constituents. That is, either of the following holds

$$(\exists e \in E)(\exists c \in C) g(e,c) \in D^0, \text{ or}$$

$$\exists e \in E, \exists c_1, c_2 \in C\,(g(e,c_1) \neq g(e,c_2)) \wedge (g(e,c_1), g(e,c_2) \in D)$$

An *absolutely impure test case* according to a set E of program elements is a test case, for which there exists at least one impure test constituent: $(\exists e \in E)\ (\exists c \in C)\ g(e,c) \in D^0$.

Definition 5. (pure coverage) A program element e is *purely covered* according to a test suite T if all test cases yield pure execution signatures: $(\forall t \in T)\ f(e,t) \notin D^0$. A program element e is impurely covered according to T if any test case yields an impure execution signature: $(\exists t \in T)\ f(e,t) \in D^0$. This concept will be used to indicate the purity of test cases in Section IV.

Note that the above definitions are independent of the number of assertions per test case. Even if there is a single assertion, the code before the assertion may explore the full execution domain of certain program elements.

B. B-Refactoring

Test code refactoring aims to rearrange test cases according to a certain task [49], [29], [7]. In this article, we present B-Refactoring, a type of test code refactoring that aims to

TABLE I: Example of three test fragments and the execution signature of an `if` element.

Test constituent	c_1	c_2	c_3	c_4	c_5	c_6	c_7
Execution signature	$\perp$	`then-branch`	$\perp$	`else-branch`	$\perp$	`else-branch`	`then-branch`
Test fragment	$\langle c_1, c_2, c_3 \rangle$			$\langle c_4, c_5, c_6 \rangle$			$\langle c_7 \rangle$

minimize the number of impure test cases in a test suite. Our definition of purity involves a set of program elements, hence there are multiple kinds of feasible refactoring, depending on the considered program elements. For instance, developers can purify a test suite with respect to a set of `if`s or with respect to a set of `try`s, etc.

Based on Definition 4, the task of test code refactoring for a set E of program elements is to find a test suite T that minimizes the amount of impurity as follows:

$$\min \sum_{t \in T} \delta(E, t) \tag{1}$$

The minimum of $\sum_{t \in T} \delta(E, t)$ is 0 when all test cases in T are pure. As shown later, this is usually not possible in practice. Note that, in this article, we do not aim to find the absolutely optimal purified test suite, but a test suite that improves dynamic analysis techniques. An impure test case can be split into a set of smaller test cases that are possibly pure.

Definition 6. (test fragment) A *test fragment* is a continuous sequence of test constituents. Given a set of program elements and a test case, i.e., a continuous sequence of test constituents, a *pure test fragment* is a test fragment that includes only pure constituents.

Ideally, an impure test case without any impure test constituent can be split into a sequence of pure test fragments, e.g., a test case consisting of two test constituents, which covers `then` and `else` branches, respectively. Given a set E of program elements and an impure test case $t = \langle c_1, \ldots, c_n \rangle$ where $(\forall e \in E)\ g(e, c_i) \in D \cup \{\perp\}$ $(1 \leq i \leq n)$, we can split the test case into a set of m test fragments. Let φ_j be the jth test fragment $(1 \leq j \leq m)$ in t. Let c_j^k denote the kth test constituent in φ_j and $|\varphi_j|$ denote the number of test constituents in φ_j. We define φ_j as a continuous sequence of test constituents as follows,

$$\varphi_j = \langle c_j^1, c_j^2, \ldots, c_j^{|\varphi_j|} \rangle$$

where $(\exists x \in D)\ (\forall e \in E)\ c_j^k \in \{x\} \cup \{\perp\}$ and $1 \leq k \leq |\varphi_j|$.

Based on the above definitions, given a test case without impure test constituents, the goal of B-Refactoring is to generate a minimized number of pure test fragments.

1) Example of B-Refactoring: In the best case, an impure test case can be refactored into a set of test fragments as above. Table I presents an example of B-Refactoring for a test case with seven test constituents $t = \langle c_1, c_2, c_3, c_4, c_5, c_5, c_6, c_7 \rangle$ that are executed on a set of `if` elements consisting of only one `if` element. Three test fragments are formed as $\langle c_1, c_2, c_3 \rangle$, $\langle c_4, c_5, c_6 \rangle$, and $\langle c_7 \rangle$.

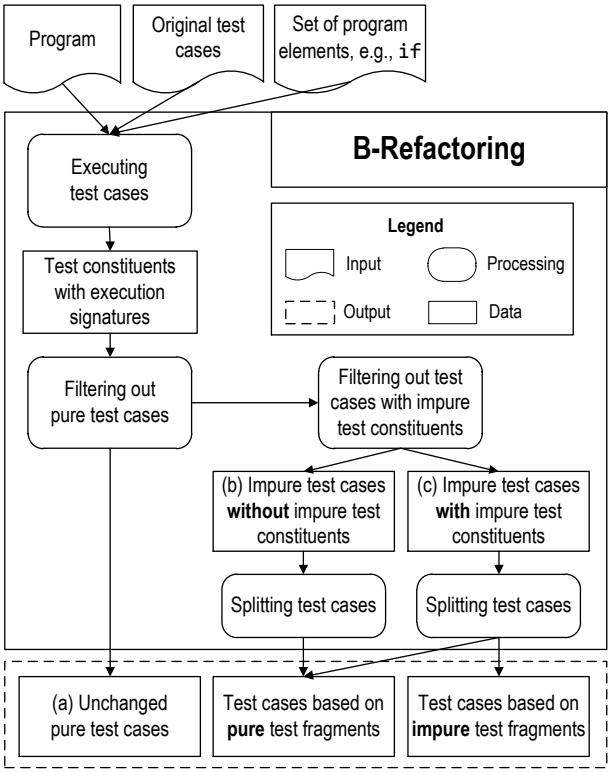


Fig. 2: Conceptual framework of B-Refactoring. This framework takes a program with test cases and a specific set of program elements (e.g., `if` elements) as input; the output is new test cases based on test fragments. The sum of test cases in (a), (b), and (c) equals to the number of original test cases.

Note that the goal of B-Refactoring is not to replace the original test suite, but to temporarily refactor the test suite to enhance dynamic analysis techniques. B-Refactoring is done on-demand, just before executing a specific dynamic analysis. Consequently, it has no impact on future maintenance of test cases. In particular, new test cases potentially created by B-Refactoring are not required to be read or modified by developers. The difference between our work and test code refactoring in general is discussed in Section VI-A.

2) Framework: Our method B-Refactoring refactors a test suite according to a criterion defined with a set of specific program elements (e.g., `if` elements) in order to purify its execution (according to the execution signatures in Section III-A). In a nutshell, B-Refactoring takes the original test suite and the requested set of program elements as input and generates purified test cases as output.

Fig. 2 illustrates the overall structure of our approach. We first instrument all test cases to monitor the test execution on the requested set E of program elements. During the test execution, we record test cases that execute E and collect the execution signatures of test constituents in the recorded test cases. Second, we filter out pure test cases that already exist

in the test suite. Third, we divide the remaining test cases into two categories: test cases with or without impure test constituents. For each category, we split the test cases into a set of test fragments. As a result, a new test suite is created, whose execution according to a set of program elements is purer than the execution of the original test suite. Note that it is not mandatory to divide test cases into categories with or without impure constituents; we make such division in Fig. 2 to show that tests with impure constituents lead to both pure and impure fragments while tests without impure constituents only lead to pure fragments. Implementation details are stated in Section III-C.

In this article, we consider a test constituent as a top-level statement in a test case. Examples of test constituents could be an assignment, a complete loop, a `try` block, a method invocation, etc. B-Refactoring does not try to split the statements that are inside a loop or a `try` branch in a test case.

3) Core Algorithm: Algorithm 1 describes how B-Refactoring splits a test case into a sequence of test fragments. As mentioned in Section III-B2, the input is a test case and a set of program elements to be purified; the output is a set of test fragments after splitting the original test case.

Algorithm 1 returns a minimized set of pure test fragments and a set of impure test fragments. In the algorithm, each impure test constituent is kept and directly transformed as an atomically impure test case that consists of only one constituent. The remaining continuous test constituents are clustered into several pure test fragments. Algorithm 1 consists of two major steps. First, we traverse all the test constituents to collect the last test constituent of each test fragment. Second, based on such collection, we split the test case into pure or impure test fragments. These test fragments can be directly treated as test cases for a dynamic analysis application.

Taking the test case in Table I as an example, we briefly describe the process of Algorithm 1. The traversal at Line 3 consists of only one program element according to Table I. If only one of the `then` and `else` branches is executed, we record this branch for the following traversal of the test case (at Line 12). If a test constituent with a new execution signature appears, its previous test constituent is collected as the last constituent of a test fragment and the next test fragment is initialized (at Line 14). That is, c_3 and c_6 in Table I are collected as the last constituents. The end constituent of the test case is collected as the last constituent of the last test fragment (at Line 20), i.e., c_7. Lines from 7 to 9 are not run because there is no impure test constituent in Table I. After the traversal of all the test constituents, Lines from 23 to 25 are executed to obtain the final three test fragments based on the collection of c_3, c_6, and c_7.

4) Validation of the Refactored Test Suite: Our algorithm for refactoring a test suite is meant to not hurt the ability of finding bugs. The test suite after refactoring should be as effective as the original test suite. Existing work on general refactoring uses precondition checking to validate the program

Input :
E, a set of program elements;
$t = \langle c_1, \ldots, c_n \rangle$, a test case with n test constituents;
D, an execution domain of the program elements in E.
Output:
Φ, a set of test fragments.

```
1  Let ℂ be an empty set of last constituents in fragments;
2  Let v = ⊥ be a default execution signature;
3  foreach program element e ∈ E do
4      v = ⊥;
5      foreach test constituent c_i in t (1 ≤ i ≤ n) do
6          if g(e, c_i) ∈ D^0 then   // Impure constituent
7              v = ⊥;
8              ℂ = ℂ ∪ c_i − 1; // End of the previous fragment
9              ℂ = ℂ ∪ c_i; // Impure fragment of one constituent
10         else if g(e, c_i) ∈ D then   // Pure constituent
11             if v = ⊥ then
12                 v = g(e, c_i);
13             else if v ≠ g(e, c_i) then   // v ∈ D
14                 ℂ = ℂ ∪ c_{i−1};
15                 v = g(e, c_i);
16             end
17         end
18     end
19 end
20 ℂ = ℂ ∪ c_n;  // Last constituent of the last fragment
21 Let c^+ = c_1;
22 foreach test constituent c_j in ℂ do
23     φ = ⟨c^+, ..., c_j⟩; // Creation of a test fragment
24     Φ = Φ ∪ φ;
25     c^+ = c_{j+1};
26 end
```

Algorithm 1: Splitting a test case into a set of test fragments according to a given set of program elements.

behavior before and after refactoring [34], [41]. In our work, we apply several techniques of precondition checking to avoid potential changes of program behaviors or compilation errors. Two major methods of precondition checking in our work are as follows.

First, we check the name conflicts of newly created variables. In our work, newly created variables are automatically renamed in a unique way. Based on the code analysis library, Spoon [35], we are aware of the list of existing method and variable names before test code refactoring and the conflicts are handled. Since our technique refactors test cases for dynamic analysis, no readability is required for the new names of methods or variables.

Second, we check expected test behaviors in test cases. In JUnit, a test case with expected behaviors will pass if the test execution outputs an exception, which is the same as the expected one (annotated with `@Test(expected=Exception.class)`). Splitting a test

case with the above annotation may change the original test result. In our implementation, we keep such test cases unchanged.

However, it is challenging to prove that the above pre-condition checking are enough to guarantee the behavioral preservation in a semantically rich and complex programming language as Java. Instead of a proof, we use mutation testing to raise the confidence that our refactoring approach does not hurt the effectiveness of the original test suite [18]. The idea of applying mutation testing is that all mutants killed by the original test suite must also be killed by the refactored one. Since in practice, it is impossible to enumerate all mutants, this validation is an approximation that compares the effectiveness of test suites before and after B-Refactoring. We present the validation results in Section IV-D.

C. Implementation

We implement B-Refactoring in Java 1.7, JUnit 4.11, and Spoon. Spoon is a Java library for source code transformation and analysis [35], which provides a static analysis platform for extracting the structure of test classes. With the support of instrumentation mentioned in Section III-B2, the purity of test constituents is collected; based on the source code transformation by Spoon, test cases can be rewritten without compiling errors. Our tool, B-Refactoring, is publicly available.[6]

B-Refactoring handles a number of interesting cases and uses its own test driver to take them into account. Four major details are listed as follows.

1) Test Transformation: As mentioned in Section III-B2, we treat top-level statements in a test case as test constituents. If one original test case is split into more than one test fragments, these new ones are named with the indexes of new fragments (as shown in Fig. 1c). When test fragments use variables that are local to the original test case before refactoring, these variables are changed as fields of the test class to maintain their accessibility.

2) Execution Order: To ensure the execution order of test fragments, the B-Refactoring test driver uses a specific annotation `@TestFragment(origin, order)` to execute test fragments in a correct order. The parameter `origin` is a string, which indicates the original method name before refactoring and `order` is a 1-based integer, which indicates the execution order. The execution order is assigned according to the original test case before refactoring. Then test methods are automatically tagged with the annotation during refactoring. Examples of this annotation are shown in Fig. 1c.

3) Handling `setUp` and `tearDown`: Unit testing can make use of common setup and finalization code. JUnit 4 uses Java annotations to facilitate writing this code. For each test case, a `setUp` method (with the annotation `@Before` in JUnit 4) and a `tearDown` method (with `@After`) are executed before and after the test case, e.g., initializing a local variable before the execution of the test case and resetting a variable after the execution, respectively. In B-Refactoring, to ensure the same execution of a given test case before and after refactoring, we include `setUp` and `tearDown` methods in the first and the last test fragments. This is illustrated in Fig. 1c.

4) Shared Variables in a Test Case: Some variables in a test case may be shared by multiple statements, e.g., one common variable in two assertions. In B-Refactoring, to split a test case into multiple ones, a shared variable in a test case is renamed and extracted as a class field. Then each new test case can access this variable; meanwhile, the behavior of the original test case is not changed. Experiments in Section IV-D also confirm the unchanged behavior of test cases.

[6]B-Refactoring, http://github.com/Spirals-Team/banana-refactoring.

IV. Empirical Study on B-Refactoring

In this section, we evaluate our technique for test code refactoring. This work addresses a novel problem statement: refactoring a test suite to enhance dynamic analysis. To our knowledge, there is no similar technique that can be used to compare against. However, a number of essential research questions have to be answered.

A. Projects

We evaluate B-Refactoring on five open-source Java projects: Apache Commons Lang (Lang for short),[7] Spojo-core,[8] Jbehave-core,[9] Apache Shindig Gadgets (Shindig-gadgets for short),[10] and Apache Commons Codec (Codec for short).[11] These projects are all under the umbrella of respectful code organizations (three out of five projects by Apache[12]). Table II lists these five projects. We select these five projects in experiments because they are widely-used open-source projects. The example of project Apache Commons Math presented in Section II-A is discarded since the time of its test execution is extremely long.

B. Empirical Observation on Test Case Purity

RQ1: What is the purity of test cases in our dataset?

We empirically study the purity of test cases for two types of program elements: `if` elements and `try` elements. The goal of this empirical study is to measure the existing purity of test cases for `if` and `try` before refactoring. The analysis for `if` facilitates the study on software repair in Section V-A while the analysis for `try` facilitates the study on dynamic verification of exception contracts in Section V-B. We show that applying B-Refactoring can improve the purity for individual program elements in Section IV-C.

[7]Apache Commons Lang 3.2, http://commons.apache.org/lang/.
[8]Spojo-core 1.0.6, http://github.com/sWoRm/Spojo.
[9]Jbehave-core, http://jbehave.org/.
[10]Apache Shindig Gadgets, http://shindig.apache.org/.
[11]Apache Commons Codec 1.9, http://commons.apache.org/codec/.
[12]Apache Software Foundation, http://apache.org/.

TABLE II: Projects in empirical evaluation.

Project	Description	Source LoC	#Test cases
Lang	A Java library for manipulating core classes	65,628	2,254
Spojo-core	A rule-based transformation tool for Java beans	2,304	133
Jbehave-core	A framework for behavior-driven development	18,168	457
Shindig-gadgets	A container to allow sites to start hosting social apps	59,043	2,002
Codec	A Java library for encoding and decoding	13,948	619
Total		159,091	5,465

TABLE III: Purity of test cases for `if` elements according to the number of test cases, test constituents, and `if` elements.

Project	Test case							Test constituent			`if` element			
	#Total	Pure		Non-absolutely impure		Absolutely impure		Total	Impure		#Total	#Executed	Purely covered `if`	
		#	%	#	%	#	%		#	%			#	%
Lang	2,254	539	23.91%	371	16.46%	1,344	59.63%	19,682	5,705	28.99%	2,397	2,263	451	19.93%
Spojo-core	133	38	28.57%	5	3.76%	90	67.67%	999	168	16.82%	87	79	45	56.96%
Jbehave-core	457	195	42.67%	35	7.76%	227	49.67%	3,631	366	10.08%	428	381	230	60.37%
Shindig-gadgets	2,002	731	36.51%	133	6.64%	1,138	56.84%	14,063	6,610	47.00%	2,378	1,885	1,378	73.10%
Codec	619	182	29.40%	123	19.87%	314	50.73%	3,458	1,294	37.42%	507	502	148	29.48%
Total	5,465	1,685	30.83%	667	12.20%	3,113	56.96%	41,833	14,143	33.81%	5,797	5,110	2,252	44.07%

TABLE IV: Purity of test cases for `try` elements according to the number of test cases, test constituents, and `try` elements.

Project	Test case							Test constituent			`try` element			
	#Total	Pure		Non-absolutely impure		Absolutely impure		#Total	#Impure		#Total	#Executed	Purely covered `try`	
		#	%	#	%	#	%		#	%			#	%
Lang	2,254	295	13.09%	1,873	83.1%	86	3.81%	19,682	276	1.40%	73	70	35	50.00%
Spojo-core	133	52	39.10%	81	60.9%	0	0.00%	999	0	0.00%	6	5	5	100.00%
Jbehave-core	457	341	74.62%	91	19.91%	25	5.47%	3,631	29	0.80%	67	57	43	75.44%
Shindig-gadgets	2,002	1,238	61.84%	702	35.06%	62	3.10%	14,063	73	0.52%	296	244	221	90.57%
Codec	619	88	14.22%	529	85.46%	2	0.32%	3,458	2	0.06%	18	16	14	87.50%
Total	5,465	2,014	36.85%	3,276	59.95%	175	3.20%	41,833	380	0.91%	460	392	318	81.12%

1) Protocol: We focus on the following metrics to present the purity level of test cases:

- #*Pure* is the number of pure test cases on all program elements under consideration;
- #*Non-absolutely impure* is the number of impure test cases without impure test constituent;
- #*Absolutely impure* is the number of test cases that consist of at least one impure test constituent.

The numbers of test cases in these three metrics are mapped to the three categories (a), (b), and (c) of test cases in Fig 2, respectively.

For test constituents, we use the following two metrics, i.e., *#Total constituents* and *#Impure constituents*. For program elements, we use the metric *#Purely covered program elements* (Definition 5) in Section III-A.

We leverage the implementation of B-Refactoring to calculate those evaluation metrics and to give an overview of the purity of test suites for the five projects.

2) Results: We analyze the purity of test cases in our dataset with the metrics proposed in Section IV-B1. Table III shows the purity of test cases for `if` elements. In the project Lang, 539 out of 2,254 (23.91%) test cases are pure for all the executed `if` elements while 371 (16.46%) and 1,344 (59.63%) test cases are impure without and with impure test constituents. In total, 1,658 out of 5,465 (30.83%) test cases are pure for the all the executed `if` elements. These results show that there is space for improving the purity of test cases and achieving a higher percentage of pure test cases.

As shown in the column *Test constituent* in Table III, 33.81% of test constituents are impure. After applying B-Refactoring, all those impure constituents will be isolated in own test fragments. That is the number of absolutely impure constituents is equal to the number of impure test cases after refactoring.

In Table III, we also present the execution purity of `if` elements. In the project Lang, 2,263 out of 2,397 `if` elements are executed by the whole test suite. Among these executed `if` elements, 451 (19.93%) are purely covered. In total, among the five projects, 44.07% of `if` elements are purely covered. Hence, it is necessary to improve the purely covered `if` elements with B-Refactoring.

For `try` elements, we use the execution domain defined in Section III-A and compute the same metrics. Table IV shows the purity of test cases for `try` elements. In Lang, 295 out of 2,254 (13.09%) test cases are always pure for all the executed `try` elements. In total, the percentage of always pure test cases and the percentage of absolutely impure test cases are 36.85%

and 3.20%, respectively. In contrast to `if` elements in Table III, the number of absolutely impure test cases in Spojo-core is zero. The major reason is that there is a much larger number of test cases in Lang (2,254), compared to Spojo-core (133). In the five projects, based on the purity of test cases according to the number of `try` elements, 81.12% `try` elements are purely covered.

Comparing the purity of test cases between `if` and `try`, the percentage of pure test cases for `if` elements and `try` elements are similar, 30.83% and 36.85%, respectively. In addition, the percentage of purely covered `try` elements is 81.12%, which is higher than that of purely covered `if`, i.e., 44.07%. That is, 81.12% of `try` elements are executed by test cases with pure execution signatures but only 44.07% of `if` elements are executed by test cases with pure execution signatures. This comparison indicates that for the same project, different execution domains of input program elements result in different results for the purity of test cases. We can further improve the purity of test cases according to the execution domain (implying a criterion for refactoring) for a specific dynamic analysis technique.

> Answer to RQ1: Only 31% (resp. 37%) of test cases are pure with respect to `if` elements (resp. `try` elements).

C. Empirical Measurement of Refactoring Quality

> RQ2: Are test cases purer on individual program elements after applying B-Refactoring?

We evaluate whether B-Refactoring can improve the execution purity of test cases. Purified test cases cover smaller parts of the control flow; consequently, they can provide better support to dynamic analysis tasks.

1) Protocol: To empirically assess the quality of our refactoring technique with respect to purity, we employ the following metric (see Definition 5): #*Purely covered program elements* is the number of program elements, each of which is covered by all test cases with pure execution signatures.

For dynamic analysis, we generally aim to obtain a higher number of purely covered program elements after B-Refactoring. For each metric, we list the number of program elements before and after applying B-Refactoring as well as the improvement: absolute and relative ($\frac{\#After-\#Before}{\#Before}$).

2) Results: Table V shows the improvement of test case purity for `if` elements before and after applying B-Refactoring. For the project Lang, 2,263 `if` elements are executed by the whole test suite. After applying B-Refactoring to the test suite, 1,250 (from 451 to 1,701) `if` elements are changed to be purely covered. The relative improvement reaches 277.16% (1,250/451). After B-Refactoring, 1,364 (5,110-3,746) `if` elements are not purely covered. The reason is that our approach cannot split all impure test cases into pure test cases due the technical details (in Section III-B2).

For all five projects, 1,494 purely covered `if` elements are obtained by applying B-Refactoring. These results indicate that the purity of test cases for `if` elements is highly improved via B-Refactoring. Note that the improvement on Lang is higher than that on the other four projects. A possible reason is that Lang is complex in implementation due to its powerful functionality. Then its test suite contains many impure test cases; the original ratio of purely covered `if` is only 19.93% (i.e., 451/2263 in Table III). Note that the original design of the test suite is only software maintenance, thus, a low ratio of purely covered `if` elements does not hurt the performance of testing. Meanwhile, after applying B-Refactoring, 1,250 `if` elements are liberated as purely covered ones. We consider the reason behind the high improvement in Lang (comparing with the other four projects in Table V) is that many original `if` elements are executed by impure test cases but not absolutely impure test cases. Hence, our work can help to highly improve the ratio of purely covered `if` elements.

TABLE V: Test case purity before and after refactoring with `ifs` as the purity criterion.

Project	#Exec `if`	Purely covered `if`			
		#Before	#After	Improvement	
				#	%
Lang	2,263	451	1,701	1,250	277.16%
Spojo-core	79	45	54	9	20.00%
Jbehave-core	381	230	262	32	13.91%
Shindig-gadgets	1,885	1,378	1,521	143	10.38%
Codec	502	148	208	60	40.54%
Total for `ifs`	5,110	2,252	3,746	1,494	66.34%

TABLE VI: Test case purity before and after refactoring with `trys` as the purity criterion.

Project	#Exec `try`	Purely covered `try`			
		#Before	#After	Improvement	
				#	%
Lang	70	35	58	23	65.71%
Spojo-core	5	5	5	0	0.00%
Jbehave-core	57	43	44	1	2.33%
Shindig-gadgets	244	221	229	8	3.62%
Codec	16	14	16	2	14.29%
Total for `trys`	392	318	352	34	10.69%

Similarly, Table VI shows the improvement for `try` elements before and after applying B-Refactoring. In Lang, 23 (from 35 to 58) `try` elements are changed to be purely covered after applying B-Refactoring. For all five projects, 34 (from 318 to 352) `try` elements change to be purely covered after B-Refactoring. Note that for Spojo-core, no value is changed before and after B-Refactoring due to the small number of test cases.

> Answer to RQ2: After B-Refactoring, `if` and `try` elements are more purely executed. The purely covered `if` and `try` are improved by 66% and 11%, respectively.

D. Mutation-based Validation for Refactored Test Suites

TABLE VII: Mutant comparison before and after applying B-Refactoring.

Project	Mutants before B-Refactoring		Mutants after B-Refactoring	
	#Killed	#Alive	#Killed	#Alive
Lang	82	18	82	18
Spojo-core	100	0	100	0
JBehave-core	62	33	62	33
Shinding-gadgets	100	0	100	0
Codec	90	10	90	10

RQ3: Does B-Refactoring maintain the fault revealing power of the test suite?

In this section, we employ mutation testing to validate that a refactored test suite does not hurt the effectiveness of the original test suite [11], [13].

1) Protocol: For each project, we generate mutants that represent bugs in the program code. A mutant is *killed* by a test suite if at least one test case fails on this mutation. To evaluate whether a refactored test suite finds the same number of bugs as the original one, the two test suites should satisfy either of the two following rules: one mutant is killed by both the original test suite and the refactor one; or one mutant is not killed by both test suites. For the sake of performance, we randomly select 100 mutants per project. We use standard mutations on boolean conditions (changes on binary operators), which are automatically generated with the support of the program analysis tool, Spoon. For each mutant, we individually run the original test suite and the purified test suite to check whether the mutant is killed. In addition, we check that the code coverage is the same. This protocol enables us to increase the confidence that the refactored test suites do not hurt the fault detection power of the original ones.

2) Results: Experimental results in Table VII shows that both the two rules in Section IV-D1 are satisfied for all the mutants. In details, all mutants that are killed by the original test suite are also killed by the refactored ones. The other mutants are alive in both original and refactored test suites, respectively. To sum up, mutation-based validation for refactored test suites shows that the refactored test suites after applying our technique have not hurt the effectiveness of finding bugs of the original ones.

Answer to RQ3: The test suites automatically refactored by B-Refactoring catch the same mutants as the original ones.

V. Applications to Dynamic Analysis

We apply B-Refactoring to improve two dynamic analysis techniques, automatic repair and exception contract analysis.

A. B-Refactoring for Automatic Repair of Three Bugs

RQ4: Does B-Refactoring improve the fixability by the automatic repair technique, Nopol [52]?

In this section, for automatic repair, we take Nopol [52], an approach to automatically fixing conditional bugs, as an example. To fix one bug, Nopol collects runtime trace of a test suite and synthesizes source code for conditional bugs to pass all test cases. To assess the value of one condition, Nopol is required to distinguish runtime conditional values of different branches, i.e., `then` and `else` branches. By applying B-Refactoring to the original test suite, more test cases that only cover one branch are temporarily provided for executing Nopol. Then Nopol can fix bugs, which are not able to be fixed with original test suites. We illustrate B-Refactoring in automatic repair with three case studies. Among these case studies, the last case study shows that even by applying B-Refactoring, a bug could be incorrectly fixed. The major reason is the presence of weak specification in test cases, which can be overcome via manually adding a test case.

1) Protocol: We present case studies on three real-world bugs in Apache Commons Lang.[13] All the three bugs are located in `if` conditions.

We choose the three bugs in case studies based on the following steps. First, we use our tool of Abstract Syntax Tree (AST) analysis, GumTree [12], to automatically extract commits, which modify existing `if` conditions. Only commits that modify no more than 5 files and no more than 10 AST changes are selected. Second, we filter out complex commits, which affect more statements than `if` conditions. Third, for each commit, we collect its test suite that is submitted at the same time of the commit. Fourth, we manually validate whether the bug can be reproduced. Then six bugs in total are collected after these four steps.

Table VIII shows these six bugs, including lines of executable code (LoC), manually-written patches by developers, and the number of test cases before and after B-Refactoring. The six collected bugs cannot be directly and correctly fixed by Nopol. Test suites of all these bugs contain impure test cases according to `if` elements. Thus, we apply B-Refactoring to obtain pure test cases. In total, 22 pure test cases are obtained after applying B-Refactoring to the 12 originally executed test cases. Note that only the executed test cases by the buggy `if`s (before and after applying B-Refactoring) are listed, not the whole test suite.

For bugs with IDs 137371, 137552, and 230921, applying B-Refactoring leads to the correct repair of these bugs via providing pure test cases to Nopol. For bugs with IDs 137231 and 825420, the failure reason is that several test cases are designed in preliminarily weak specifications. Even employing B-Refactoring, these bugs cannot be correctly fixed. For the bug with ID 1075673, Nopol cannot synthesize a patch because Nopol searches for a value for a null object, which is not available during the test execution. This bug reveals a threat to the design of Nopol, which is important to repair methods, but not relevant to impure test cases. In this paper, we choose three bugs as case studies, i.e., bugs with IDs 137371, 137552, and 825420.

[13]For more details, visit http://fisheye6.atlassian.com/changelog/commons?cs=137371, http://fisheye6.atlassian.com/changelog/commons?cs=137552, and http://fisheye6.atlassian.com/changelog/commons?cs=825420.

TABLE VIII: Evaluation of the effect of B-Refactoring on automatic repair for `if`-condition bugs. Traces of test cases after applying B-Refactoring enable a repair approach to find patches.

Case study	ID	LoC	Manually-written patch by developers	Applying B-Refactoring	#Test cases		Repair result
					Before	After	
-	137231	10.4K	`text == null || repl == null || with == null || repl.length() == 0`	Yes	1	2	Incorrect patch
1	137371	11.0K	`lastIdx <= 0`	Yes	1	3	Correct patch
2	137552	12.9K	`len < 0 || pos > str.length()`	Yes	1	3	Correct patch
-	230921	15.8K	`substr == null || startIndex >= size`	Yes	3	6	Correct patch
3	825420	17.4K	`className == null || className.length() == 0`	Yes	5	6	Incorrect patch
-	1075673	18.9K	`cs == null || cs.length() == 0`	Yes	1	2	No patch
Total					12	22	

```
1  String chopNewline(String str) {
2    int lastIdx = str.length() -1;
3
4    // PATCH: if (lastIdx <= 0) {
5    if (lastIdx == 0)
6      return "";
7    char last = str.charAt(lastIdx);
8    if (last == '\n')
9      if (str.charAt(lastIdx -1) == '\r')
10       lastIdx--;
11   else
12     lastIdx++;
13   return str.substring(0, lastIdx);
14 }
```

(a) Buggy program

```
1  void testChopNewLine(){
2    ...
3    assertEquals(FOO + "\n" + FOO,
4        StringUtils.chopNewline(FOO
5        + "\n" + FOO));
6
7  // B-Refactoring splits here
8    assertEquals(FOO + "b\n",
9        StringUtils.chopNewline(FOO
10       + "b\n\n"));
11
12 // B-Refactoring splits here
13   assertEquals("",
14       StringUtils.chopNewline("\n"));
15 }
```

(b) Test case

Fig. 3: Code snippets of a buggy program and a test case in Case study 1. The buggy `if` condition is at Line 5 of Fig. 3a; the test case in Fig. 3b executes both the `else` and `then` branches of the buggy `if`. Then B-Refactoring splits the test case into three test cases (at Lines 7 and 12 in Fig. 3b).

2) Case Study: We show how B-Refactoring influences the repair of the bug with ID 137371 as follows. Fig. 3 shows a code snippet with a buggy `if` condition at Line 5 of bug with ID 137371. In Fig. 3a, the method `chopNewLine` aims to remove the line break of a string. The original `if` condition missed the condition of `lastIdx < 0`. In Fig. 3b, a test case `testChopNewLine` targets this method. We show three test constituents, i.e., three assertions, in this test case (other test constituents are omitted for the sake of space). The first and third assertions cover the `else` branch (viewing as an empty `else` branch) of the `if` condition at Line 5 of `chopNewLine` while the second assertion covers the `then` branch. Such a test case confuses Nopol; that is, it cannot identify unambiguously the covered branch of this test case and cannot generate a patch for this bug.

B-Refactoring can split the test case into three test cases, as shown at Lines 7 and 12 in Fig. 3b. We replace the original test case with three new test cases after B-Refactoring. Then the repair approach can generate a patch, which behaves the same as the manual patch at Line 4 in Fig. 3a.

Results on the two other bugs (with ID 137552 and ID 825420) can be found in Section Appendix. B-Refactoring also enables Nopol to find the patch for the bug with ID 137552; Nopol cannot correctly fix the bug with ID 825420, even with the support of B-Refactoring. To further exploring the fixability, in addition to automatic refactoring, we manually add a test case that specifies a missing situation, which was an omission in the original design of the test suite.

Based on the three case studies, we show that improving the number of purely covered `if` elements can improve the number of fixed bugs by Nopol; meanwhile, prior results in Table V shows the number of purely covered `if` elements is improved by 66.34% by applying B-Refactoring to the original test suites. Hence, a number of potential bugs with impurely covered `if` elements could be newly fixed with B-Refactoring.

To sum up, we have shown that our approachB-Refactoring enables Nopol to automatically repair previously unfixed bugs with Nopol, by providing a refactored version of the test suite that produces optimized traces for the technique under consideration.

> Answer to RQ4: B-Refactoring improves the fixability of the Nopol program repair technique on real-world bugs, which cannot be fixed before applying B-Refactoring.

TABLE IX: B-Refactoring for exception contract checking

Project	Before			After			Improvement on #unknown	
	#Source-independent	#Source-dependent	#Unknown	#Source-independent	#Source-dependent	#Unknown	#	%
Lang	23	5	22	37	6	7	15	68.18%
Spojo-core	1	0	0	1	0	0	0	n/a
Jbehave-core	7	2	33	8	2	32	1	3.03%
Shindig-gadgets	30	12	38	31	13	36	2	5.26%
Codec	8	0	2	10	0	0	2	100.00%
Total	69	19	95	87	21	75	20	21.05%

B. B-Refactoring for Exception Contract Analysis

RQ5: Does B-Refactoring improve the identification of exception contracts by the contract verification technique, SCTA [8]?

In this section, we employ an existing dynamic analysis technique of exception contracts, called SCTA by Cornu et al. [8]. SCTA aims to verify an exception handling contract, called source-independence, which states that `catch` blocks should work in all cases when they catch an exception. Assertions in a test suite are used to verify correctness. The process of SCTA is as follows. To analyze exception contracts, exceptions are injected at the beginning of `try` elements to trigger the `catch` branches; then, a test suite is executed to record whether a `try` or `catch` branch is covered by each test case. *SCTA requires that test cases execute only the `try` or the `catch`.*

However, if both `try` and `catch` branches are executed by the same test case, SCTA cannot identify the coverage of the test case. In this case, the logical predicates behind the algorithm state that the contracts cannot be verified because the execution traces of test cases are not pure enough with respect to `try` elements. According to the terminology presented in this article, we call such test cases covering both branches *impure*. If all the test cases that execute a `try` element are impure, no test cases can be used for identifying the source-independence. To increase the number of identified `try` elements and decrease the number of unknown ones, we leverage B-Refactoring to refactor the original test cases into purer test cases.

1) Protocol: We apply B-Refactoring on the five projects in Section IV-A. We aim to identify the source-dependency of more exception contracts (`try` elements); that is, the number of unknown `try` elements on source-dependency should be reduced. Then the goal of this experiment is *to evaluate how many `try` elements are recovered from unknown ones*. We apply B-Refactoring to the test suite before analyzing the exception contracts. That is, we first refactor the test suite and then apply SCTA on the test suites before and after refactoring. Similar to the original work [8], we assess the number of unknown source-dependency of `try`s to evaluate B-Refactoring in exception contract analysis.

In exception contract analysis, a `try` is *source-independent* if the `catch` block proceeds equivalently, whatever the source of the caught exception is in the `try` block [8]. We analyze exception contracts with the following metrics:[14]

- #*Source-independent* is the number of verified source-independent `try` elements;
- #*Source-dependent* is the number of verified source-dependent `try` elements;
- #*Unknown* is the number of unknown `try` elements, because at least one test case is impure.

The last metric is the key one in this experiment. The goal is to decrease this metric by refactoring, i.e., to obtain less try-catch blocks, whose execution traces are too impure to apply the verification algorithm.

2) Results: We investigate the results of the exception contract analysis before and after B-Refactoring.

Table IX presents the number of source-independent `try` elements, the number of source-dependent `try` elements, and the number of unknown ones. Taking the project Lang as an example, the number of unknown `try` elements decreases by 15 (from 22 to 7). This enables the analysis to prove the source-independence for 14 more `try` (from 23 to 37) and to prove source-dependence for one more (from 5 to 6). That is, by applying B-Refactoring to the test suite in project Lang, we can detect whether these 68.18% (15/23) `try` elements are source-independent or not.

For all the five projects, 21.05% (20 out of 95) of `try` elements are rescued from unknown ones. This result shows that B-Refactoring can refactor test suites to cover simple branches of `try` elements. Such refactoring helps the dynamic analysis to identify the source independence.

Answer to RQ5: Applying B-Refactoring to test suites improves the ability of verifying the exception contracts of SCTA. 21% of unknown exception contracts are reduced.

VI. Discussions

In this section, we present our discussion on differences between B-Refactoring and classical refactoring, differences between B-Refactoring and our previous work on test case purification, performance of B-Refactoring, and threats to the validity in our work.

A. Differences with Classical Refactoring

As mentioned in Section III-A, B-Refactoring aims to enhance dynamic analysis techniques. This leads to several

[14]Note that the sum of the three metrics is constant before and after applying B-Refactoring.

differences between our work and existing refactoring techniques, including program refactoring and test code refactoring [29] as follows.

First, the goal is different. B-Refactoring benefits dynamic analysis techniques while classical refactoring benefits developers. We design B-Refactoring to improve dynamic analysis (e.g., automatic repair); that is, the users of B-Refactoring are programs and not humans. In contrast, most existing refactoring techniques aim to help human developers [22] and the users of classical refactoring are developers.

Second, a test suite after applying B-Refactoring temporarily replaces the original test suite only to conduct the task of dynamic analysis. On the contrary, the code resulting from classical refactoring is meant to replace the original code.

Third, a refactored test suite by B-Refactoring does not have any readability requirements because no developer will ever read this refactored test suite. Renaming, a common concern of classical refactoring, such as the difficulty of finding "good" names [42], is not applicable to B-Refactoring. For instance, a variable can be renamed as `a_b_c` without the concern of test readability.

B. Differences with Our Previous Work

In our previous work, test case purification [54], we split failing test cases to assist fault localization. Four major differences between B-Refactoring in this article and our previous work [54] are detailed as follows.

First, test case purification is designed to improve the performance of a single technique, fault localization, while B-Refactoring is more general in the sense that it can improve multiple tasks.

Second, in test case purification, only failing test cases are refactored: skipped assertions in failing test cases are extracted as new test cases. In B-Refactoring, both passing and failing test cases are refactored according to traces during test execution.

Third, the core criterion for splitting test cases is different. B-Refactoring splits test cases based on the execution monitored in application code, which is executed by test cases, e.g., `then` and `else` branches based on the execution of `if` elements. In test case purification, the failing test cases are split based on assertions in test code (and not in application code); that is, the refactoring criterion focuses on test code and not application code.

Fourth, test case purification uses dynamic slicing, while in B-Refactoring, no slicing technique is used because we do not need to remove any statement.

C. Performance of B-Refactoring

We discuss the time cost of performing B-Refactoring as follows. In general, applying B-Refactoring comprises four phases. As shown in Section III-B, first, given a specific task of dynamic analysis, a project as well as its test suite under refactoring is instrumented so as to collect the runtime trace between each test constituent and each program element. Second, the test suite is run to collect the trace based on the instrumentation. Third, according to the collected trace, the original test suite is transformed into a new test suite with smaller test cases. Fourth, the new test suite after refactoring is run as a test suite in the task of dynamic analysis (e.g., automatic repair in Section V-A).

The first and the third phases last for a few seconds, always less than one minute for our dataset on a standard laptop. The time cost of the second and the fourth phases depend on the project under study. For a mid-sized project, such as Lang in our dataset, a typical execution of a test suite lasts within minutes. The time cost of the second phase is nearly the same as the test execution; the time cost of the fourth phase is nearly the same as the time cost of executing the test suite in a specific task of dynamic analysis.

This is preliminary evidence that B-Refactoring is fast. The time cost is a negligible part in the dynamic analysis that is performed. Note that for a given task, B-Refactoring is used offline, without strong requirements on the execution time.

D. Threats to Validity

We present threats to the validity of our B-Refactoring results in three categories.

Construct validity. In this article, we use three case studies on real-world bugs to demonstrate the benefit of applying B-Refactoring to test suites. Improvement based on these case studies indicates that our proposed technique can help to better use test suites for automatic repair. Collecting more real bugs for evaluating B-Refactoring is possible, but time-consuming [52]. We leave the evaluation on more real bugs as future work. In addition, results of test case purity in five studied projects are diverse. In Sections IV-B and IV-C, we show detailed analysis of these results. However, a further exploration on test suites should be conducted to find out what these results are caused by. Due to the lack of historical test data, we have not investigated the reason of the diverse results, which is left as future work. In Section IV-D, we apply the idea of mutation testing to evaluate the refactored test suite. We validate that the test suite after refactoring does not lose the ability of testing via randomly selected program mutants. However, we note that our validation of test suites is incomplete. That is, not all program behaviors are checked by mutants and the mutation space is not completely explored (for the sake of time).

Internal validity. Test code can be complex. For example, a test case can have loops and internal mocking classes. In our implementation, we consider test constituents as top-level statements, thus complex test constituents are simplified as atomic ones. Handling more complex constituents can reduce the number of impure test cases, e.g., reducing the number of absolutely impure test cases in Table III. But the complex structure of test code will increase the effort of implementing our technique. Hence, by skipping the process of internal statements inside test code, our work could be viewed as a trade-off between refactoring all test constituents and the implementation effort.

External validity. We have shown that B-Refactoring improves the efficiency of program repair and contract verifi-

cation. However, the two considered approaches stem from our own research. This is natural, since our expertise in dynamic analysis makes us aware of the nature of these problems. For further assessing the generic nature of our refactoring approach, experiments involving other dynamic analysis techniques are required. Moreover, besides Nopol [52] and SCTA [8], there exist other methods in both automatic repair and exception contract analysis. We have not applied B-Refactoring to these methods since most of these tools are not open available and are not implemented in Java as B-Refactoring. Moreover, experiments in our work mainly focus on `if` and `try` program elements. Both of these program elements can be viewed as a kind of branch statements. Our proposed work can also be applied to other elements, like method invocations (in Section III-A). To show more evidence of the improvement on dynamic analysis, experiments could be conducted for different program elements.

VII. Related Work

We list the related work to our article in three categories: the approach to test code refactoring and two application scenarios.

A. Test Code Refactoring

Test code refactoring [49] is a general concept of making test code better understandable, readable, or maintainable. Based on 11 test code smells, Deursen et al. [49] first propose the concept of test code refactoring as well as six refactoring rules, including reducing test case dependence and adding exploration for assertions. Extension on this work by Van Deursen & Moonen [48] and Pipka [36] propose how to refactor test code for the *test first* rule in extreme programming. Guerra & Fernandes [16] define a set of representation rules for different categories of test code refactoring. Moreover, Xu et al. [51] propose directed test suite augmentation methods to detect affected code by code changes and to generate test cases for covering these code. Recent work by Xuan et al. [55] proposes test case mutation to reproduce crashes with stack trace. Test case mutation generates multiple variants for a given test case to trigger potential bugs.

Technical issues in refactoring have been studied. Schäfer et al. [42] focus on conducting better renaming systems in Java Integrated Development Environment (IDE) while their follow-up work [43] addresses another technique, how to extract methods during refactoring. Overbey et al. [34] propose a method of differential precondition checking, which identifies the behavior preservation in refactoring. To evaluate refactoring techniques, Soares et al. [46] automatically generate programs as well as test suites to detect bugs of refactoring engines; Gligoric et al. [15] propose an end-to-end approach to testing refactoring engines with real-world projects and evaluate their reliability. In this article, we have not employed the above testing techniques to validate our refactoring approach; instead, we leverage the technique of program mutation to evaluate our approach. In our work, program mutants are generated and executed with test suites before and after refactoring; experiments show that program mutants that killed by test suites are the same with and without refactoring.

Refactoring techniques in source code [29] have been introduced to test code. Existing known patterns in refactoring are applied to test cases to achieve better-designed test code. Chu et al. [7] propose a pattern-based approach to refactoring test code to keep the correctness of test code and to remove the bad code smells. Alves et al. [2] employ pattern-based refactoring on test code to make better regression testing via test case selection and prioritization. In contrast to modifying one test via the above pattern-based refactoring on test code, our work in this article aims to split one test case into a set of small and pure test cases. The new test cases can assist a specific software task, e.g., splitting test cases to execute single branches in `if` elements for software repair and to trigger a specific status of `try` elements for exception handling.

Two concepts in test generation relates to our work, i.e., test case adaptation and test case optimization. *Test case adaptation*, introduced by Mirzaaghaei et al. [31] aims to repair test cases to suit for the code change. On the other hand, *test case optimization*, introduced by Baudry et al. [4], [5], generates new test cases to improve the ability of fault localization. Hence, both test case adaptation and test case optimization generate new test cases. However, in our work, we aim to improve the usage of test cases for dynamic analysis techniques.

B. Automatic Software Repair

Automatic software repair aims to generate patches to fix software bugs. Software repair employs a given set of test cases to rank potential faulty statements [1], [53] and validate the correctness of generated patches. Weimer et al. [23] propose GenProg, a genetic-programming based approach to fixing C bugs. This approach views a fraction of source code as an AST and updates ASTs by inserting and replacing known AST nodes. Nguyen et al. [33] propose SemFix, a semantic-analysis based approach, also for C bugs. This approach combines symbolic execution, constraint solving, and program synthesis to narrow down the search space of repair expressions.

Martinez & Monperrus [26] mine historical repair actions based on fine-granulated ASTs with a probabilistic model. Kim et al. [21] propose Par, a pattern-based repair approach via frequent ways of fixing bugs. The repair patterns in their work are used to avoid nonsensical patches due to the randomness of some mutation operators. Qi et al. [37] investigate the strength of random search in GenProg and show that the random search (without genetic programming) based repair method, RSRepair, can achieve even better performance than GenProg. Kaleeswaran et al. [20] propose MintHint, a repair hint method for identifying expressions that are likely to occur in patches, instead of fully automated generating patches. Mechtaev et al. [28] address the simplicity of generated patches with a maximum satisfiability solver. Long & Rinard [24] propose

a staged program repair method for synthesizing conditional bugs.

Barr et al. [3] investigate the "plastic surgery" hypothesis in genetic programming based repair like GenProg and show that patches can be constructed via reusing existing code. Martinez et al. [27] target the redundancy assumptions for existing code. Tao et al. [47] explore how to leverage machine-generated patches to assist human debugging. Monperrus [32] discusses the problem statement and the evaluation criteria of software repair. Zhong & Su [57] examine 9,000 real-world patches and summarize 15 findings for fault localization and faulty code fix in automatic repair. A recent study by Qi et al. [38] shows that only 2 out of 55 generated patches by GenProg and 2 out of generated patches by RSRepair are correct; all the others fail to be expected behaviors due to experimental issues and weak test cases. Martinez et al. [25] empirically evaluate the results of GenProg [23], Kali [38], and Nopol [52] on 224 real bugs from the Defects4J dataset. This study indicates the difficulty of repairing real bugs and the status of primarily weak specifications in test cases.

In existing work [9], [52], we propose Nopol, a specific repair tool targeting buggy `if` conditions. In this article, we leverage Nopol as a tool in one application scenario of automatic software repair, which investigates real-world bugs on `if`.

C. Automatic Analysis of Exception Handling

Exception handling aims to analyze and to enhance the processing of software exceptions. Sinha & Harrold [44] propose representation techniques with explicit exception occurrences (explicitly via `throw` statements) and exception handling constructs. Their following work by Sinha et al. [45] develops a static and dynamic approach to analyzing implicit control flows caused by exception handling.

Robillard & Murphy [39] present the concept of exception-flow information and design a tool that supports the extraction and view of exception flows. Fu & Ryder [14] develop a static exception-chain analysis for the entire exception propagation in programs. Zhang & Elbaum [56] study the faults associated with exceptions that handle noisy resources and propose an approach to amplifying the space of exceptional behavior with external resources. Moreover, Bond et al. [6] present an efficient origin tracking technique for null and undefined value errors in the Java virtual machine and a memory-checking tool. Mercadal et al. [30] propose an approach that relies on an architecture description language, which is extended with error-handling declarations.

In existing work [8], we propose an approach to detect the types of exception handling on nine Java projects. In this article, the approach [8] serves as a platform to examine whether B-Refactoring can improve the ability of detecting exception contracts.

VIII. Conclusions

This article addresses the problem of impure traces of test cases We propose B-Refactoring, a technique to split test cases into small fragments in order to increase the efficiency of dynamic program analysis. Our experiments on five open-source projects show that our approach effectively improves the purity of test cases. We show that applying B-Refactoring enhances the performance of existing analysis tasks, namely repairing `if`-condition bugs and analyzing exception contracts.

In future work, we plan to apply B-Refactoring to other kinds of program analysis, such as test case prioritization. Moreover, we will explore the reason of designing impure test cases by analyzing and understanding existing tests in open-source projects. A human study on testers who design impure test cases could be helpful to further understand the hidden knowledge of test design. We also plan to extend our implementation of B-Refactoring to deal with more complex statements (e.g., loops) in test cases. An advanced tool that handles fine-grained test statements can effectively reduce the impurity of test suites.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China (under grant 61502345).

References

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[2] E. L. Alves, P. D. Machado, T. Massoni, and S. T. Santos. A refactoring-based approach for test case selection and prioritization. In *2013 8th International Workshop on Automation of Software Test*, pages 93–99. IEEE, 2013.

[3] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317, 2014.

[4] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test case optimization using a bacteriological adaptation model: application to. net components. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 253–256. IEEE, 2002.

[5] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, pages 82–91. ACM, 2006.

[6] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. *ACM SIGPLAN Notices*, 42(10):405–422, 2007.

[7] P.-H. Chu, N.-L. Hsueh, H.-H. Chen, and C.-H. Liu. A test case refactoring approach for pattern-based software development. *Software Quality Journal*, 20(1):43–75, 2012.

[8] B. Cornu, L. Seinturier, and M. Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 57:66–76, 2015.

[9] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[11] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 210–219. IEEE, 1999.

[12] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

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

[14] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 230–239. IEEE, 2007.

[15] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 629–653, 2013.

[16] E. M. Guerra and C. T. Fernandes. Refactoring test code safely. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 44–44. IEEE, 2007.

[17] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[18] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[19] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[20] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276. ACM, 2014.

[21] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[22] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[24] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178, 2015.

[25] M. Martinez, T. Durieux, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *arXiv preprint arXiv:1505.07002*, 2015.

[26] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[27] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proceedings of the 36th International Conference on Software Engineering*, pages 492–495, 2014.

[28] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

[29] T. Mens and T. Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.

[30] J. Mercadal, Q. Enard, C. Consel, and N. Loriant. A domain-specific approach to architecturing error handling in pervasive computing. *ACM Sigplan Notices*, 45(10):47–61, 2010.

[31] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 231–240. IEEE, 2012.

[32] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[33] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[34] J. L. Overbey, R. E. Johnson, and M. Hafiz. Differential precondition checking: a language-independent, reusable analysis for refactoring engines. *Automated Software Engineering*, pages 1–28, 2014.

[35] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, page na, 2015.

[36] J. U. Pipka. Refactoring in a "test first"-world. In *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng*, 2002.

[37] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[38] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.

[39] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[40] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[41] M. Schäfer and O. De Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM.

[42] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 277–294, 2008.

[43] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping stones over the refactoring rubicon. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 369–393, 2009.

[44] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *Software Engineering, IEEE Transactions on*, 26(9):849–871, 2000.

[45] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 336–345. IEEE, 2004.

[46] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Trans. Software Eng.*, 39(2):147–162, 2013.

[47] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[48] A. Van Deursen and L. Moonen. The video store revisited – thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002.

[49] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[50] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[51] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266. ACM, 2010.

[52] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, to appear.

[53] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200. IEEE, 2014.
[54] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63. ACM, 2014.
[55] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 910–913, 2015.
[56] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, pages 595–605. IEEE Press, 2012.
[57] H. Zhong and Z. Su. An empirical study on fixing real bugs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 2015.

APPENDIX

CASE STUDIES ON REPAIRING REAL-WORLD BUGS

We evaluate B-Refactoring on three real-world bugs in Apache Commons Lang. Detailed description on these bugs can be found in Table VIII. The first case study is in Section V-A and the other two case studies are as follows.

A. Case study 2

The code snippet in Fig. 4 presents an `if`-condition bug with ID 137552 in Apache Commons Lang. In Fig. 4a, the method `mid` is to extract a fixed-length substring from a given position. The original `if` condition at Line 7 did not deal with the condition of `len < 0`, which is expected to return an empty string. In Fig. 4b, a test case `testMid_String` targets this method. Three assertions are shown to explain the coverage of branches. Two assertions at Lines 3 and 11 cover the `else` branch of the `if` condition while the other assertion at Line 7 covers the `then` branch. A repair approach, like Nopol, cannot generate a patch for this bug because the test case `testMid_String` covers both branches of the `if` condition at Line 7 in the method `mid`.

We apply B-Refactoring to split the test case into three test cases, as shown at Lines 6 and 10 in Fig. 4b. Such splitting can separate the coverage of `then` and `else` branches; that is, each new test case only covers either a `then` or `else` branch. Then the repair approach can generate a patch, `len <= -1 && str.length() < pos`, which behaves the same to the manual patch at Line 6 in Fig. 4a.

B. Case study 3

This bug is with ID 825420 in Apache Commons Lang. Fig. 5 shows a code snippet with a buggy `if` condition at Line 12. In Fig. 5a, two methods `getPackageName(Class)` and `getPackageName(String)` work on extracting the package name of a class or a string. The original `if` condition missed checking the empty string, i.e., the condition of `className.length() == 0`. In Fig. 5b, we show two test cases that examine the behavior of these two methods. For the first test case `test_getPackageName_Class`, we present two assertions. We do not refactor this test case because this test case is pure (the first assertion executes the `else` branch while the second assertion does not execute any branch). For the second test case `test_getPackageName_String`, two assertions are shown. The first one is passing while the second is failing. Thus, we split this test case into two test cases to distinguish passing and failing test cases. Such splitting is specifically designed to support the repair by Nopol.

Based on B-Refactoring, we obtain two test cases, as shown at Line 16 in Fig. 5b. Then the repair approach can generate a patch as `className.length() == 0`. Note that this patch is different from the real patch because the condition `className == null` is ignored. The reason is that in the original test suite, there exists no test case that validates the `then` branch at Line 13. That is, the patch is incorrect.

To generate the same patch as the real one at Line 9, we manually add one test case `test_manually_add` at Line 23 in Fig. 5b. This test case ensures the behavior of the condition `className == null`. Based on this manually added test case and the test cases by B-Refactoring, the repair approach can generate a patch that is the same as the real one.

Summary. In summary, we empirically evaluate our B-Refactoring technique on three real-world `if`-condition bugs from Apache Commons Lang. All these three bugs cannot be originally repaired by the repair approach, Nopol. The reason is that one test case covers both the `then` and `else` branches. Then Nopol cannot decide which branch is covered and cannot generate repair constraints for this test case. With B-Refactoring, we separate test cases into pure test cases to cover only a `then` or `else` branch. Based on the test cases after applying B-Refactoring, the first two bugs are correctly fixed while the third bug is not. For the third bug, one test case is ignored by developers in the original test suite. By manually adding a new test case, this bug can also be fixed via the test suite after B-Refactoring.

```
1  String mid(String str, int pos, int len) {
2    if (str == null)
3      return null;
4
5    // PATCH:
6    // if (len < 0 || pos > str.length())
7    if (pos > str.length())
8      return "";
9
10   if (pos < 0)
11     pos = 0;
12   if (str.length() <= (pos + len))
13     return str.substring(pos);
14   else
15     return str.substring(pos, pos + len);
16 }
```

(a) Buggy program

```
1  void testMid_String() {
2    ...
3    assertEquals("b", StringUtils
4          .mid(FOOBAR, 3, 1));
5
6  // B-Refactoring splits here
7    assertEquals("", StringUtils
8          .mid(FOOBAR, 9, 3));
9
10 // B-Refactoring splits here
11   assertEquals(FOO, StringUtils
12         .mid(FOOBAR, -1, 3));
13 }
```

(b) Test case

Fig. 4: Code snippets of a buggy program and a test case in Case study 2. The buggy `if` statement is at Line 7 in Fig. 4a while the test case in Fig. 4b executes the `else`, the `then`, and the `else` branches of the buggy statement, respectively. Then B-Refactoring splits the test case into three test cases.

```
1  String getPackageName(Class cls) {
2    if (cls == null)
3      return StringUtils.EMPTY;
4    return getPackageName(cls.getName());
5  }
6
7  String getPackageName(String className){
8
9    // PATCH: if (className == null
10         || className.length() == 0)
11
12   if (className == null)
13     return StringUtils.EMPTY;
14   while (className.charAt(0) == '[')
15     className = className.substring(1);
16   if (className.charAt(0) == 'L' &&
17         className.charAt(className
18         .length() -1) == ';')
19     className = className.substring(1);
20   int i = className.lastIndexOf(
21         PACKAGE_SEPARATOR_CHAR);
22   if (i == -1)
23     return StringUtils.EMPTY;
24   return className.substring(0, i);
25 }
```

(a) Buggy program

```
1  // Do not need to be split.
2  void test_getPackageName_Class() {
3    assertEquals("java.util", ClassUtils
4          .getPackageName(Map.Entry.class));
5    assertEquals("", ClassUtils
6          .getPackageName((Class)null));
7    ...
8  }
9
10 void test_getPackageName_String() {
11   ...
12   assertEquals("java.util", ClassUtils
13         .getPackageName(
14         Map.Entry.class.getName()));
15
16 // B-Refactoring splits here
17   assertEquals("", ClassUtils
18         .getPackageName(""));
19 }
20
21 // Manually added test case
22 // to generate a correct condition
23 void test_manually_add() {
24   assertEquals("", ClassUtils
25         .getPackageName((String)null));
26 }
```

(b) Test cases

Fig. 5: Code snippets of a buggy program and a test case in Case study 3. The buggy `if` statement is Line 12 of Fig. 5a while two test case in Fig. 5b executes `then` and `else` branches of the buggy statement. B-Refactoring splits the second test case into two test cases and keeps the first test case. The last test case `test_manually_add` is manually added for explanation.