Studying the Impact of Early Test Termination Due to Assertion Failure on Code Coverage and Spectrum-based Fault Localization

by

Md. Ashraf Uddin

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
June 2023

Thesis advisor

Shaowei Wang

Author

Md. Ashraf Uddin

Studying the Impact of Early Test Termination Due to Assertion Failure on Code Coverage and Spectrum-based Fault Localization

Abstract

An assertion is commonly used to validate the expected program's behavior (e.g., if the returned value of a method equals an expected value) in software testing. Although it is a recommended practice to use only one assertion in a single test to avoid code smell (e.g., Assertion Roulette), we observe that it is common to have multiple assertions in a single test. One issue with tests that have multiple assertions is that when the test fails at an early assertion, the test will terminate at that point, and the remaining testing code will not be executed. This, in turn, can potentially reduce the code coverage and the performance of techniques that rely on code coverage information (e.g., spectrum-based fault localization). We refer to such a scenario as *early test termination*. Understanding the impact of *early test termination* on test coverage is important for software testing and debugging, particularly for the techniques that rely on coverage information obtained from the testing. In this study, we investigated 207 versions of 6 open-source projects. We found that a non-negligible portion of the failed tests (19.1%) is early terminated due to assertion failure, which leads to the skipping of 15.3% to 60.5% of the test code on average, and a negative impact on testing coverage. To mitigate *early test termination*, we propose two approaches, i.e., *Trycatch* (adding a try-catch block surrounding an assertion) and *Slicing* (slicing a test into a set of independent sub-tests, in which only one assertion and its dependent code are contained). After applying our approaches, the line/branch coverage get improved in 55% of the studied versions. Moreover, *Slicing* improves the performance of SBFL by 15.1% and 10.66% in terms of Mean First Rank (MFR) for Ochiai and

Tarantula, respectively. We also provide actionable suggestions to prevent *early test termination*, and approaches to mitigate *early test termination* if it already exists in their project.

Contents

List of Figures

List of Tables

Dedicated to my parents and supervisor for their supports and encouragement...

Chapter 1

Introduction

Testing is an effective and practical way of finding bugs and improving software quality. However, due to the scale of modern software and the complexity of tests, it can be difficult for developers to know the cause of test failures and formulate effective solutions. Prior studies have proposed various automated testing techniques that leverage code coverage and test execution information to assist developers with test failure diagnosis or repair. For example, researchers have proposed using code coverage information in failed/passed tests for tasks such as spectrum-based fault localization (SBFL) [1–8] and automated program repair [9–13]. For instance, Ochiai [14], and Tarantula [15] are two most popular SBFL techniques, leverage both passed/failed tests together with their coverage information to identify the faulty location in the program.

In testing, assertions are commonly used to verify the program's expected behavior (e.g., if the returned value of a method equals an expected value). Although it is a recommended practice to use only one assertion in a test to avoid code smell (e.g., Assertion Roulette [16]), prior studies [17, 18] found that many tests may still contain multiple assertions. An inherent concern associated with tests encompassing multiple assertions is that in the event of a failure occurring at an initial assertion, the test early halts its execution. This early termination subsequently leads to the omission of the

remaining test code. As a consequence, there exists the potential for a reduction in code coverage, occasionally resulting in the failure to expose the buggy line of code (see more details in Subsection 2.3). Furthermore, this phenomenon also impairs the efficacy of Spectrum-Based Fault Localization (SBFL) and automated program repair techniques. We refer to the scenario where a test has multiple assertions and fails at an early assertion (i.e., not the last assertion) as *early test termination*. Understanding the negative impact of *early test termination* on test coverage holds significant importance within the domains of software testing and debugging. This significance is particularly pronounced when considering techniques that depend on the coverage data derived from testing, serving as a cornerstone for their analytical and corrective procedures.

Therefore, in this study, we investigated 207 versions collected from six open-source projects used in Defect4J [19] and T-Evos [1]. We aim to understand the prevalence of *early test termination*, and examine its impact on the test coverage and effectiveness of SBFL techniques that rely on coverage information. We formulate the following research questions in our research:

- **RQ1: How prevalent is early test termination?**

  We investigate how prevalent *early test termination* is in failed tests and the amount of test code that is skipped due to *early test termination*.

  Results: Among the failed tests, a non-negligible portion (19.1%) is early terminated due to assertion failure and they occur in 29% of studied versions. Due to *early test termination*, on average 15.3% to 60.5% of the test code is skipped.

- **RQ2: How does early test termination affect test coverage?**

  We propose two approaches *Trycatch* and *Slicing* to enforce the continuous execution of tests even if *early test termination* happens. In *Trycatch*, we modify a test by adding a try-catch block surrounding each assertion in the test to enforce it to keep executing. In *Slicing*, we use program slicing to disassemble a test with multiple assertions into multiple independent sub-tests, which

contain only one assertion and its dependent code. We refer to the original setting as *Original*. We compare the testing coverage before (*Original*) and after applying *Trycatch* and *Slicing*.

Results: In general, *early test termination* negatively impacts the coverage in the production code in terms of all granularities (i.e., instruction, line, branch, and method). *Trycatch* and *Slicing* improve the line/branch coverage in 55% of the studied versions compared to *Original*.

- **RQ3: How does early test termination affect the performance of spectrum-based fault localization?**

  We select Ochiai and Tarantula as the representative of SBFL techniques. We compare the performance of both Ochiai and Tarantula in three configurations: *Original*, *Trycatch*, and *Slicing*.

  Results: *Slicing* and *Trycatch* improve the performance of SBFL by 15.1% and 6.7% on average in terms of Mean First Rank (MFR), respectively, compared to *Original*. *Slicing* improves the performance of SBFL in 30% of the studied versions, while *Trycatch* only improves in 18.3% of the versions. Overall, *Slicing* outperforms *Trycatch* for SBFL.

We summarize our contribution as follows:

- We conducted the first study on early test termination due to assertion failure (i.e., *early test termination*). We provided evidence of the prevalence of *early test termination* and its negative impact on testing coverage.
- We proposed two approaches (*Trycatch* and *Slicing*) to mitigate the impact of early test failures on test coverage.
- We investigated the effectiveness of the two proposed approaches in improving the performance of SBFL techniques (i.e., Ochiai and Tarantula) and demonstrated that *Slicing* performs the best.

- We provide actionable suggestions to prevent *early test termination*, and approaches to mitigate *early test termination* if it already exists in their project.

1.1 Organization of this Thesis

This thesis is organized as follows:

- Chapter 2 provides background knowledge, which is utilized in this thesis.
- Chapter 3 presents related work.
- Chapter 4 discusses experimental design.
- Chapter 5 presents the RQ results.
- Chapter 6 provides a discussion and threats to validity.
- Chapter 7 concludes the thesis.

Chapter 2

Background

In this chapter, we first discuss the phenomenon of early test termination and its potential impact on SBFL. Then, we provide a motivating example to illustrate the impact of early test termination.

2.1 Early Test Termination

Software testing is an effective technique to find bugs in a system. When a test fails, developers can then investigate the failing code statements to identify the potential cause. To assist developers with diagnosing the causes of test failures, researchers have proposed various techniques such as SBFL [1–4, 6–8]. However, when a test terminates prematurely, the generated test coverage and test failure messages become incomplete, which affects how developers and techniques such as SBFL locate the root cause of failures.

We first formally define early test termination. Suppose there is a test t that has n statements $\{s_1, s_2, ..., s_i, ..., s_n\}$. We define t as an early terminated test if t terminates at a statement s_i, where $i < n$. A test may terminate early due to various reasons, such as errors in object initialization, method invocation, or assertion. In this thesis, we focus the study on early test termination due to

assertion failure. Although having multiple assertions in a test can be considered a test smell [16], we observed that this is a common practice in the studied systems (as shown in Table 4.1, 34.6% to 73.3% of the tests have multiple assertions in our studied projects). We also observe that early test termination due to assertion failure happens frequently (see more details in Chapter 5). In JUnit, a test will terminate immediately at the point where an assertion fails and will not execute the code after that point and report the result of the test as failed immediately. For instance, in Figure 2.1, the test testRootEndpoints has multiple assertions and it stops at the third assertion and the rest of the code does not get executed. As a result, the remaining test code after the failed assertion would not be executed, and potentially reduce the coverage on the production code. Our goal is to study the impact of early test termination due to assertion failure on test coverage and the performance of SBFL. For simplicity, we refer to early test failure due to assertion failure as *early test termination* and the test where *early test termination* happens as $Test_{earlyTerminated}$ in the rest of the paper.

2.2 Spectrum-based Fault Localization

In this work, we focus on Spectrum-Based Fault Localization (SBFL) as the subject technique to understand the impact of *early test termination*. We choose SBFL because it relies directly on the quality of the collected code coverage. SBFL uses statistical formulas to measure the suspiciousness of program units based on the program execution traces. Execution traces, also called program spectra, contain details of the covered statements in failed and passed tests. The suspiciousness scores computed from program spectra are then used to generate a ranked list of program units that are most likely to be responsible for the fault. A number of SBFL techniques have been developed over the decades, such as Tarantula [15], Ochiai [14], DStar [8], and BARINEL [6]. For instance, Ochiai and Tarantula are commonly used and popular SBFL techniques. Ochiai is defined as follows:

```
1  public void testRootEndpoints() throws Exception {
2      UnivariateRealFunction f = new SinFunction();
3      UnivariateRealSolver solver = new BrentSolver();
4      // endpoint is root
5      double result = solver.solve(f, Math.PI, 4);
6      assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
7
8      result = solver.solve(f, 3, Math.PI);
9      assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
10
11     result = solver.solve(f, Math.PI, 4, 3.5);
12     // assertion failure, terminate here
13     -> assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
14
15     result = solver.solve(f, 3, Math.PI, 3.07);
16     // assertion failure
17     assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
18 }
19
```

Figure 2.1: A motivating example of early test termination occurs when an assertion fails on line 13, causing all the remaining tests and assertions to be skipped.

$$Ochiai, Suspicious\_score(s) = \frac{e_f(s)}{\sqrt{(e_f(s) + n_f(s))(e_f(s) + e_p(s))}},$$

where $e_p(s)$ and $e_f(s)$ are the numbers of passed tests and failed tests that execute statement s, respectively. $n_f(s)$ is the number of failed tests that do not execute s.

As shown in the definition of Ochiai, the suspiciousness score of a statement is based on the execution of passed and failed tests. Therefore, the performance of SBFL is potentially impacted by *early test termination*. For instance, suppose a test has two assertions a_1 and a_2 to test two branches b_1 and b_2 in the production code, respectively. If the test fails at a_1, then only b_1 is covered. If no

other tests cover b_2, the Ochiai score for all the statements in b_2 will be 0. However, a_2 may also fail if the test were able to continue executing, and we would miss the fault in b_2 because of *early test termination*.

According to the formula of Tarantula:

$$Tarantula, Suspicious\_score(s) = \frac{\frac{e_f(s)}{(e_f(s)+n_f(s))}}{\frac{e_f(s)}{(e_f(s)+n_f(s))} + \frac{e_p(s)}{(e_p(s)+n_p(s))}},$$

where $e_p(s)$ and $e_f(s)$ are the numbers of passed tests and failed tests that execute statement s, respectively. $n_f(s)$ and $n_p(s)$ are the numbers of failed tests and passed tests that do not execute s. From the definition of Tarantula, the suspiciousness score of a statement is based on both execution and not the execution of passed and failed tests.

2.3 A Motivating Example

Prior studies have leveraged code coverage to identify faulty locations, and code coverage is critical for the effectiveness of prior techniques [1–3, 5–8]. However, *early test termination* can potentially reduce the code coverage and therefore negatively impact the performance of the techniques rely on it. Let us illustrate our insight through a concrete example shown in Figure 2.1, which is a test adapted from the Apache Commons Math library[1], to test the root-finding functionality of the $BrentSolver$ class. It evaluates the first assertion statement at line 6. However, it aborts at line 13 since the third assertion fails, and the remaining code has not been executed. The aborted execution omits the unexecuted assertions (i.e., assertions at line 17) that are in the same test.

Nevertheless, the remaining code contains additional information that may be useful for fault localization. First, a bug's root cause can be directly reflected in the code coverage. Identifying additional coverage can help differentiate suspicious program elements, which can help locate the

[1] https://commons.apache.org/proper/commons-math/

root cause (i.e., buggy statements). If the remaining test code were executed in this example, one additional branch would be covered, providing potentially valuable information for diagnosing the buggy statements. Moreover, executing additional assertions can also improve fault localization accuracy, since assertions verify the correctness of the program behavior [20]. However, the failure of earlier assertions prevents the execution of later assertions. If the remaining assertions could be executed in Figure 2.1 and more assertion failures are identified, we would be able to identify additional incorrect program behavior that may help in diagnosing bugs without re-executing the tests. Therefore, in this study, we investigate the impact of reduced coverage on the effectiveness of coverage-based fault localization techniques (i.e., SBFL).

Chapter 3

RELATED WORK

3.1 Assertion Related Studies

Understanding and improving the usage of assertion in software testing. A few studies have been conducted to understand assert practice in testing [17, 21–23]. For instance, Bavota et al. performed an exploratory analysis of 18 systems to analyze the distribution of test smells and show that Assertion Roulette (i.e., a code smell occurs when a test method has multiple non-documented assertions) is presented in 62% of the total JUnit classes [17]. Although we did not examine if the assertion is documented or not, we also observed that a test containing multiple assertions is not rare. Bai et al. conducted an empirical study with 42 CS students to assess the impact of Assertion Roulette on programming quality measures and found that Assertion Roulette is no longer a smell for less experienced populations [21]. Zamprogno et al. examined more than 30k assertions from 105 projects and revealed that assertions fall into twelve high-level categories (e.g., Equality, Boolean, and Existence), and the vast majority more than 90% of the assertions are simple [22]. They also found 17.4% of the test cases have three or more assertions, which is compatible with our finding in RQ1. Jia et al. investigated the quality of existing test cases in deep learning libraries, and they found that developer-written test-case assertions need to be strengthened to better

detect faults [23]. On other hand, various solutions were proposed to improve the assert practice in unit testing [24–28]. For instance, Fraser and Arcuri proposed EvoSuite, that uses mutation testing to produce a set of assertions that maximizes the number of seeded defects in a class that can be revealed by the test cases. Watson et al. employed a Neural Machine Translation (NMT) based approach called Atlas to automatically generate meaningful assert statements providing a test method and a focal method to test the correctness of the focal method [24]. Similarly, to address Atlas's limitation from exposure bias and disappearance of gradient, Yu et al. propose an integrated approach to combine an information retrieval-based approach with a DL-based approach (e.g., ATLAS) [26].

3.2 Exception Handling (Trycatch) Related Works

Trycatch block is used for dealing with exceptional conditions and recovering from various exceptions [29, 30]. In this paper [31], researchers have proposed a neural approach to automated exception handling, which can predict the locations of try blocks and automatically generate the complete catch blocks. A deep learning-based technique is used to identify the types of runtime exceptions and the statement that detected runtime exceptions [32]. The objective of this paper is to enrich the ongoing discourse by offering quantitative metrics regarding the contemporary utilization of exception handling by programmers [33]. Here researchers have conducted an analysis of 32 distinct applications, encompassing both Java and .NET environments.

3.3 Works on Slicing

A plethora of scholarly works have been documented [34–36] since the initial proposition of static slicing by Weiser in 1979 [37]. A drawback intrinsic to static slicing is its inclination to incorporate executable statements that could potentially influence the value of a given variable at a specific statement. This characteristic can lead to the inclusion of extraneous statements within a slice,

which may not be logically appropriate. This phenomenon arises due to the inherent limitations of static analysis in predicting certain runtime values. To mitigate the inclusion of such superfluous statements within both slices and their broader counterparts, called dices, the adoption of dynamic slicing [38, 39] emerges as a more suitable alternative to static slicing. However, it is imperative to note that dynamic slicing-based methodologies carry their own set of limitations. Notably, these techniques are incapable of capturing execution omission errors, a circumstance where the execution of critical statements within a program might be inadvertently omitted, subsequently culminating in operational failures [40].

Our study differs from previous ones in that, we concentrate on examining a specific occurrence of test failure, which arises when an assert statement fails in the middle of a test case, thereby preventing the execution of code after that point. We also investigate the effects of this type of failure on the coverage of the production code.

Investigating the factors that influence testing coverage and the corresponding techniques Masri et al. investigated the prevalence of factors that impair the effectiveness of coverage-based fault localization [41], e.g., the condition for failure is met but the program does not fail, the faulty statement is executed but the program does not fail, etc. They provided certain interesting findings, such as 72% of their studied buggy versions exhibit the condition for failure was met but the program did not fail. More specially, three conditions must be satisfied for program failure to occur: (1) the defect's location must be reached, (2) the program's state must become infected and (3) the infection must propagate to the output. Coincidental correctness (CC) occurs when the program produces the correct output, while condition (1) is satisfied but conditions (2) and (3) are not satisfied. Several studies have been done on this and demonstrate the prevalence of Coincidental correctness and its negative impact on fault detection and localization techniques [41–44]. For instance, Assi et al. investigated the impact of coincidental correctness on test suit reduction (TSR), test case prioritization (TCP), and SBFL [44]. They found that TSR and TCP, the negative impact of CC was

very significant and for SBFL, the impact widely varied based on he metric used. Chekam et al. investigated the impact of clean program assumption (i.e., no known fault in the program) on the coverage measures (i.e., statement, branch, and weak/string mutation testing) and they found that the coverage differs between faulty and clean programs in all measures [45]. They also found that the increased coverage in the statement, branch, and weak mutation testing has little or no effect on fault revelation, while there is a strong connection between fault revelation and strong mutation testing. Madeyski et al. investigate the impact of test-first programming on branch coverage and mutation score of unit tests and found that the impact is minor [46].

Different from prior studies, we focus on investigating the impact of *early test termination* on the coverage and corresponding techniques that rely on it (i.e., SBFL). Our findings show that *early test termination* does negatively impact testing code in various granularities, lines, branches, and methods, as well as impact the effectiveness of SBFL.

Chapter 4

Experimental Design

In this chapter, we first present our research questions (RQs). We then present the data collection process and our approach to answering the research questions.

4.1 Research Questions

The goal of our study is to understand the prevalence of *early test termination* and the impact of *early test termination* on the test coverage and performance of debugging techniques that rely on test coverage. Therefore, we formulate our research to answer the following research questions:

RQ1: How prevalent is early test termination? In RQ1, we investigate how prevalent *early test termination* is in failed tests and the amount of test code that is skipped due to *early test termination*. In addition, we also investigate the prevalence of tests with multiple assertions, since such tests provide the soil for *early test termination*. Note that, if a test only has one assertion, it usually could not be the case of *early test termination*, since the assertion usually is the last line of the code (we have not observed any cases in our dataset where the last line is not an assertion).

RQ2: How does early test termination affect test coverage? In RQ2, we aim to investigate the impact of *early test termination* on the test coverage of production code. More specifically, we

compare the test coverage in two settings: *Original* setting where we do not apply any approach to eliminate the *early test termination*; and *no_stop* setting, where we enforce $Test_{earlyTerminated}$ to keep executing the rest of the code after the failed assertion(s). We introduce how we enforce $Test_{earlyTerminated}$ to keep executing in Section 4.3. By understanding this, we could provide insights to practitioners into the impact of *early test termination* on test coverage.

RQ3: How does early test termination affect the performance of spectrum-based fault localization? In RQ3, we aim to investigate the impact of *early test termination* on the performance of techniques that rely on test coverage information. In this study, we select to study SBFL as our studied case. Note that our methodology could also be used for other techniques that rely on test coverage. Similar to RQ2, we also compare the performance of SBFL techniques in two settings: *early test termination* and *no_stop*.

4.2 Data Preparation

4.2.1 Subject projects

We select six open-source projects for our experiments selected from benchmark datasets Defects4J [19] and T-Evos [1]. Table 4.1 gives an overview of the descriptive statistics of those projects. Defects4J contains real bugs extracted from large open-source projects, which enables controlled experiments in software debugging and testing research [19]. Defects4J has been widely used for evaluating fault localization techniques in the research community [47–50]. We select five projects (i.e., jackson, jfreechart, commons-math (math for short), joda-time, and commons-lang (lang for short)) from Defects4J, because they are from different domains and were used as the benchmark to evaluate various SBFL in prior studies [4, 51]. To increase the diversity of our study, we further expand our dataset with T-Evos, following an existing study [4]. T-Evos provides continuous test execution and failure data to help understand testing practices and evaluate automated

Table 4.1: The overview statistics of our studied projects, which include two parts: the statistics of all versions of studied projects (***All***) and those of versions where *early test termination* happens (***Early test termination***)*.

	All				***Early test termination***				
Project	**Version**	**LOC**	**Avg test**	**Avg T**$_{multi}$	**Version**	**T**$_{total}$	**T**$_{early}$	**T**$_{earlyAssert}$	**C**$_{noexecuted}$
fastjson	12	6K	4,874.0	1691.6 (34.7%)	4	82	32	8	8 (46.0%)
jackson-core	9	18K	396.6	160.9 (40.6%)	2	31	29	2	4.5 (15.3%)
jfreechart	3	92K	2,178.3	1,472.3 (67.6%)	2	4	3	3	19 (60.5%)
math	104	199.2K	2,926.6	1,075.9 (36.8%)	29	170	81	37	20.13 (44.7%)
joda-time	26	26K	3,926.3	2,673.3 (68.1%)	5	74	32	9	18.2 (47.1%)
lang	53	18.68K	1,927.0	1,413.1 (73.3%)	18	110	60	31	13.72 (49.6%)
total	207	N/A	N/A	N/A	60	471	237	90	N/A

* For ***All***, we present the number of versions, the average number of LOC, and the average number of tests in each version, the average number and ratio of tests with multiple assertions (T$_{multi}$) for each project. For ***Early test termination***, we present the number of versions, total failed tests (T$_{total}$), early terminated tests including the ones earlier terminated due to other reasons (T$_{early}$), early terminated tests due to assertion failure (T$_{earlyAssert}$), and average amount (lines of code and ratio) code in a test that is not executed due to assertion failure (C$_{noexecuted}$) for each project.

testing techniques. With T-Evos, we include a new project (i.e., fastjson) in our study. For each subject project, we execute the program on the selected versions provided by the benchmarks. We also verify that the number of tests is sufficient, the program builds successfully, and there is at least one failed test. In total, we collected 207 versions (i.e., commits with failed tests) from six projects for our experiment. Each project has an average number of tests ranging from 1,076 to 4,874.

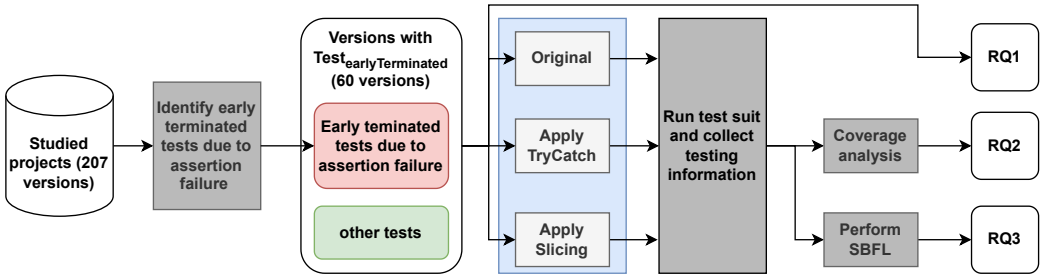

Figure 4.1: The overview framework of answering RQs.

4.3 Methodology

Figure 4.1 presents the overall framework for answering our RQs. First, we identify early terminated tests due to assertion failure (i.e., $Test_{earlyTerminated}$). In RQ1, we perform analysis on those identified $Test_{earlyTerminated}$. In RQ2 and RQ3, we only focus on the versions of the projects that contain $Test_{earlyTerminated}$. We evaluate the performance of SBFL in two settings: *early test termination* and *no_stop*. We have applied *no_stop* approach to execute all lines of code within a test block as early termination skips the rest of the code where assertion failure happens. This early test failure reduces the production code coverage. Therefore, we enforce the program to execute the rest of the code even if assertion failure occurs. For the *no_stop* setting, we apply *Trycatch* as **Baseline** and *Slicing* as **Advanced** technique on $Test_{earlyTerminated}$ to eliminate the *early test termination* and enforce the continuous execution of $Test_{earlyTerminated}$. Subsequently, we re-execute the test while implementing the *Trycatch* and *Slicing* methodologies under the *no_stop* configuration. We gather coverage data and assess its influence on SBFL techniques, to answer RQ2 and RQ3 mentioned in Section 4.1.

4.3.1 Identifying early terminated tests due to assertion failure

The first step of our study is to identify $Test_{earlyTerminated}$ and their corresponding versions. For this purpose, we need to build and run all the tests on every version of our studied projects. Then

```
[INFO] Running com.alibaba.json.bvt.serializer.enum_. EnumTest4
[ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time
elapsed: 0.157 s <<< FAILURE! - in com.alibaba.json.bvt.serializer.
enum_. EnumTest4
[ERROR] com.alibaba.json.bvt.serializer.enum_. EnumTest4.
test_for_enum Time elapsed: 0.108 s <<< FAILURE!
junit.framework. AssertionFailedError: expected: <1> but was:<0>
at com.alibaba.json.bvt.serializer.enum .EnumTest4.test for enum
(EnumTest4.java:20)
```

Figure 4.2: An example of an error message generated when running test file $EnumTest4.java$.

we extract and analyze the failed tests for each version. Among the failed tests, we identify the ones that are $Test_{earlyTerminated}$ by examining the reports generated by JUnit. More specifically, we analyze the testing report to examine: 1) if the terminated line is in the middle of the corresponding test; 2) if the test failure is due to assertion failure based on our definition in Chapter 2. To this end, we collect the log information printed during testing and extract the location of each failure, meanwhile examining whether the failure is due to assertion failure by matching keywords (e.g., "AssertionFailedError"). We identify the location of each failed assertion by analyzing the line number printed in the stack trace. For instance, Figure 4.2 shows the error message, in which "...AssertionFailedError: expected: $< 1 >$ was $< 0 >$" indicates the failure of an assertion and the stack trace "com.alibaba.json.bvt.serializer.enum .EnumTest4.test for enum (EnumTest4.java:20)" indicates the location of the assertion failure. Eventually, we identified 90 $Test_{earlyTerminated}$ distributed in 60 versions (see Table 4.1). Our analysis of the RQs is based on those 60 versions that contain at least one $Test_{earlyTerminated}$.

4.3.2 Enforcing the execution for early terminated test (*no_stop* setting)

To enforce the execution of early terminated tests due to assertion failure, we propose two approaches and elaborate on them below.

***Trycatch*:** We follow an intuitive idea for the first approach and it is considered as our **Baseline** approach. In this approach, we modify a test to enforce it to keep executing even if an assertion failure happens. There are several ways that can achieve this goal. For instance, JUnit 5 provides "asser-

```
1     @Test
2  public void testRootEndpoints() throws MathException {
3     ArrayList<Throwable> er = new ArrayList();
4     UnivariateRealFunction f = new SinFunction();
5     UnivariateRealSolver solver = new BrentSolver();
6     ...
7     result = solver.solve(f, Math.PI, 4, 3.5);
8     // add try-catch block
9     try{
10        //assertion failure
11        assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
12    }catch{Throwable t}{
13        cr.add(t)
14    }
15    ...
16    // ensure the testing result to be the same as original
17    if(er.size() > 0){
18        assertEquals('1', '0'),
19    }
20 }
21
```

Figure 4.3: An example of test after applying *Trycatch* on the motivating example.

tAll()" which asserts that all provided assertions get executed, even if some assertions fail. After test execution, all failures will be aggregated and reported in a MultipleFailuresError [52]. JUnit 4 provides "ErrorCollector" which allows execution of a test to continue after the first problem is found [53]. However, "ErrorCollector" only supports JUnit 4 or above, and "assertAll()" only supports JUnit 5. Some old versions of our studied projects use JUnit 3. To find a way that works for all JUnit versions, we use a try-catch block to catch the AssertionFailedError thrown by assertion failure and enforce the test to keep executing (we call this approach *Trycatch*). *Trycatch* block is used usually for error handling and allows the code outside of the try block to be executed continuously

even if an error occurs in the try block. We add *Trycatch* block outside of failed assertion statement to enforce the execution of early terminated tests. More specifically, we identify every assertion in a test by using regular expressions, such as "assertEquals(.*)" and "assertTrue(.*)", and add a *Trycatch* block around the assertions. For instance, after applying *Trycatch*, the code in Figure 2.1 is modified as shown in Figure 4.3. We add *Trycatch* blocks for all assertions. For space limitation, we only show the *Trycatch* block for line 11. Note that our modification should not change the test result (i.e., failed or passed) of the original test. For this purpose, we add an ArrayList<Throwable> er to collect the AssertionFailedError. If at the end of test execution, the size of the er is larger than 0, which indicates at least one failure happened, we throw an AssertionFailedError in the condition block (lines 17 - 19).

***Slicing*:** Intuitively, *early test termination* is caused by the dependency of statements in the test, i.e., the execution of a later assertion depends on the success of the earlier assertion. Therefore, a potential solution is to disassemble a test with multiple assertions into multiple independent sub-tests, in which only one assertion and its dependent code are contained. We propose an approach to do so using program slicing [54] (we call this approach *Slicing*) and it is our **Advanced** approach for *no_stop*. Compared with *Trycatch*, *Slicing* enables the independent execution of every assertion and its associated code, therefore providing finer-grained granularity than *Trycatch*, especially when multiple assertion failures happen in a single test and may provide more information for SBFL. Now we introduce how *Slicing* is applied to a test. Given a test $t = \{p_1, a_1, p_2, a_2, ..., p_i, a_i, ..., p_n, a_n\}$, where n is the number of assertions in t, a_i is the i^{th} assertion, and p_i is the code block before two assertions a_i and a_{i-1}. We disassemble t into a set of n tests T_{slice}, i.e., $T_{slice} = \bigcup_{i=1}^{n} slicing(t, a_i)$, where $slicing(t, a_i)$ returns the slice of a given code t from a starting point a_i. We will replace the original t with T_{slice}, when running tests. For instance, a sliced test generated from the code in Figure 2.1 using line "assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());" as the starting point is shown in Figure 4.4. We use an open-source tool JavaSlicer [55] to perform program slicing.

```
1     @Test
2     public void testRootEndpoints_3() throws Exception {
3     UnivariateRealFunction f = new SinFunction();
4     UnivariateRealSolver solver = new BrentSolver();
5     double result = solver.solve(f, Math.PI, 4);
6     result = solver.solve(f, Math.PI, 4, 3.5);
7     // assertion failure
8     assertEquals(Math.PI, result, solver.getAbsoluteAccuracy());
9 }
10
```

Figure 4.4: An example of sliced test based on the motivating example.

Note that it is possible that there are dependencies between assertions (although we do not observe such cases in our study). For example, two assertions $a = assertEquals(v1.setValue(4), 1)$ and $b = assertEquals(v1.getValue(), 1)$. The variable $v1$ in b is dependent on the method invocation $v1.setValue(4)$ in a. One solution is to only keep one assertion's (a) dependent statement(s) ($v1.setValue(4)$) in another assertion (b) when performing slicing, while removing the assertion statement itself.

In theory, the test coverage in the production code achieved by *Trycatch* and *Slicing* is the same, since both of those two approaches enforce the execution of all the test code even if one or multiple assertions fail. But for SBFL, *Slicing* provided a more fine-grained spectrum than *Trycatch*, since it disassembles one test into multiple independent sub-tests. Note that in this study, we only applied *Trycatch* and *Slicing* in the $Test_{earlyTerminated}$ for answering our RQs. In reality, practitioners can apply *Trycatch* and *Slicing* on all existing tests as they wish.

4.3.3 Re-running test suit and collecting testing information

For RQ2 and RQ3, we need to rerun all the tests for which we have applied *Slicing* and *Trycatch*, and collect the execution information (i.e., coverage information and test results) for further analysis. Although Defect4J provides a command to get coverage information, it only provides the coverage ratio for the entire project. Our study requires statement-level coverage information for performing SBFL. Therefore, we use Jacoco [56] to collect the coverage information of each individual test. Jacoco is one of the most popular code coverage tools that instruments bytecode to trace the execution during test runs. Jacoco has been widely used for coverage analysis in various studies [1, 57, 58], which can provide coverage information in different granularities, e.g., line, branch, method, and class. However, applying Jacoco is not straightforward and cannot be fully automated. We need to manually configure the maven build script to add the Jacoco dependency since there may be dependency issues that require manual fixes (e.g., some libraries are no longer available in the central Maven repository and need to be manually downloaded, or some versions in the original setting are not compatible with Jacoco). Hence, we manually resolve the issues and update our automation scripts accordingly. We spent took hundreds of hours of manual effort to resolve the compilation issues, which limits the scale of the studied dataset. As one of the first studies on the impact of *early test termination*, we believe our collected dataset provides valuable insight for future research. We also made the data publicly online to encourage future studies in this direction [59].

4.4 Approaches for Answering the RQs

Approach to answer RQ1. To examine the prevalence of *early test termination*, we count the number of $Test_{earlyTerminated}$ among failed tests. We also analyze the characteristics of $Test_{earlyTerminated}$, such as the location of the assertion failure and the amount of test code that is skipped due to *early test termination*. We identify each assertion in a test and the location of each assertion failure as we

described in Section 4.3.1.

Approach to answer RQ2. We collect the coverage information in three configurations as described in Section 4.3.3: *Original*, *Trycatch*, and *Slicing*. We compare their coverage of production code in terms of different granularities (i.e., branches, lines, methods, and instructions). We also count the number of versions for each project, in which the coverage gets improved after applying *Trycatch* and *Slicing* in different coverage granularities.

Approach to answer RQ3. In this RQ, we select Ochiai and Tarantula as the SBFL techniques for evaluation, since these two techniques have good performance and are a commonly used approach for benchmarking SBFL techniques [2, 47, 60]. We compare the performance of both Ochiai and Tarantula in three configurations: *Original*, *Trycatch*, and *Slicing*. We use the implementation of Ochiai and Tarantula from Flacoco [61]. We select Flacoco since it supports both JUnit 3, 4, and 5. In addition, Flacoco is based on Jacoco, which aligns with the tool we use in the experiment setting.

In this RQ, we consider three evaluation metrics: mean EXAM score (*EXAM* for short) [62], Top@k [63], and Mean First Rank (MFR) [64]. These metrics are widely used in prior fault localization studies [2, 47].

EXAM shows the percentage of statements that need to be inspected until it reaches the first faulty statement. The formula of *EXAM* score is defined as follows:

$$\textit{mean EXAM Score} = \frac{1}{n}\sum_{i}^{n}\frac{\textit{Rank of the faulty statements}_i}{\textit{Total number of statement}},$$

where s_i is the ith faulty statement. *EXAM* score ranges from 0 to 1 (inclusive), with a smaller score indicating the better performance of an SBFL technique. As an illustration of the *EXAM* score computation, consider the following suspiciousness score from five code statements (s_1, s_2, s_3, s_4, and s_5), which are 0.6, 0.7, 1.0, 0.5, and 0.8. Assume that s_2 is the faulty statement, the *EXAM* score will be $\frac{3}{5} = 60\%$, since the developer needs to inspect three statements (60% of the code base) to reach the faulty one.

Top@k measures the proportion of bugs with at least one faulty statement localized in the top k positions of the ranked list. We choose to include k=5 and k=10, since a survey by [65] found that the majority of the practitioners only consider a fault localization instance to be successful if the faulty statement appears in the top 5 and 10 positions respectively. A larger Top@k value indicates the better performance of an SBFL technique.

MFR computes, for each project, the mean of the first relevant faulty statement rank for all bugs since the localization of the first buggy element for each bug can be quite crucial for localizing all faulty statements. A smaller MFR indicates the better performance of an SBFL technique.

Note that in the scenario when some statements have the same suspiciousness score, we assign the average rank to these statements by following prior studies [2,47]. The average rank is calculated as $(\frac{n}{2}) + (k - 1)$, where n is the number of statements that have the same suspiciousness score, and k is the best rank of the statement.

Similar to RQ2, we also count the number of versions for each project, in which the performance of SBFL gets improved after applying *Trycatch* and *Slicing* in terms of MFR.

We collected our data and ran our experiments on a Linux server with one Nvidia RTX 3090 GPU, AMD Ryzen 9 12-Core CPU with 64 GB Ram, and 18TB hard drive.

Chapter 5

RESULTS OF THE RESEARCH QUESTIONS

5.1 RQ1: How prevalent is early test termination?

A significant portion of test cases have multiple assertions in a single test, ranging from 34.7% to 73.3% in our different studied projects. From Table 4.1, we observe that fastjson has the lowest percentage of tests containing multiple assertions in a single test, at 34.7% on average. In addition, in the two largest projects math and lang, 36.8% and 73.3% of the tests have multiple assertions. Our findings are compatible with prior studies [17, 22], which show that a test having multiple assertions is prevalent in real-world projects and sets the stage for *early test termination* to occur.

Among the failed tests, a non-negligible portion (19.1%) is due to assertion failures, occurring in 29% of the studied versions. Table 4.1 shows the results of early test termination in the studied projects. Our analysis, conducted across 207 studied versions, reveals that 29% of them contain at least $Test_{earlyTerminated}$, indicating that *early test termination* is prevalent in the studied

projects. Among the 471 failed tests observed, a significant portion (i.e., 50.3%) was terminated early, with a non-negligible portion (i.e., 19.1%) of these early terminated tests being due to assertion failure. To better understand the reasons for early termination that are not due to assertion failure, we manually investigated 20 samples, all of which failed due to errors in object construction or method invocation. For instance, in jfreechart, an error happens frequently when the method $clone$ of $CategoryTableXYDataset$ is invoked, causing the test to terminate early. Future studies may investigate the effect of such failures on testing practices.

On average, 15.3% to 60.5% of the test code is not executed in $Test_{earlyTerminated}$ due to *early test termination*. We examine the amount of code that is not executed due to assertion failure. From Table 4.1, we see that on average, 15.3% to 60.5% of code in $Test_{earlyTerminated}$ is not executed due to assertion failure in our studied projects. The results indicate that around half of the test code in $Test_{earlyTerminated}$ is skipped due to *early test termination*. For instance, in Figure 2.1, there are in total 10 lines of statements, and the assertion fails at line 13, leaving 2 lines of code unexecuted, which accounts for 20% of the test code.

On average, 34.7% to 73.3% of tests have multiple assertions in our different studied projects. Among the failed tests, a non-negligible portion (19.1%) is due to assertion failures and it occurs in 29% of the studied versions, which is not rare. On average, 15.3% to 60.5% of the testing code is not executed in $Test_{earlyTerminated}$ due to *early test termination* in our studied projects.

5.2 RQ2: How does early test termination affect test coverage?

In general, *early test termination* negatively impacts the coverage in the production code in terms of all granularities (i.e., instruction, line, branch, and method). Table 5.2 presents the average amount of missing coverage in terms of instructions, lines, branches, and methods in three settings: *Original*, *Trycatch*, and *Slicing*. Since the coverage results are the same for *Trycatch* and *Slicing*, we show them together. We observe that the average coverage across all the granulari-

Table 5.1: The number of versions that get improved, deteriorated, and tied after applying *Trycatch* and *Slicing* compared with *Original* in terms of different coverage granularities (i.e., Instruction (Ins), Line (Ln), Branch (Br), and Method (Md)). Note the results of *Trycatch* and *Slicing* are the same.

Project	**#Improved**				**#Tied**			
	Ins	**Ln**	**Br**	**Md**	**Ins**	**Ln**	**Br**	**Md**
math	17	18	16	6	12	11	13	23
joda-time	2	2	4	1	3	3	1	4
jfreechart	1	1	0	1	1	1	2	1
jackson	2	2	1	0	0	0	1	2
fastjson	4	4	4	0	0	0	0	4
lang	8	6	8	0	10	12	10	18
Total	34	33	33	8	26	27	27	52

* Note that the number of versions that get deteriorated is zero across all studied projects. To save space, we do not show the results of deteriorated cases.

ties improved after applying *Trycatch* and *Slicing*. In other words, *early test termination* does lead to a lower coverage compared to without *early test termination*. For instance, for math, on average, 10.86 more lines, 5.89 more branches, and 2 more methods are covered after enforcing the $Test_{earlyTerminated}$ to continue execution. We observe a similar trend for other projects.

The impact of *early test termination* will be amplified if we only compare the production code that is only covered by $Test_{earlyTerminated}$, rather than the entire test suit before and after applying *Trycatch* and *Slicing*. For instance, in jfreechat, after applying *Trycatch* and *Slicing*, 16 more lines, 4.5 more branches, and 5 more methods are covered compared to *Original*, of which 106 lines, 17 branches, and 28.5 methods are covered. This is reasonable, since in jfreechat, an average of 60.5% of the test code is skipped due to *early test termination*. Intuitively, if there are fewer tests in the test

suit and assertions fail earlier in the testing process, the impact of *early test termination* is likely to be even more pronounced.

***Trycatch* and *Slicing* improve the line/branch coverage in 55% of the studied versions compared to *Original*.** To provide a better view of how many versions get improved, we compare the coverage of *Original*, *Trycatch*, and *Slicing* for every version. Table 5.1 presents the number of versions in which the coverage gets improved, deteriorated, and tied after applying *Trycatch* compared with *Original* in terms of four granularities, i.e., instruction, line, branch, and method. First of all, the coverage of no version gets deteriorated in all granularities, which is expected, since both *Slicing* and *Trycatch* enforce $Test_{earlyTerminated}$ to keep executing even if an assertion failure happens. In more than half of the versions, the instruction (34/60), line (33/60), and branch (33/60) coverage get improved. In 8 out of 60 versions, the method coverage also gets improved. This finding indicates that different assertions in a test verify different parts of the program, *early test termination* would miss the verification of certain production code.

In general, *early test termination* negatively impacts the coverage in the production code in terms of all granularities (i.e., instruction, line, branch, and method). *Trycatch* and *Slicing* improve the line/branch coverage in 55% of the studied versions compared with *Original*.

5.3 RQ3: How does early test termination affect the performance of spectrum-based fault localization?

Both *Slicing* and *Trycatch* can improve the performance of SBFL by 15.1% and 6.7% on average in terms of MFR for Ochiai, respectively, compared with *Original*. In terms of MFR for Tarantula, both *Slicing* and *Trycatch* can improve the performance of SBFL by 10.66% and 3.81% respectively. Table 5.3 and Table 5.4 present the performance of SBFL in three configurations *Original*, *Trycatch*, and *Slicing* in terms of MFR, EXAM, and Top@k for Ochiai and Tarantula respectively. In general, we observe that both *Slicing* and *Trycatch* outperform *Original* in terms of

Table 5.2: Comparison of the entire coverage in three configurations: *Original*, *Trycatch*, and *Slicing* *.

Original				
Project	**Instruction**	**Line**	**Branch**	**Method**
math	13,799.68	2,763.51	1,583.13	590.68
joda-time	6,271	1,377.6	1,085.6	351
jfreechart	93,353	21,786	11,475.5	3,004
jackson	13,242	3,034	2,197	511.5
fastjson	14,120	3,334	3,809	135
lang	2,821	660	620.1	155.7
***Trycatch* and *Slicing* (improvement)**				
Project	**Instruction**	**Line**	**Branch**	**Method**
math	13,747.37 (52.31)	2,752.65 (10.86)	1,577.24 (5.89)	588.68 (2)
joda-time	6,269.6 (1.4)	1,377.2 (0.4)	1,084.4 (1.2)	350.8 (0.2)
jfreechart	93,347 (6)	21,785 (1)	11,475.5 (0)	3,003 (1)
jackson	13,234 (8)	3,032 (2)	2,196 (1)	511.5 (0)
fastjson	14,112 (8)	3,332 (2)	3,807 (2)	135 (0)
lang	2,815 (6)	659.56 (0.44)	619.44 (0.66)	155.67 (0.03)

* Note that the number presented in the table is for the amount of **missing** coverage in terms of instructions, lines, branches, and methods across all versions of a particular project. For instance, the average number of the missing branches in math is 1583.13 for *Original*. Therefore, a small value indicates better coverage. The value in "()" indicates the extra coverage achieved by *Trycatch* and *Slicing* over *Original*. Note that the results of *Slicing* and *Trycatch* on coverage are the same.

MFR and EXAM in all projects for both Ochiai and Tarantula, except jfreechat, which only has two versions. In the two largest projects math and lang, *Slicing* and *Trycatch* improve the value of MFR for both formulas (Ochiai and Tarantula). For Ochiai, Table 5.3 presents the improvement in the value of MFR from 15.24 to 9.13 (40%) and 11.65 (23.6%) in math, and improve the value of MFR from 30.7 to 27.5 (20.1%) and 23.9 (10.1%) in lang respectively for *Slicing* and *Trycatch*. Table 5.4 presents the improvement for Tarantula, in the value of MFR from 16.41 to 10.62 (35.28%) and

Table 5.3: Comparison of the performance of SBFL in different configurations: *Original*, *Trycatch*, and *Slicing* in terms of MFR, EXAM, Top@k (k = 5 and 10) for **Ochiai**. The cells presenting the best result are marked in bold.

	Original				**Trycatch**				**Slicing**			
Project	**MFR**	**EXAM**	**Top@k**		**MFR**	**EXAM**	**Top@k**		**MFR**	**EXAM**	**Top@k**	
			5	**10**			**5**	**10**			**5**	**10**
math	15.2	0.00191	25.9%	30.1%	11.7	0.00188	**30.9%**	**34.3%**	**9.1**	**0.00183**	26.0%	33.0%
joda-time	596.2	0.03306	0.0%	0.0%	562.6	0.03318	0.0%	0.0%	**431.4**	**0.02733**	0.0%	0.0%
jfreechart	125.5	0.00136	0.0%	0.0%	125.5	0.00136	0.0%	0.0%	125.5	0.00136	0.0%	0.0%
jackson-core	633.0	0.0383	0.0%	0.0%	631.5	0.03821	0.0%	0.0%	**627.5**	**0.03796**	0.0%	0.0%
fastjson	7,198.3	0.16767	0.0%	0.0%	7,182.5	0.167	0.0%	0.0%	**7,175.3**	0.167	0.0%	0.0%
lang	30.7	0.00237	6.92%	10.23%	27.6	0.00236	6.92%	10.23%	**23.9**	**0.00228**	**10.72%**	**14.90%**

Table 5.4: Comparison of the performance of SBFL in different configurations: *Original*, *Trycatch*, and *Slicing* in terms of MFR, EXAM, Top@k (k = 5 and 10) for **Tarantula**. The cells presenting the best result are marked in bold.

	Original				**Trycatch**				**Slicing**			
Project	**MFR**	**EXAM**	**Top@k**		**MFR**	**EXAM**	**Top@k**		**MFR**	**EXAM**	**Top@k**	
			5	**10**			**5**	**10**			**5**	**10**
math	16.41	0.00196	25.91%	29.55%	13.13	0.00190	**32.63%**	**36.06%**	**10.62**	**0.00185**	**32.36%**	**36.05%**
joda-time	586.8	0.03051	0.0%	0.0%	583.2	0.03002	0.0%	0.0%	**469.6**	**0.02715**	0.0%	0.0%
jfreechart	125.5	0.00135	0.0%	0.0%	125.5	0.00135	0.0%	0.0%	125.5	0.00135	0.0%	0.0%
jackson-core	633.0	0.0383	0.0%	0.0%	627.5	0.03796	0.0%	0.0%	**627.5**	**0.03796**	0.0%	0.0%
fastjson	7,271.0	0.16937	0.0%	0.0%	7232.25	0.16846	0.0%	0.0%	**7221.5**	**0.16821**	0.0%	0.0%
lang	25.55	0.00226	10.68%	25.32%	25.33	0.00225	9.29%	24.00%	**23.72**	**0.00213**	**10.90%**	**29.17%**

13.13 (19.98%) in math, and improve the value of MFR from 25.55 to 23.72 (7.16%) and 25.33 (0.87%) in lang.

On average, *Slicing* and *Trycatch* improve the performance of SBFL by 15.1% and 6.7% for

Table 5.5: The number of versions that get improved (Imp), deteriorated (Det), and tied after applying *Trycatch* and *Slicing* compared with original on **Ochiai** in terms of MFR.

	Original vs Trycatch			**Original vs Slicing**		
Project	**#Tied**	**#Imp**	**#Det**	**#Tied**	**#Imp**	**#Det**
math	23	3	3	23	5	1
joda-time	3	1	1	0	5	0
jfreechart	1	0	1	1	0	1
jackson-core	1	1	0	1	1	0
fastjson	0	3	1	0	3	1
lang	12	3	3	12	4	2
Total	40	11	9	37	18	5

Ochiai, and 10.66% and 3.81% for Tarantula compared with *Original* in terms of MFR across all projects, respectively. In terms of Top@k, *Slicing* and *Trycatch* improves *Original* in both math and lang. We do not see improvement in other projects in terms of Top@k, since even the first faulty statements of those projects are ranked beyond 100, although MFR gets improved, but none of the fault statements could be identified in the top 10 results.

Let's consider an example that illustrates why *Slicing* and *Trycatch* outperforms *Original* in some cases. In version 72 of math, both *Slicing* and *Trycatch* outperform *Original*. The test shown in Figure 2.1 tests the function $solve()$ shown in Figure 5.1, which has two buggy lines located in two branches. In *Original*, the test terminates at the third assertion (line 13) and misses the opportunity to reveal the bug (line 16) in the second branch (lines 15 - 18), which is verified by the last assertion in the test (line 17) due to *early test termination*. While in *Slicing* and *Trycatch*, all assertions are executed and both of the buggy lines are identified by SBFL.

***Slicing* outperforms or has the same performance as *Trycatch* for SBFL in terms of MFR**

```
1   public double solve(final UnivariateRealFunction f, final double min, final double max
    , final double initial) throws MaxIterationsExceededException,
    FunctionEvaluationException {
2       ...
3       // return the first endpoint if it is good enough
4       double yMin = f.value(min);
5       if (Math.abs(yMin) <= functionValueAccuracy) {
6           setResult(yMin, 0);    //buggy line
7           return result;
8       }
9       // reduce interval if min and initial bracket the root
10      if (yInitial * yMin < 0) {
11          return solve(f, min, yMin, initial, yInitial, min, yMin);
12      }
13      // return the second endpoint if it is good enough
14      double yMax = f.value(max);
15      if (Math.abs(yMax) <= functionValueAccuracy) {
16          setResult(yMax, 0); //buggy line
17          return result;
18      }
19      ...
20  }
21
```

Figure 5.1: The function which is tested by the code in Figure 2.1.

and EXAM in all projects. When comparing *Slicing* and *Trycatch*, we can see that *Slicing* outperforms or has the same performance as *Trycatch* in terms of EXAM and MFR in all studied projects. If we look at the largest two projects math and lang. In math, *Slicing* improves MFR and EXAM score both for Ochiai and Tarantula by 21.6%, 2.6% and 19.11%, 2.63 respectively, compared with *Trycatch*. In lang, *Slicing* improves MFR and EXAM by 13.1%, 3.4% for Ochiai, and 6.35%, 5.33% for Tarantula, respectively compared with *Trycatch*. In terms of Top@k for Ochiai, in lang, we ob-

Table 5.6: The number of versions that get improved (Imp), deteriorated (Det), and tied after applying *Trycatch* and *Slicing* compared with original on **Tarantula** in terms of MFR.

	Original vs Trycatch			**Original vs Slicing**		
Project	**#Tied**	**#Imp**	**#Det**	**#Tied**	**#Imp**	**#Det**
math	23	3	3	23	5	1
joda-time	3	1	1	0	5	0
jfreechart	2	0	0	2	0	0
jackson-core	1	1	0	1	1	0
fastjson	0	4	0	0	4	0
lang	12	3	3	8	7	3
Total	41	12	7	34	22	4

serve that *Slicing* outperforms *Trycatch* by 54.9% and 45.7% when k equals 5 and 10, respectively. For Tarantula, *Slicing* outperforms *Trycatch* by 14.77% and 17.72% when k equals 5 and 10, respectively for lang. While in math, *Trycatch* performs better than *Slicing* in both formulas (Ochiai and Tarantula). In other relatively smaller projects, *Slicing* has better or equivalent performance as *Trycatch*.

***Slicing* improves the performance of SBFL in 30% and 36.67% of the studied versions for Ochiai and Tarantula respectively. Similarly, for both Ochiai and Tarantula, *Trycatch* demonstrates improvements in 18.3% and 20% of the versions, respectively.** To better understand in which cases the performance gets improved and deteriorated. We present Table 5.5 and Table 5.6 to show the number of versions in which the performance of Ochiai and Tarantula gets tied, improved, and deteriorated after applying *Slicing* and *Trycatch*. We observe that in the majority of the versions, the performance of SBFL gets tied. This is reasonable, since as Table 5.1 shows, in around 45% (27 out of 60) of the versions, the line coverage does not change after applying *Slicing* and *Trycatch*.

Across all projects, we observe that *Slicing* improves the performance of Ochiai and Tarantula at 30% (18 out of 60) and 36.67% (22 out of 60), respectively, and *Trycatch* improves it at 18.3% (11 out of 60) and 20.0% (12 out of 60), both for Ochiai and Tarantula respectively. Overall, *Slicing* outperforms *Trycatch* in terms of both formulas.

Both *Slicing* and *Trycatch* can improve the performance of SBFL by 15.1% and 6.7% on average in terms of MFR for Ochiai, respectively, compared with *Original*. In terms of MFR for Tarantula, both *Slicing* and *Trycatch* can improve the performance of SBFL by 10.66% and 3.81% respectively. *Slicing* improves the performance of SBFL in 30% and 36.67% of the studied versions both for Ochiai and Tarantula respectively, while *Trycatch* improves in 18.3% and 20.0% of the versions for both Ochiai and Tarantula. Overall, *Slicing* outperforms *Trycatch* for SBFL.

Chapter 6

DISCUSSION

6.1 Deteriorated cases analysis

In RQ3, although we observe *Slicing* and *Trycatch* improve the performance of both Ochiai and Tarantula in most of the cases compared with *Original*, particularly for *Slicing*, we still observe a small portion of deteriorated cases. We further manually analyze those deteriorated cases to understand the reason that leads to the downgrade of the performance. We find that *Trycatch* and *Slicing* could bring some noise to both formulas (Ochiai and Tarantula) and causes performance degradation. For example, if a non-buggy statement a is only covered by the testing code after the assertion failure in a $Test_{earlyTerminated}$. In *Original*, a is not returned by both formulas, since the statement is not covered in any failed test, which is expected. While after applying *Trycatch*, a is covered by the code after the assertion failure and returned as a buggy statement by both formulas, causing a false alarm. Similar noise will be introduced in *Slicing* when a non-buggy statement is executed in multiple failed test cases (e.g., a test has multiple assertion failures). In such a way, noise is introduced. For instance, in the method (shown in Figure 5.1), the non-buggy statement *"return result;"* at line 17 is only covered when lines 15 to 17 after the assertion failure (line 13) in Figure 2.1 is executed. In the *Original* setting, *return result;* is not executed in any failed tests,

therefore both Ochiai and Tarantula do not return it as a buggy statement. While, in *Trycatch*, since we enforce the continuous execution of the test code after assertion failure, *return result;* is executed by a failed test, and thus, is returned by SBFL as the rank 1 suspicious statement (suspicious score is 1 since only this test executes *return result;* and fails).

6.2 Threats to Validity

Internal Validity In RQ3, we evaluate SBFL using three commonly used metrics (i.e., EXAM, top@k, and MFR). Although there might be other metrics that could be used for our task, these metrics are commonly used in evaluating SBFL techniques [2, 47].

External Validity Threats to external validity relate to the generalizability of our findings. In RQ3, we selected Ochiai and Tarantula as our subject techniques. *Early test termination* may have a different influence on the effectiveness of other SBFL techniques. However, our proposed framework could be applied to any techniques that rely on test coverage information. Another threat relates to our studied projects. Our findings might not be generalized to other projects. To mitigate the threat, we selected 6 projects that are in various domains from popular benchmarks Defect4J and T-Evos. Among all the 471 versions, 57 versions contain 90 $Test_{earlyTerminated}$. Future research may further expand the study on other datasets. In this study, we focus on early test termination due to assertion failure, there are other reasons (e.g., error in method invitation and object construction) that could lead to early test termination. We encourage future research to investigate other reasons.

6.3 Implications of our findings

We encourage practitioners to use mechanisms (e.g., *assertAll* and *ErrorCollector* to avoid *early test termination* if they have to contain multiple assertions in a single test. In RQ1, we observe that On average, 15.3% to 60.5% of the testing code is skipped in $Test_{earlyTerminated}$ due to *early*

test termination in studied projects. In RQ2, we observe that *early test termination* does reduce the testing coverage in all granularities. Our proposed approaches (*Trycatch* and *Slicing*) that enforce the execution of the code even if an assertion failure happens, improve the line/branch coverage in 55% of the studied versions compared with *Original*. In the recent versions of JUnit, certain mechanisms have been developed, such as *ErrorCollector* starting from JUnit 4 and *assertAll* starting from JUnit 5, to enable the test to keep executing even if an assertion fails in the middle. although having multiple assertions in a single test can be considered a test smell and not recommended, this is a common practice in the studied projects. Therefore, we encourage practitioners to use such mechanisms to avoid *early test termination* if they have to contain multiple assertions in a single test.

We recommend that developers limit each test to a single assertion. In cases where this is not feasible, using *Slicing*. Although both *Slicing* and *Trycatch*, or even other mechanisms (e.g., *assertAll* and *ErrorCollector*) as we mentioned in Section 4.3 can avoid *early test termination* as we observed in RQ2. Our findings in RQ3 show that *Slicing* provides more accurate testing information than other mechanisms and outperforms *Trycatch* in locating faults, since *Slicing* disassembles a test with multiple assertions into a set of independent sub-tests, that enable the execution of every assertion and its associated code independently and provides fine-grained granularity. Therefore, we strongly encourage developers to limit each test to a single assertion whenever possible, and to apply *Slicing* wherever multiple assertions are necessary.

Chapter 7

Conclusion

To assist developers in validating and improving the software quality, assertions in tests are used to verify expected program expected behavior. However, when tests with multiple assertions fail, they will terminate early and skip the remaining assertions, leading to what we refer to as *early test termination*. In this paper, we first study the prevalence of *early test termination* in failed tests and conduct experiments on six widely-used open-source projects in software debugging and testing research from Defects4J and T-Evos benchmarks. Our findings show that 50.3% of failed tests are early terminated, with 19.1% of them caused by assertion failure. We also demonstrate that *early test termination* negatively impacts the code coverage across all granularities and can affect the performance of SBFL. To address this issue, we propose two approaches, *Trycatch* and *Slicing*, that catch assertion failures and enforce test execution to mitigate the impact of early test termination. Our evaluation indicates that Slicing and Trycatch can improve the performance of SBFL by 15.1%, 6.7% for Ochiai, and 10.66%, 3.81% for Tarantula on average in terms of MFR, respectively. Lastly, we provide recommendations to practitioners on testing practices that can prevent *early test termination*. Our study emphasizes the impact of *early test termination* on code coverage and the performance of fault localization techniques that rely on it. We also suggest

that practitioners limit each test to a single assertion or use *Slicing* when multiple assertions are necessary.

Bibliography

[1] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. T-evos: A large-scale longitudinal study on ci test execution and failure. *IEEE Transactions on Software Engineering*, 2022.

[2] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus Agus Haryono, Shaowei Wang, and David Lo. Real world projects, real faults: evaluating spectrum based fault localization techniques on python projects. *Empirical Software Engineering*, 27(6):147, 2022.

[3] Shaowei Wang, David Lo, Lingxiao Jiang, Hoong Chuin Lau, et al. Search-based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 556–559. IEEE, 2011.

[4] An Ran Chen, Tse-Hsun Chen, and Junjie Chen. How useful is code change information for fault localization in continuous integration? In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[5] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[7] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590, 2015.

[8] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[9] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[10] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172*, 2017.

[11] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[12] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. A correlation study between automated program repair and test-suite metrics. In *Proceedings of the 40th International Conference on Software Engineering*, pages 24–24, 2018.

[13] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[14] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coeffi-

cients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[15] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[16] Assertion Roulette. `http://xunitpatterns.com/Assertion%20Roulette.html`. [Online; accessed 2023-03-05].

[17] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[18] Dong Jae Kim. An empirical study on the evolution of test smell. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 149–151, 2020.

[19] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[20] Jifeng Xuan and Martin 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, 2014.

[21] Gina R Bai, Kai Presler-Marshall, Susan R Fisk, and Kathryn T Stolee. Is assertion roulette still a test smell? an experiment from the perspective of testing education. In *2022 IEEE*

Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 1–7. IEEE, 2022.

[22] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M Atlee. Dynamic human-in-the-loop assertion generation. *IEEE Transactions on Software Engineering*, 2022.

[23] Li Jia, Hao Zhong, and Linpeng Huang. The unit test quality of deep learning libraries: A mutation analysis. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 47–57. IEEE, 2021.

[24] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409, 2020.

[25] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 54–64, 2022.

[26] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*, pages 163–174, 2022.

[27] Yoonki Song, Suresh Thummalapenta, and Tao Xie. Unitplus: Assisting developer testing in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 26–30, 2007.

[28] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[29] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. Java language specification, second edition: The java series. 2000.

[30] Anders Hejlsberg, Scott Wiltamuth, and Peter H. Golde. C# language specification. 2003.

[31] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. Learning to handle exceptions. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 29–41, 2020.

[32] Rongfang Li, Bihuan Chen, Fengyi Zhang, Chao Sun, and Xin Peng. Detecting runtime exceptions by deep code representation learning with attention-based graph neural networks. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 373–384, 2022.

[33] Bruno Cabral and Paulo Marques. Exception handling: A field study in java and .net. In *European Conference on Object-Oriented Programming*, 2007.

[34] David W. Binkley and Mark Harman. A survey of empirical results on program slicing. *Adv. Comput.*, 62:105–178, 2004.

[35] Frank Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3, 1994.

[36] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30:1–36, 2005.

[37] Mark D. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. 1979.

[38] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 1990.

[39] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, 1988.

[40] X. Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2007.

[41] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 1–5, 2009.

[42] Robert M Hierons. Avoiding coincidental correctness in boundary value analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):227–241, 2006.

[43] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *2009 IEEE 31st International Conference on Software Engineering*, pages 45–55. IEEE, 2009.

[44] Rawad Abou Assi, Wes Masri, and Chadi Trad. How detrimental is coincidental correctness to coverage-based fault detection and localization? an empirical study. *Software Testing, Verification and Reliability*, 31(5):e1762, 2021.

[45] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608. IEEE, 2017.

[46] Lech Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184, 2010.

[47] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[48] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[49] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2016.

[50] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 2016.

[51] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[52] Class Assertions. `https://junit.org/junit5/docs/5.0.0-M2/api/org/junit/jupiter/api/Assertions.html`. [Online; accessed 2023-03-05].

[53] Class ErrorCollector. `https://junit.org/junit4/javadoc/4.12/org/junit/rules/ErrorCollector.html`. [Online; accessed 2023-03-05].

[54] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[55] A program slicer for Java, based on the system dependence graph (SDG). `https://github.com/mistupv/JavaSlicer`, 2022. [Online; accessed 2023-03-05].

[56] EclEmma team. JaCoCo Java Code Coverage Library. `https://www.jacoco.org/jacoco/`, 2022. [Online; accessed 2023-03-05].

[57] André Silva, Matias Martinez, Benjamin Danglot, Davide Ginelli, and Martin Monperrus. Flacoco: Fault localization for java based on industry-grade coverage. *arXiv preprint arXiv:2111.12513*, 2021.

[58] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*, pages 10–13, 2018.

[59] Dataset Repository. `https://zenodo.org/record/7783450`. [Online; created 2023-03-29].

[60] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.

[61] André Silva, Matias Martinez, Benjamin Danglot, Davide Ginelli, and Martin Monperrus. Flacoco: Fault localization for java based on industry-grade coverage. Technical Report 2111.12513, arXiv, 2021.

[62] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE, 2008.

[63] Tien-Duy B Le, David Lo, and Ming Li. Constrained feature selection for localizing faults. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 501–505. IEEE, 2015.

[64] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

[65] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016.