Test Case Purification for Improving Fault Localization

Jifeng Xuan
INRIA Lille - Nord Europe
Lille, France
jifeng.xuan@inria.fr

Martin Monperrus
University of Lille & INRIA
Lille, France
martin.monperrus@univ-lille1.fr

ABSTRACT

Finding and fixing bugs are time-consuming activities in software development. Spectrum-based fault localization aims to identify the faulty position in source code based on the execution trace of test cases. Failing test cases and their assertions form test oracles for the failing behavior of the system under analysis. In this paper, we propose a novel concept of spectrum driven test case purification for improving fault localization. The goal of test case purification is to separate existing test cases into small fractions (called *purified test cases*) and to enhance the test oracles to further localize faults. Combining with an original fault localization technique (e.g., Tarantula), test case purification results in better ranking the program statements. Our experiments on 1800 faults in six open-source Java programs show that test case purification can effectively improve existing fault localization techniques.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Test case purification, spectrum-based fault localization, test case atomization, dynamic program slicing

1. INTRODUCTION

Finding and fixing bugs are essential and time-consuming activities in software development. Once a bug is submitted, developers must allocate some effort to identify the exact location of the bug in source code [15]. The problem of localizing bugs in a program is known as *fault localization*, which consists of automatically ranking program entities (e.g., program methods or statements) based on an oracle of the bug, usually a failing test case [39]. *Spectrum-based fault localization* (also known as *coverage-based fault localization*) is a family of methods that use the execution trace of test cases (i.e., the coverage data) to measure the faultyness probabilities of program entities. For example, Tarantula [15], Jaccard [1], and Ochiai [1] are popular spectrum-based fault localization techniques. According to the fault localization rankings, the developers manually examine the program under debugging to find out the location of the bug.

In modern test-driven software development, unit testing plays an important role for ensuring the quality of software. A unit test framework, such as JUnit for Java, NUnit for .Net, and CPPUnit for C++, provides a platform for developers to manage and automatically execute test cases [21]. Each test case is formed as a test method, which employs a test oracle to ensure the expected behavior. The test oracle in a test case is implemented as a set of executable *assertions* for verifying the correctness of the program behavior. For instance, an open source project, Apache Commons Lang (Version 2.6), consists of 1874 test cases with 10869 assertions testing the behavior of over 55K lines of code. That is, each test case includes 5.80 assertions on average. If an assertion in a test case is violated, the unit test framework aborts the execution of this test case and reports the test result (i.e., the test case is failed).

Test cases can be employed for fault localization [31], [34], [6]. Aborting the execution of a failing test case omits all the unexecuted assertions that are in the same test case. However, the effectiveness of fault localization depends on the quantity of test oracles. *Our key intuition is that recovering the execution of those omitted assertions can lead to more test cases and further enhance the ability of fault localization.*

In this paper, we propose the concept of spectrum driven test case purification (*test case purification* for short) for improving fault localization. The goal of test case purification is to generate *purified* versions of failing test cases, which include only one assertion per test and excludes unrelated statements of this assertion. We leverage those purified test cases to better localize software faults in Java projects. Test case purification for fault localization consists of three major phases: test case atomization, test case slicing, and rank refinement. First, test case atomization generates a set of single-assertion test cases for each failed test case; second, test case slicing removes the unrelated statements in all the failing single-assertion test cases; third, rank refinement

FSE'14 , November 16–22, 2014, Hong Kong, China

combines the spectra of purified test cases with an existing fault localization technique (e.g., Tarantula) and sorts the statements as the final result.

We evaluate our work on six real-world open-source Java projects with 1800 seeded bugs. We compare our results with six mature fault localization techniques. Our experimental results show that test case purification can effectively improve the results of existing techniques. Applying test case purification achieves better fault localization on 18 to 43% of faults (depending on the subject program) and performs worse on only 1.3 to 2.4% of faults. In terms of fault localization, test case purification on Tarantula (*Tarantula-Purification* for short) obtains the best results among all the techniques we have considered. Tarantula-Purification performs better than Tarantula on 43.28% of the faults with an average fault-localization improvement of 36.44 statements. With Tarantula-Purification, developers can save half of the effort required for examining faulty statements.

This paper makes the following major contributions.

1. We propose the concept of spectrum driven test case purification for improving spectrum-based fault localization. In contrast to novelty in the suspiciousness metric that is common in the fault localization literature, we explore a novel research avenue: the manipulation of test cases to make the best use of existing test data.

2. We empirically evaluate our approach on 1800 seeded faults on six real-world projects. We compare the fault localization effectiveness of six state-of-the-art techniques (Tarantula, SBI, Ochiai, Jaccard, Ochiai2, and Kulczynski2) with and without test case purification.

The remainder of this paper is organized as follows. Section 2 presents the background and motivation of our work. Section 3 proposes the approach to test case purification for improving fault localization. Sections 4 and 5 show the data sets in the experiments and the experimental results. Section 6 states the threats to validity in our work. Section 7 lists the related work and Section 8 concludes this paper.

2. BACKGROUND AND MOTIVATION

2.1 Terminology

We define the major terms used in this paper to avoid ambiguous understanding. A *test case* (also called a *test method*) is an executable piece of source code for verifying the behavior of software. In JUnit, a test case is formed as a test method, which consists of two major parts, a test input and a test oracle. A *test input* is the input data to execute the program while a *test oracle* determines the correctness of the software with respect to its test input. Test oracles are created by developers according to business and technical expectations. A test oracle is implemented as a set of executable assertions to ensure that the software performs as expected. A *test suite* is a set of test cases.

An *assertion* is a predicate (a binary expression) that indicates the expected behavior of a program. If an assertion is not satisfied, an exception is thrown. Then the test case containing this assertion aborts and the testing framework reports the failure. For example, `assertEquals(a, b)` in JUnit is widely used to ensure the equality of values `a` and `b`. In practice, a single test case can consist of many assertions (see Section 4.1 for details).

A *subject program* (also called a *proband* [31] or an *object program* [15]) is a program under test. Based on a unit testing framework, like JUnit, a test suite can be automatically executed to test the program.

A *program entity* represents an analysis granularity for fault localization. For instance, a program entity can be a class, a method, a statement, etc. In this paper, we focus a widely-used program entity, i.e., a statement [15], [39], [3].

A *spectrum* of a test case is a set of program entities decorated with execution flags. For a given test case, a *flag* of a program entity indicates whether the test case executes (a.k.a. covers) this particular program entity.

In this paper, we focus on subject programs written in Java and tested with JUnit, a unit testing framework. Both JUnit 3 and JUnit 4 are widely used in current Java projects. An intuitive difference between these two versions is that a test case in JUnit 4 starts with a specific annotation `@Test` and a test case in JUnit 3 is named with a specific convention (in a `testMethod` style). Our work supports test cases in both versions of JUnit. Figure 1(b) briefly illustrates an example of a test case in JUnit 4.

2.2 Spectrum-Based Fault Localization

Spectrum-based fault localization [15], [1], [33] (also known as coverage-based fault localization [31]) is a family of approaches to identifying the exact location of bugs in source code. Popular techniques include Tarantula [15] and Ochiai [1]. The input of those approaches is the subject program with its test suite. Spectrum-based fault localization executes the whole test suite and collects the spectrum of each test case. All spectra of test cases form a *spectrum matrix* (also called a *test coverage matrix*) and each element in the matrix indicates whether a test case covers a statement. Based on the spectrum matrix, a fault localization approach calculates the suspiciousness for all statements and ranks them according to their suspiciousness. A detailed description of existing fault localization techniques can be found in Section 4.2.

2.3 Motivation

Figure 1 shows a fraction of a subject program in Apache Commons (AC) Lang 2.6. AC Lang is an extension library for the Java programming language. Figure 1(a) lists several methods for the calculation of the maximum and the minimum for IEEE 754 floating-point numbers [40]. Note that we have omitted the modifiers of methods and several statements to reduce the space.

We inject a fault at Line 20, i.e., `if(! Float.isNaN(a))`, by negating the original conditional expression. Then we execute all the test cases of AC Lang and the only failing test case during execution can be found in Figure 1(b). We call this failing test case `t1`. The test case `t1` aborts since the assertion at Line 6 is unsatisfied. In all the test cases of AC Lang, only `t1` fails and 11 statements from Line 3 to Line 21 are executed by `t1` (as shown in Figure 1(a). We use Tarantula [15] as an example of fault localization technique. Based on the actual execution of Tarantula, all the 11 statements executed by `t1` are ranked with the same suspiciousness. Thus, it is hard to identify the fault at Line 20 from these statements.

However, `t1` is aborted at Line 6 and the last assertion at Line 9 has not been executed. Thus, we consider making use of the unexecuted assertion to improve fault localization. As shown in Figure 1(c), we create three copies (`a1`, `a2`, and `a3`) of `t1`; for each copy, we force two assertions to not throw an

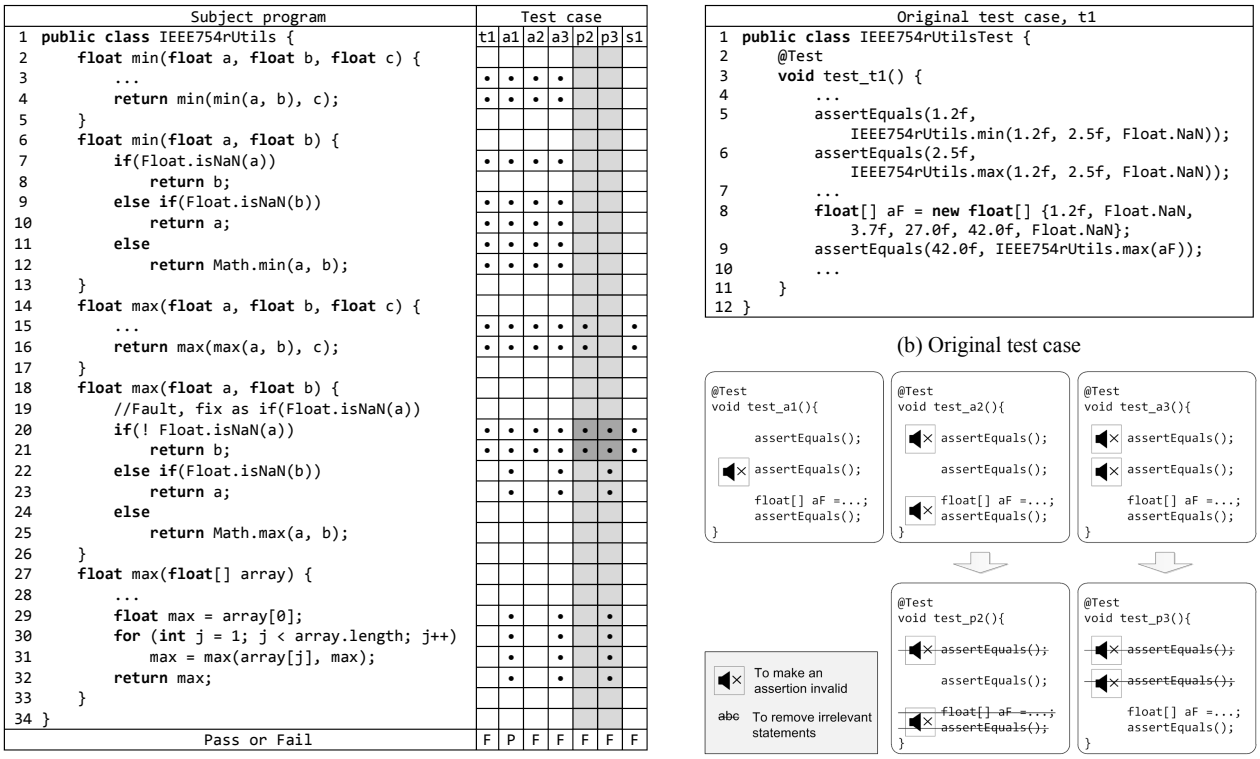

Figure 1: Example of test case purification. The subject program and test cases are extracted from Apache Commons Lang 2.6. Test cases `t1` is the original test cases. Test cases `a1`, `a2`, `a3`, `p2`, and `p3` are generated during test case purification. Test case `s1` is an extra example for the explanation.

exception even if the assertion is unsatisfied. That is, each of test cases `a1`, `a2`, and `a3`, has only one valid assertion. Then we execute test cases `a1`, `a2`, and `a3`; we find that `a2` and `a3` fail at Line 6 and Line 9, respectively (actually, `a2` in this execution expresses the same behavior as `t1`). For each of `a2` and `a3`, we remove the irrelevant statements to Line 6 and Line 9, respectively; then we get two smaller test cases `p2` and `p3`. We execute `p2` and `p3` and the spectra are represented as columns in Figure 1(a) in gray. Based on the spectra of `p2` and `p3`, statements at Line 20 and Line 21 are executed twice and six other statements are executed only once. Thus, we can rank the two statements at Line 20 and Line 21 as faulty statement, prior to the other statements.

The reason for ranking the last two statements is that these statements are the frequently executed ones by failing test cases. In other words, the fault in source code causes the failure of `p2` and `p3` and the spectra of `p2` and `p3` are different. Thus, the two statements are the most suspicious based on the evidence from the two test cases `p2` and `p3`. Moreover, if we directly remove irrelevant statements for the original test case `t1`, all the dependent statements like Lines 15 and 16 will be kept, as shown in `s1` in Figure 1(a). For a large subject program, a large number of dependent statements often interrupt the identification of the fault.

This example motivates our work, test case purification for fault localization. We use test case purification to generate small fractions of test cases to improve the existing techniques in fault localization.

3. TEST CASE PURIFICATION

In this section, we propose the concept of spectrum driven test case purification (*test case purification* for short) for fault localization. We first present the framework in Section 3.1. Then we show the details of the three main phases in Sections 3.2, 3.3, and 3.4, respectively. Finally, we discuss the extensibility of test case purification in Section 3.5.

3.1 Framework

The main goal of test case purification is to generate purified test cases from each failing test case. A *purified test case* is a short test case with only one assertion and is generated by removing several statements from the original failing test case. We employ such purified test cases to improve existing techniques on fault localization.

Figure 2 illustrates the framework of test case purification for fault localization. This framework consists of three major phases: test case atomization, test case slicing, and rank refinement. Given a specific technique on fault localization, the input of test case purification is a subject program with its test suite and the final output is a ranking of statements. Both the input and the output are the same as those in typical fault localization techniques, e.g., Tarantula and Ochiai.

In **test case atomization**, each original failing test case with k assertions is replaced by k single-assertion test cases. A single-assertion test case is a copy of the original test case, but only one out of k original assertions is kept. In **test case slicing**, each single-assertion test case is treated

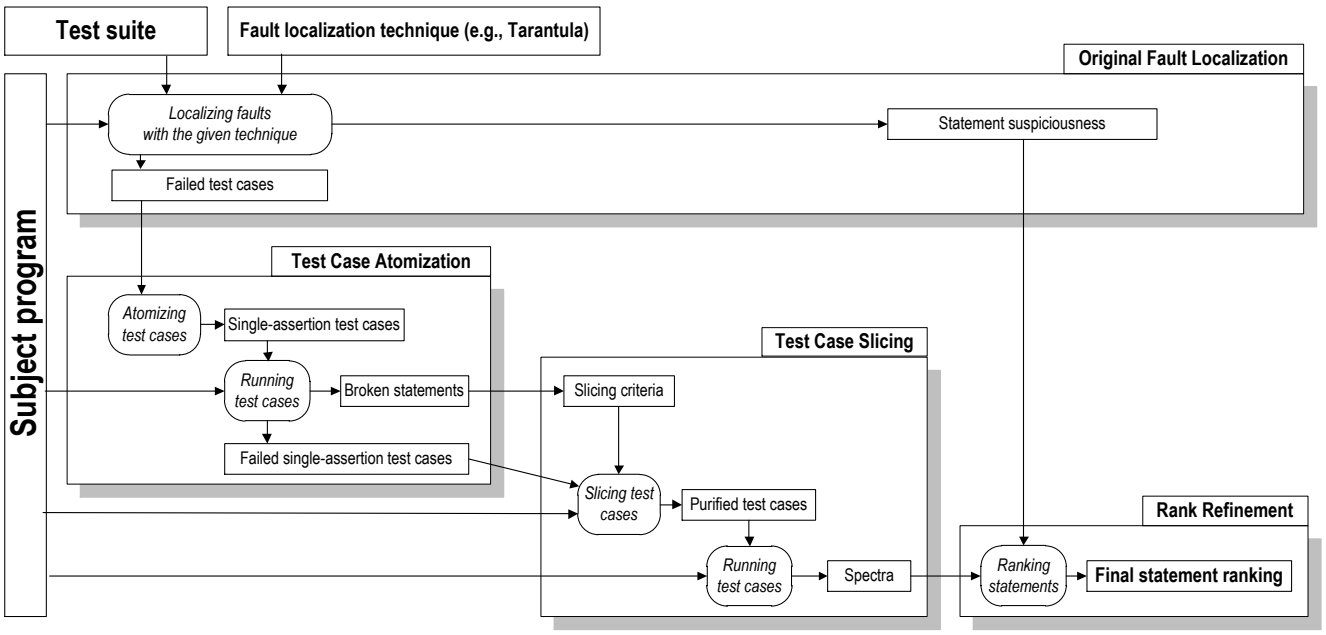

Figure 2: Framework of test case purification for fault localization. This framework consists of three phases: test case atomization, test case slicing, and rank refinement.

as a program. We use dynamic slicing technique to remove irrelevant statements in each single-assertion test case. Then short test cases are generated as purified test cases. In **rank refinement**, we re-rank the statements in an existing fault localization technique based on the spectra of all the purified test cases.

In our work, test case purification for fault localization is run automatically. We describe the implementation details in Section 4.3.

3.2 Test Case Atomization

The goal of test case atomization is to generate a set of test cases for each failing test case. As the term *atomization* suggests, we consider each assertion has an atomic part in a test case. Given a failing test case with k assertions, we create k copies for this test case and we transform $k-1$ assertions into regular test case statements for each copy (no exception from the assertion reaches the testing framework if the assertion fails). To transform an assertion into a regular test case statement, we surround this assertion with a `try-catch` structure shown in Figure 3.

In Java, the class `java.lang.Throwable` is a superclass of all the exceptions. As mentioned in Section 2.1, an exception will be thrown to the test case if an assertion is not satisfied. Based on the above structure, the exception will be caught as `throwable` and the test case will not be interrupted[1]. Based on the surrounding structure in Figure 3, a set of k single-assertion test cases are created to replace each originally

```
try {
    /* assertion */
}
catch (java.lang.Throwable throwable) {
    /* do nothing */
}
```

Figure 3: A surrounding structure for transforming an assertion into a regular test case statement (no exception from the assertion reaches the testing framework if the assertion fails).

failing test case. A failing test case with only one assertion will be kept without handling.

Note that in JUnit, two kinds of interruptions will stop the execution of a test case, namely a failure and an error. A *failure* is caused by an unsatisfied assertion, which is designed by developers; an *error* is caused by a fault, which is not considered by developers [10]. Thus, an error may appear in any statement of a test case. In test case atomization, we only deal with the failures (in assertions) in JUnit. If an error appears, the execution of a single-assertion test case will be aborted because an error usually causes severe problems, which are beyond the expected test cases by developers.

After generating single-assertion test cases, we compile and execute all the single-assertion test cases. Meanwhile, we collect the failing ones among these test cases; for each failing single-assertion test case, we record its position that aborts the execution. This position is referred as a *broken statement*. For example, a broken statement in a single-assertion test case could be an assertion (i.e., the exact assertion left in the test case) or a statement that throws an

[1]Sometimes an assertion is originally surrounded by a `try-catch` statement, e.g., writing files may throw an `IOException` in Java. Directly adding the surrounding structure to this assertion will cause the compiling error. In this case, we collect the candidate exceptions and add a fraction of dead code to make the compiling pass, like `if (false) throw new IOException()`.

unexpected error. Finally, each failing single-assertion test case as well as its broken position is collected.

3.3 Test Case Slicing

The goal of test case slicing is to generate purified test cases before collecting their spectra. Given a failing single-assertion test case resulting from test case atomization, we slice this test case by removing irrelevant statements.

Program slicing can be mainly divided into two categories: static slicing [5] and dynamic slicing [38]. Informally, static slicing keeps all the possible statements based on static data and control dependencies while dynamic slicing keeps the actually executed statements in the dynamic execution (with dynamic data and control dependencies). In test case slicing, we use a dynamic slicing technique to remove statements in test cases since dynamic slicing may lead to more removal of statements [38]. In dynamic slicing, a slicing criterion should be specified before execution the program. A slicing criterion is defined as a pair $< b, V >$, where b is a statement in the object program and V is a set of variables to be observed at b.

We perform dynamic slicing and slice single-assertion test cases during its execution by the Junit framework. Our slicing criterion for a test case is its broken assertion with all the variables at this statement. Then we execute the dynamic slicing technique to collect the statements that will be removed. After the slicing, each failing single-assertion test case in test case atomization is updated with a purified test case. Then we execute these purified test cases on the project program and record the spectra for next phase.

3.4 Rank Refinement

The goal of rank refinement is to re-rank the statements by an existing fault localization technique with the spectra in the phase of test case slicing.

In all the purified test cases, we keep only one test case if two or more than test cases have the same spectrum. As mentioned in Section 3.3, all the purified test cases are failing test cases. Let S be a set of candidate statements. We define the ratio of a statement $s \in S$. First, for a statement $s \in S$ that is covered during the execution of all the purified test cases, $ratio(s) = \frac{\beta_{ef}(s)}{\beta_{ef}(s)+\beta_{nf}(s)}$, where $\beta_{ef}(s)$ and $\beta_{nf}(s)$ are the numbers of test cases covering and non-covering s. Second, for a statement s that is not covered by any purified test case, we directly set $ratio(s) = 0$.

The output of an existing fault localization technique, such Tarantula or Ochiai, is the suspiciousness values for all the candidate statements. Let $susp(s)$ be the suspiciousness value of a statement $s \in S$ in a fault localization technique. Then we normalize the $susp(s)$ as 0 to 1 for all the statements in S. The normalized suspiciousness value is defined as $norm(s) = \frac{susp(s)-min(S)}{max(S)-min(S)}$, where $min(S)$ and $max(S)$ denote the minimum score and the maximum score for all the statements in S, respectively.

For each statement $s \in S$, both $ratio(s)$ and $norm(s)$ is between 0 and 1 (both inclusive). Then we refine the ranking of each statement s by combining $ratio(s)$ and $norm(s)$. The final score of s is defined as $score(s) = norm(s) \times \frac{1+ratio(s)}{2}$. Then for all the statements $s \in S$, the final score $score(s)$ is between 0 and 1 (both inclusive). Based on the final scores of all the statements, we re-rank the statements as the result of fault localization by test case purification.

3.5 Discussion

Basic fault localization technique. Test case purification modifies the existing test cases. Consequently, the spectra are changed and the suspicious statements according to a fault localization technique (e.g., Tarantula) are re-ranked. Many other fault localization techniques can be used instead, such as Ochiai, Jaccard, and SBI. We examine the results for six fault localization techniques in Section 5.

Method of rank refinement. We define the new score of each statement s as $score(s)$. This definition can be replaced by other formulae, for example, the average of $norm(s)$ and $ratio(s)$, i.e., $\frac{norm(s)+ratio(s)}{2}$, or the geometric mean, i.e., $\frac{2\times norm(s)\times ratio(s)}{norm(s)+ratio(s)}$. Results of such refinement methods can be further explored.

4. EXPERIMENTAL SETUP

4.1 Subject Programs

We select six open-source subject programs for our experiments. Table 1 gives the key descriptive statistics of those subject programs. All six programs are Java libraries, which are widely used in fault localization research [6], [14], [30], [31]. We compute the size metric (Source Line of Code - SLoC) with CLOC[2]. The subject programs in our selection are provided with large test suites written in JUnit. For each subject program, we execute the original program with its dependent libraries. We confirm that the whole test suite passes, i.e., our experimental configuration is correct.

We follow existing work in [14], [31] and use mutation testing tools to create faulty versions. A *mutant* of a program is a copy of the original program with a single change. For instance, a mutant may contain one change of negating a conditional statement. Mutants are meant to simulate likely faults made by developers. Some of mutants (known as *equivalent mutants*) provide the same observable output as the original program. We employ six mutant operators to generate all the mutants for a given subject program. Table 2 presents the six mutant operators for generating faulty versions. In our work, we use the PIT tool[3] to generate mutants which has implemented all these six operators. We discard equivalent mutants and keep the faulty versions. Finally, we randomly select 300 mutants from all the seeded faulty versions for each subject program as the final dataset of faulty programs. A thorough study by Steimann et al. [31] has shown that a sample size of 300 mutants gives stable fault localization results.

4.2 Techniques in Comparison

As explained in Section 3, the goal of test case purification is to improve existing fault localization by maximizing the usage of all the assertions. In our experiments, we evaluate the effectiveness of test case purification on six well-studied fault localization techniques: Tarantula, Statistical Bug Isolation (SBI), Ochiai, Jaccard, Ochiai2, and Kulczynski2 ([1], [20], [24], [34], [37]).

Jones et al. [16], [15] propose Tarantula for fault localization. Tarantula ranks statements by differentiating the execution of failing and passing test cases. SBI is proposed by Liblit et al. [19] and calculates the suspiciousness value.

[2]CLOC, `http://cloc.sourceforge.net/`.
[3]PIT 0.27, `http://pitest.org/`.

Table 1: Subject programs with source code, test suites, and faulty versions

Subject program	Program source		Test suite						Faulty version	
	#Classes	SLoC	#Classes	SLoC	JUnit version	#Test cases	#Assertions	#Assertions per test case	#Mutants	#Faults
JExel 1.0.0 beta13	45	2638	43	9271	4	343	335	0.98 †	347	313
JParsec 2.0.1	100	9869	38	5678	4	536	869	1.62	1698	1564
Jaxen 1.1.5	197	31993	100	16330	3	520	585	1.13	3930	1878
Apache Commons (AC) Codec 1.9	56	13948	53	14472	4	547	1446	2.64	2525	2251
Apache Commons (AC) Lang 2.6	83	55516	127	43643	3	1874	10869	5.80	8830	7582
Joda Time 2.3	157	68861	156	69736	3	4042	16548	4.09	9197	7452

† In some programs, assertions are abstracted into a specific class, which are not the same assertions in JUnit. In our work, we only handle the assertions in JUnit. Thus, the # assertions per test case can be less than 1.

Table 2: Mutant operators for generating faulty versions

Mutant operator	Description
Invert negatives	Invert an integer or a floating-point number as its negative
Return values	Change a returned object to null, or increase (or decrease) a returned number
Math	Replace a binary math operator with another math operator
Negate conditionals	Negate a condition as its opposite
Conditional boundary	Add or remove the boundary to a conditional statements
Increments	Convert between an increment (++, +=) and a decrement (–, -=)

Table 3: Six spectrum-based fault localization techniques in comparison

Technique	Definition
Tarantula	$\frac{\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{nf}(s)}}{\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{nf}(s)}+\frac{\alpha_{ep}(s)}{\alpha_{ep}(s)+\alpha_{np}(s)}}$
SBI	$\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{nf}(s)}$
Ochiai	$\frac{\alpha_{ef}(s)}{\sqrt{(\alpha_{ef}(s)+\alpha_{nf}(s))(\alpha_{ef}(s)+\alpha_{ep}(s))}}$
Jaccard	$\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{nf}(s)+\alpha_{ep}(s)}$
Ochiai2	$\frac{\alpha_{ef}(s)\alpha_{np}(s)}{\sqrt{(\alpha_{ef}(s)+\alpha_{ep}(s))(\alpha_{np}(s)+\alpha_{nf}(s))(\alpha_{ef}(s)+\alpha_{nf}(s))(\alpha_{ep}(s)+\alpha_{np}(s))}}$
Kulczynski2	$\frac{1}{2}\left(\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{nf}(s)}+\frac{\alpha_{ef}(s)}{\alpha_{ef}(s)+\alpha_{ep}(s)}\right)$

Their work shows that the predicted suspicious statements correlate with the root cause. Ochiai is proposed by Abreu et al. [1], which counts both failing test cases and executing test cases. Jaccard is also proposed by Abreu et al. [1]. Those four techniques are the most widely-used ones for the evaluation of fault localization. Ochiai2 by Naish et al. [24] is an extension version of Ochiai; the difference is that Ochiai2 considers the impact of non-executed or passing test cases. Kulczynski2 by Naish et al. [24] is another widely-used metric. Evaluations of Ochiai2 and Kulczynski2 can be found in [20], [24], [34].

Generally, a spectrum-based fault localization technique can be formalized as a formula of calculating the suspiciousness values,

$$susp(s) = f\big(\alpha_{ef}(s), \alpha_{nf}(s), \alpha_{ep}(s), \alpha_{np}(s)\big)$$

where $\alpha_{ef}(s)$ and $\alpha_{nf}(s)$ are the numbers of failing test cases that execute and do not execute the statement s while $\alpha_{ep}(s)$ and $\alpha_{np}(s)$ are the numbers of passing test cases that execute and do not execute the statement s, respectively. Table 3 summarizes the six techniques that we consider for evaluating test case purification.

For a given fault localization technique, the *wasted effort* of localizing the faulty statement is defined as the rank of the faulty statement in the ranking according to the suspiciousness values. For statements with the same suspiciousness values, the wasted effort is the average rank between all of them. Formally, the wasted effort of fault localization is defined as

$$\text{StmtEffort} = |s \in S | susp(s) > susp(s^*)| + \frac{1}{2}|s \in S | susp(s) = susp(s^*)| + \frac{1}{2}$$

where S is a set of candidate statements, $s^* \in S$ is the faulty statement, and $|\cdot|$ indicates the size of a set.

4.3 Implementation

We now discuss the implementation details of our experiment. Our test case purification approach is implemented in Java 1.6. Our experiments run on a machine with an Intel Xeon 2.67 CPU and an Ubuntu 12.04 operating system. Our implementation automatically runs the three phases in Figure 2.

In our work, test suites are automatically executed with Ant 1.8.4[4] and JUnit 4.11. We set the timeout of running a faulty program as five times of that of the originally correct version to avoid performance bugs [25], which may be potentially generated during the program mutation. We execute an existing fault localization technique to compute the original suspiciousness values. We implement the six existing fault localization techniques on top of GZoltar 0.0.3[5]. GZoltar [7] is a library for facilitating and visualizing fault localization. We use GZoltar to collect the program spectra.

In the phases of test case atomization and test case slicing, we directly manipulate test cases with Spoon 1.5[6]. Spoon [27] is a library for Java source code transformation and analysis. With the support by Spoon, a Java test class is considered as an abstract syntax tree; and we modify source code via programming abstractions. Spoon also handles annotations in Java hence our implementation fully supports both JUnit 3 and JUnit 4.

In the phase of test case slicing, we slice test cases with JavaSlicer[7]. JavaSlicer [12] efficiently collects runtime trace

[4]Ant 1.8.4, `http://ant.apache.org/`.
[5]GZoltar 0.0.3, `http://www.gzoltar.com/`.
[6]Spoon 1.5, `http://spoon.gforge.inria.fr/`.
[7]JavaSlicer, `https://www.st.cs.uni-saarland.de/javaslicer/`.

Table 4: Number of faults where test case purification improves existing fault localization techniques (column *Positive*), worsens (column *Negative*) and has no impact (column *Neutral*). Each number is computed over 1800 seeded faults in six subject programs.

Technique in comparison	Positive		Negative		Neutral	
	# Faults	Percent	# Faults	Percent	# Faults	Percent
Tarantula	779	**43.28**	44	**2.44**	977	54.28
SBI	722	**40.11**	24	**1.33**	1054	58.56
Ochiai	373	**20.72**	28	**1.56**	1399	77.72
Jaccard	360	**20.00**	28	**1.56**	1412	78.44
Ochiai2	330	**18.33**	28	**1.56**	1442	80.11
Kulczynski2	666	**37.00**	24	**1.33**	1110	61.67

for a subject program and removes traces offline with dynamic backward slicing. JavaSlicer requires specifying the point of a thread. Thus, we develop a driver program to facilitate the test case slicing. Since program slicing techniques may cost time and resources, it is necessary to decide how many test cases should be sliced. Based on our experience, it seems that slicing failing test classes one by one is the most efficient, compared to handling failing test cases one by one or all the failing test cases together.

5. EXPERIMENTAL RESULTS

In this section, we present our experimental results on test case purification. Section 5.1 presents the overall comparison based on all seeded faults in six subject programs; Section 5.2 discusses the detailed results for each subject program; Section 5.3 evaluates the time cost of test case purification.

5.1 Overall Comparison

We compare the capability of our test case purification technique to improve six existing fault localization techniques on six subject programs.

Table 4 presents the average fault localization results on 1800 seeded faults with mutation. The columns *Positive* gives the absolute and relative numbers of faults, which are improved after applying test case purification, compared to basic techniques in fault localization. Column *Negative* indicates the number of faults when the basic fault localization gives better results. Column *Neutral* shows the number of faults, which are not changed after applying test case purification.

As shown in Table 4, test case purification improves fault localization for basic fault-localization techniques. For instance, by applying test case purification, 779/1800 (43%) of faults for Tarantula achieve lower wasted efforts (i.e. faults are easier to be localized and the results are better). The number of faults where purification worsen the ranking is small (worse in 2.44% of faults for Tarantula), and much smaller than the number of faults that are improved. Except Tarantula, test case purification decreases the effectiveness of fault localization in no more than 28/1800 faults.

We note that for all the six techniques in our comparison, we obtain neural results on over 50% of faults. The main reason is that some of the considered faults are easy to localize. For example, for Jaccard, root-cause statements for 389/1800 faults are directly ranked as the first; in those cases, our approach cannot improve the localization since the results are already optimal. Meanwhile, for Jaccard again, 1009/1800 root-cause statements are ranked between the 2nd to the 10th position; and consequently the localization of these faults is hard to improve. In Section 5.2, we will show that our test case purification works well for the difficult faults, which are originally localized beyond the top-10 statements.

Table 5 presents the wasted effort with or without applying test case purification on 1800 faults. The wasted effort is measured with the absolute number of statements to be examined before finding the faulty one (see Section 4.2). It is the main cost of fault localization. In total, there are 12 competing techniques (six fault localization techniques with or without purification). Tarantula with test case purification (called *arantula-Purification* for short) gives the best results among 12 techniques for three of six subject programs. Ochiai-Purification gives the best results for the remaining three subject programs. The last row in Table 5 gives the average results over all six subject programs. According to this aggregate measure, purification test case improves the wasted effort from 72.06 statements (Tarantula) to 35.62 statements (Tarantula-purification). By applying test case purification with Tarantula, developers save 36.44 statements to examine. In the worst case, they still save 8 statements.

Summary. Applying test case purification to the state-of-the-art fault localization techniques results in up to 43% positive results with the price of 2.4% worsened faults. Among 12 techniques in comparison, Tarantula-Purification obtains the best results, which are 18.22% better than the best original technique according to our experimental setup (without purification, Ochiai is the best technique with an average wasted effort of 43.56 statements).

5.2 Detailed Comparison per Fault Category

To better understand the effectiveness of test case purification, we analyze all faults in our six subject programs in details. Let $s_{original}$ denotes the original fault localization result, i.e., the wasted effort of localizing the faulty statements as described in Section 4.2. We divide the faults in subject programs into three categories according to $s_{original}$, namely faults with $s_{original} = 1$, $1 < s_{original} \leq 10$, and $s_{original} > 10$. For example, the faults with $s_{original} = 1$ can be viewed as the easy category where there is no space for improving the fault localization since the results are optimal. Similarly, faults with $1 < s_{original} \leq 10$ can be viewed as the medium category. It is a reasonable task for a developer to examine the top-10 suspicious statements in a program; Le & Lo [17] also suggest that localizing a fault in top-10 statements is a proof of effectiveness. Results of such faults can be improved a bit. Faults with $s_{original} > 10$ can be viewed as representing the hard category. More wasted efforts may need to be checked to localize the faults.

Table 6 shows the detailed evaluation on faults in those three categories according to $s_{original}$. Each line is the comparison between an original fault localization technique and test case purification. For each category, we list the positive, negative, neutral (as in Table 4), and total faults, respectively. We evaluate test case purification with both # Faults (the number of faults) and StmtSave (the average saved effort obtained by applying test case purification). Note that StmtSave may be below zero because applying test case purification may lead to worse results.

Table 5: Wasted effort (measured with the absolute number of statements to be examined before finding the fault). The wasted effort for our dataset is given on all six considered fault-localization techniques with and without test case purification. The last row averages over all subject programs.

Subject program	Original technique						Test case purification					
	Tarantula	SBI†	Ochiai	Jaccard	Ochiai2†	Kulczynski2	Tarantula	SBI	Ochiai	Jaccard	Ochiai2	Kulczynski2
JExel	45.89	45.89	25.15	30.74	30.14	34.83	21.56	35.52	**21.22**	26.90	26.98	26.96
JParsec	47.76	47.76	20.67	22.46	22.46	27.02	**15.96**	18.32	20.37	21.52	21.67	17.64
Jaxen	105.88	105.88	39.02	56.38	56.38	83.46	38.92	52.01	**34.99**	45.39	46.12	70.44
AC Codec	57.04	57.04	48.27	48.53	48.53	56.25	**44.68**	49.06	46.37	47.13	47.47	48.98
AC Lang	25.66	25.66	21.92	21.99	21.99	25.61	21.24	22.12	**20.95**	20.99	21.21	22.07
Joda Time	150.13	150.13	106.35	129.03	129.03	136.53	**71.34**	104.58	101.71	124.67	124.93	100.37
Average	72.06	72.06	43.56	51.52	51.42	60.62	**35.62**	46.93	40.93	47.77	48.06	47.74

† For some cases, the group of Tarantula and SBI (as well as the group of Ochiai and Ochiai2) produce very similar results. Studies in [19], [24] show evidences on their similarity. The spectra are usually different, which are shown in Table 6.

For faults with $s_{original} > 10$, applying test case purification can obtain positive and neutral results with few negative results. Taking Tarantula as an example, the effectiveness of fault localization on 524/687 faults (76.27%) is improved by applying test case purification and worsened for 30 faults (4.37% in column *Negative*). For Ochiai, localization on 178 out of 394 faults (45.18%) is improved. Test case purification can save over 65 statements on average for Tarantula or SBI, 36 statements for Kulczynski2, and over 10 statements for Ochiai, Jaccard, or Ochiai2. In JExel, applying test case purification never leads to negative results. That is, fault localization on all the faults with $s_{original} > 10$ can be improved or unchanged. In both Jaxen and AC Lang, test case purification can lead to non-negative results on five original techniques except Tarantula (one negative result). In Joda Time, between 11 and 15 cases are worsened. Test case purification in Joda Time performs the worst among our six subject programs. A potential reason is that Joda Time consists of over 68 thousand SLoC; this scale probably hinders fault localization.

For faults with $1 < s_{original} \leq 10$, test case purification can also work well. Five out of the six fault localization techniques obtain no more than four negative results; an exception is Tarantula, which obtains 13 negative results. On most of subject programs, applying test case purification can improve the original fault localization, but Tarantula in JParsec as well as Tarantula and Ochiai in Joda Time achieve a little decrease. Note that the result of a fault with $1 < s_{original} \leq 10$ may not have enough space to improve since the faulty statement has been ranked in top-10 statements.

For faults with $s_{original} = 1$, the results are already optimal. A good approach cannot decrease the results for such faults. In our work, only one fault with $s_{original} = 1$ out of 4092 cases in all the subject programs gets a negative result by applying test case purification. In other words, 99.66% of faults keep an optimal rank under test case purification.

For all the techniques in our experiments, five out of six subject programs have less than 10 negative results among 300 faults. Joda Time contributes the most negative results, e.g., 20 negatives for Tarantula. On the other side, applying test case purification to Tarantula improves the most among the six original fault localization techniques.

One major reason for the negative results is that there exists dependency between test cases. For example, if two test cases share a static object and one test case creates the object with a fault (the source code of creating the object contains a faulty statement), then the other test case may fail due to the propagation of the fault. Such propagation makes the second test case fail but the spectrum of the test case does not contain the fault statement. Based on our manual checking, the dependency of test cases is the major reason of negative results. We will further discuss the reasons for negative results in Section 6.2.

Summary. Based on the comparison with six techniques on six subject programs, test case purification can improve original techniques in fault localization. For the hard-localized faults (with initial rankings beyond 10 statements), test case purification saves the effort of examining more than 10 statements in average.

5.3 Computation Time

As shown in Table 6, Tarantula-Purification obtains the best results among all the techniques in comparison. In this section, we present the computation time of our work. Table 7 lists the computation time of Tarantula-Purification on six subject programs. For each subject program, we list the computation time (in seconds) of the original fault localization and the three phases in test case purification.

The whole process of test case purification costs 275.59 seconds on average. The most time-consuming part is the phase of test case slicing. A major reason for the large computation time is that dynamic program slicing is a complex task and requires monitoring the runtime traces [12], [38]. Comparing with the time of original fault localization techniques, i.e., 59.79 seconds, the time of test case purification is still acceptable. We plan to explore further techniques to improve the phase of test case slicing.

Summary. The computation time of test case purification (275 seconds per fault) is acceptable since the whole process can be executed automatically.

6. THREATS TO VALIDITY

We discuss threats to the validity of our results with respect to experiment construction and method construction.

6.1 Experiment Construction

In our work, we evaluate test case purification based on six existing fault localization techniques. Experiments are conducted in six typical open-source subject programs in Java. However, comparing with the large number of existing fault localization techniques, the generality of our work should be further studied.

Table 6: Detailed evaluations on three categories of faults (optimally-localized, easy-localized, hard-localized). Columns *StmtSave* measure the saved effort obtained with test case purification.

Subject program	Technique in comparison	$s_{original} = 1$	$1 < s_{original} \leq 10$					$s_{original} > 10$						Sum					
		Neutral	Positive	Negative	Neutral	Total		Positive		Negative		Neutral	Total		Positive		Negative		Neutral	Total	
		#Faults	#Faults	#Faults	#Faults	#Faults	StmtSave	#Faults	StmtSave	#Faults	StmtSave	#Faults	#Faults	StmtSave	#Faults	StmtSave	#Faults	StmtSave	#Faults	#Faults	StmtSave
Jexel	Tarantula	50	35	0	104	139	0.95	90	79.60	0	0.00	21	111	64.54	125	58.37	0	0.00	175	300	24.32
	SBI	50	30	0	109	139	0.55	81	37.46	0	0.00	30	111	27.33	111	28.02	0	0.00	189	300	10.37
	Ochiai	66	13	0	151	164	0.11	38	30.59	0	0.00	32	70	16.61	51	23.15	0	0.00	249	300	3.94
	Jaccard	66	13	0	151	164	0.08	38	29.96	0	0.00	32	70	16.26	51	22.58	0	0.00	249	300	3.84
	Ochiai2	66	10	0	155	165	0.06	36	26.13	0	0.00	33	69	13.63	46	20.66	0	0.00	254	300	3.17
	Kulczynski2	51	32	0	117	149	0.65	73	31.01	0	0.00	27	100	22.64	105	22.48	0	0.00	195	300	7.87
Jparsec	Tarantula	65	36	3	107	146	-0.09	73	137.60	4	-123.25	12	89	107.33	109	92.90	7	-83.93	184	300	31.80
	SBI	65	29	2	115	146	0.22	70	126.21	2	-18.00	17	89	98.87	99	89.67	4	-11.63	197	300	29.44
	Ochiai	89	34	0	140	174	0.37	11	21.55	5	-42.20	21	37	0.70	45	6.69	5	-42.20	250	300	0.30
	Jaccard	89	28	0	140	168	0.26	15	20.63	5	-13.90	23	43	5.58	43	8.20	5	-13.90	252	300	0.94
	Ochiai2	89	26	0	142	168	0.22	14	18.96	5	-13.30	24	43	4.63	40	7.56	5	-13.30	255	300	0.79
	Kulczynski2	73	29	2	120	151	0.21	56	50.34	2	-18.00	18	76	36.62	85	33.66	4	-11.63	211	300	9.38
Jaxen	Tarantula	34	21	0	75	96	0.86	148	135.29	1	-19.00	21	170	117.67	169	118.97	1	-19.00	130	300	66.96
	SBI	34	19	0	77	96	0.67	149	108.03	0	0.00	21	170	94.68	168	96.19	0	0.00	132	300	53.87
	Ochiai	63	18	2	138	158	0.18	56	21.12	0	0.00	23	79	14.97	74	16.53	2	-6.25	224	300	4.04
	Jaccard	63	13	2	141	156	0.19	59	55.38	0	0.00	22	81	40.34	72	45.85	2	-2.25	226	300	10.99
	Ochiai2	63	8	2	146	156	0.10	57	53.69	0	0.00	24	81	37.78	65	47.38	2	-2.25	233	300	10.25
	Kulczynski2	43	31	0	92	123	0.86	109	34.86	0	0.00	25	134	28.36	140	27.90	0	0.00	160	300	13.02
AC Codec	Tarantula	34	36	0	73	109	1.23	99	40.64	8	-56.44	50	157	22.75	135	30.80	8	-56.44	157	300	12.35
	SBI	34	26	0	83	109	0.83	86	30.77	8	-43.06	63	157	14.66	112	24.44	8	-43.06	180	300	7.98
	Ochiai	42	14	0	136	150	0.13	29	19.53	8	-1.69	71	108	5.12	43	13.62	8	-1.69	249	300	1.91
	Jaccard	42	11	0	139	150	0.11	21	20.02	8	-1.81	79	108	3.76	32	13.64	8	-1.81	260	300	1.41
	Ochiai2	42	11	0	139	150	0.09	18	17.81	8	-1.81	82	108	2.83	29	11.50	8	-1.81	263	300	1.06
	Kulczynski2	34	27	0	86	113	0.81	82	29.77	8	-43.94	63	153	13.66	109	23.23	8	-43.94	183	300	7.27
AC Lang	Tarantula	43	98	6	104	208	0.31	33	38.59	2	-6.50	14	49	25.72	131	10.83	8	-11.63	161	300	4.42
	SBI	43	96	0	112	208	0.66	29	31.88	0	0.00	20	49	18.87	125	8.50	0	0.00	175	300	3.54
	Ochiai	50	92	0	127	219	0.55	10	17.35	0	0.00	21	31	5.60	102	2.87	0	0.00	198	300	0.98
	Jaccard	48	94	0	127	221	0.59	10	16.95	0	0.00	21	31	5.47	104	2.88	0	0.00	196	300	1.00
	Ochiai2	48	89	0	132	221	0.52	10	11.85	0	0.00	21	31	3.82	99	2.35	0	0.00	201	300	0.78
	Kulczynski2	43	96	0	112	208	0.64	29	32.09	0	0.00	20	49	18.99	125	8.50	0	0.00	175	300	3.54
Joda Time	Tarantula	67 †	29	4	88	121	-0.14	81	307.67	15	-92.37	15	111	212.04	110	227.13	20	-75.30	170	300	78.79
	SBI	68	27	1	93	121	0.33	80	172.97	11	-27.59	20	111	121.93	107	129.71	12	-25.42	181	300	45.55
	Ochiai	81	24	2	124	150	-0.19	34	50.96	11	-29.14	24	69	20.46	58	30.40	13	-29.23	229	300	4.64
	Jaccard	81	25	1	124	150	0.21	33	52.05	12	-37.63	24	69	18.35	58	30.17	13	-34.85	229	300	4.35
	Ochiai2	81	22	1	127	150	0.14	29	56.95	12	-37.54	28	69	17.41	51	32.82	13	-34.77	236	300	4.10
	Kulczynski2	70	27	1	95	123	0.33	75	149.30	11	-41.86	21	107	100.35	102	110.19	12	-38.50	186	300	36.16
All	Tarantula	293	255	13	551	819	0.47	524	128.72	30	-78.73	133	687	94.74	779	87.41	44	-60.39	977	1800	36.44
	SBI	294	227	3	589	819	0.54	495	91.66	21	-32.57	171	687	65.05	722	63.47	24	-29.00	1054	1800	25.12
	Ochiai	391	195	4	816	1015	0.22	178	28.40	24	-22.71	192	394	11.45	373	14.33	28	-22.04	1399	1800	2.63
	Jaccard	389	184	3	822	1009	0.26	176	39.90	25	-21.42	201	402	16.14	360	20.25	28	-19.34	1412	1800	3.75
	Ochiai2	389	166	3	841	1010	0.21	164	38.76	25	-21.26	212	401	14.53	330	19.92	28	-19.20	1442	1800	3.36
	Kulczynski2	314	242	3	622	867	0.57	424	55.31	21	-40.38	174	619	36.52	666	35.98	24	-35.83	1110	1800	12.87

† Only one fault with $s_{original}$ has a negative result. That is test case purification makes the original result worse on one fault in Joda Time when comparing with Tarantula.

To conduct large-scale experiments, we employ mutation testing techniques to inject faults in subject programs. We use six widely-used mutant operators to generate all faulty versions of the subject programs under consideration. Then we randomly select 300 faulty versions as the final faults. A potential threat is that the type of mutant operators may impact the effectiveness of fault localization. For example, a fault localization technique may be good at handling a specific type of faults. We have not checked the results for this issue. We leave it as one of our future work.

Table 7: Time of Tarantula with test case purification (in seconds)

Subject program	Original fault localization	Test case atomization	Test case slicing	Rank refinement	Total
JExel	5.70	6.54	369.89	4.45	386.58
JParsec	8.11	4.21	137.03	5.91	155.27
Jaxen	14.08	6.38	204.71	6.49	231.67
AC Codec	128.88	41.38	105.60	4.20	280.06
AC Lang	33.31	18.59	79.27	9.69	140.86
Joda Time	168.63	18.06	254.60	17.82	459.11
Average	59.78	15.86	191.85	8.09	275.59

6.2 Method Construction

In Section 3.4, we propose a rank refinement method to leverage the spectra of purified test cases to improve an original fault localization technique. Our method is a simple formula to combine spectra of test case purification and the original fault localization. Other formulae can be used for the combination, e.g., the average and the weighted average. We plan to design new methods to make better use of the spectra of test case purification in the future.

In our work, we generate purified test cases to improve fault localization. Based on the spectra of purified test cases, we rank frequent statements in such spectra prior to other statements. Our experiments show that test case purification can obtain non-negative results on most faults. A potential assumption is that test cases are executed independently. That is, the results of a test case should not impact the results of other test cases. This assumption can be satisfied since most of test cases are well-designed. As mentioned in Section 5.2, sometimes test cases suffer from dependencies. This is a challenge topic in fault localization since there is no explicit relationship between the failing test case and the faulty statement.

We use JavaSlicer [12] as the implementation tool in test case slicing. As mentioned by the authors of JavaSlicer, this tool has some known limitations. For example, traces of native methods and Java standard library classes may be missed. To our knowledge, JavaSlicer is the most easy-to-use slicing tool for Java 1.6. In our implementation of test case slicing, we write a program to check potential missing statements by JavaSlicer, but the implementation may still miss some statements.

7. RELATED WORK

To our knowledge, this paper is the first work to directly manipulate test cases to improve fault localization. We list the related work as follows.

7.1 Fault Localization Techniques

Fault localization aims to localize the faulty position in programs. Tarantula by Jones et al. [16] is an integrated framework to localize and visualize faults. Empirical evaluations of Tarantula on fault localization can be found in [15]. Abreu et al. [1] propose Ochiai and Jaccard for fault localization. All of Tarantula, Ochiai, and Jaccard can be viewed as the state-of-art in spectrum-based fault localization. Naish et al. [24] propose a family of fault localization methods and empirically evaluate their results. Recent work by Zhang et al. [39] addresses the problem of how to identify faults with only failed runs. Xie et al. [32] propose a theoretical analysis on multiple ranking metrics of fault localization and divide these metrics into categories according to their effectiveness.

Santelices et al. [29] combine multiple types of code coverage to find out the faulty positions in program. Baah et al. [3] employ potential outcome model to find out the dynamic program dependencies for fault localization. Xu et al. [34] develop a noise-reduction framework for localizing Java faults. This work is a general framework that can be used to improve multiple existing fault localization techniques. DiGiuseppe & Jones [9] recently propose a semantic fault diagnosis approach, which employs natural language processing to detect the fault locations. Xuan & Monperrus [35] develop a learning-based approach to combining multiple ranking metrics for fault localizing. Steimann et al. [31] discuss the threats to validity in the empirical assessments of fault localization. Their work also presents the theoretical bounds of the accuracy in fault localization.

Hao et al. [13] propose a test-input reduction approach to reduce the cost of inspecting the test results. Gong et al. [11] design a diversity-maximization-speedup approach to reduce the manual labeling of test cases and improve the accuracy of fault localization. Yoo et al. [36] address the problem of fault localization prioritization. Their work investigates how to rank remaining test cases to maximize fault localization once a previous fault is found.

Baudry et al. [4] leverage the concept of dynamic basic blocks to maximize the ability of diagnosing faults with a test suite. Artzi et al. [2] directly generate test cases for localizing faults in invalid Html programs in dynamic web applications. This work does not require the test oracles since a web browser can report the crashes once invalid Html programs are found. Fault localization is also used as a phase of predicting a candidate position of the patch in software repair, such as GenProg [18] and Nopol [8].

In our work, we address the same problem statement of fault localization. In contrast to existing work, test case purification is a framework to make better use of existing test cases. Our approach directly operates on test cases and can be generally applied to most of existing fault localization techniques.

7.2 Mutation and Slicing Based Fault Localization

Mutation-based fault localization has been recently proposed. The kernel idea of mutation-based fault localization is to localize faults by injecting faults. Zhang et al. [37] propose FIFL, a fault injecting approach to localizing faulty edits in evolving Java programs. Candidate edits are ranked based on the suspiciousness of mutants. Papadakis & Le Traon [26] develop Metallaxis-FL, a mutation-based technique for fault localization on C programs. Their work shows that test cases that are able to kill mutants can enable accurate fault localization. Moon et al. [23] recently propose MUSE, an approach based on both mutants of faulty statements and mutants of correct statements.

Slicing-based fault localization leverages program slicing to remove the statements in programs to find out the final faulty statements. Zhang et al. [38] employ dynamic slicing to reduce the size of C programs to avoid the distribution by irrelevant statements. Mao et al. [20] combine both statistic slicing and dynamic slicing to identify the faulty statements in programs. They empirically evaluate the slicing-based techniques on multiple fault localization techniques. Xie et al. [33] propose a new concept of metamorphic slice, based on the integration of metamorphic testing and program slicing. Metamorphic slices localize faults without the requirement of test oracles.

Existing work on mutation-based and slicing-based fault localization aims to change the subject program to identify the faulty parts in the program. In our work, test case purification changes test cases for fault localization rather than subject programs. We make better use of existing test cases (test oracles) to improve the effectiveness of fault localization.

8. CONCLUSION

In this paper, we propose a test case purification approach for improving fault localization. Our work directly manipulates test cases to make better use of existing test oracles. We generate small fractions of test cases, that we call purified test cases, to collect discriminating spectra for all assertions in the test suite under consideration. Our experimental results show that test case purification can effectively improve original fault localization techniques. Only a small fraction of faults (1.3 to 2.4%) suffer from worsened results. The results show that the benefits of test case purification exist on six fault localization techniques.

As future work, we plan to conduct experiments on other Java projects to further investigate the performance of our work. We plan to design new ranking methods to combine with the spectra of test case purification. Moreover, we want to explore how to reduce the time cost of test case slicing to facilitate test case purification. We plan to check the applicability of the idea of test case purification for other software problems, e.g., regression testing [28] or automatic software repair [22].

9. ACKNOWLEDGMENTS

Our work is built on the top of open-source libraries. We thank Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez (for Spoon), Alexandre Perez, José Carlos Campos, and Rui Abreu (for GZoltar), Clemens Hammacher, Martin Burger, Valentin Dallmeier, and Andreas Zeller (for JavaSlicer) for their contributions.

This research is done with support from EU Project Diversify FP7-ICT-2011-9 #600654 and an INRIA Postdoctoral Research Fellowship.

10. 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*, pages 89–98. IEEE, 2007.

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60. ACM, 2010.

[3] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 146–156. ACM, 2011.

[4] 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.

[5] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[6] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 257–267. IEEE, 2013.

[7] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381. ACM, 2012.

[8] F. DeMarco, J. Xuan, D. L. 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.

[9] N. DiGiuseppe and J. A. Jones. Semantic fault diagnosis: automatic natural-language fault descriptions. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 23. ACM, 2012.

[10] E. Gamma and K. Beck. Junit. org. *URL: http://www. junit. org*, 2005.

[11] L. Gong, D. Lo, L. Jiang, and H. Zhang. Diversity maximization speedup for fault localization. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 30–39. IEEE, 2012.

[12] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 49–55. IEEE Computer Society, 2009.

[13] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei. Test input reduction for result inspection to facilitate fault localization. *Automated software engineering*, 17(1):5–31, 2010.

[14] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[15] J. A. Jones and M. J. 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. ACM, 2005.

[16] 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.

[17] T.-D. B. Le and D. Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 310–319. IEEE, 2013.

[18] 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.

[19] B. Liblit, A. Aiken, M. Naik, and A. X. Zheng. Scalable statistical bug isolation. In *In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[20] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[21] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[22] 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 International Conference on Software Engineering*, pages 234–242, 2014.

[23] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 153–162. IEEE, 2014.

[24] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[25] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 562–571. IEEE, 2013.

[26] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 2013.

[27] R. Pawlak, C. Noguera, and N. . Petitprez. Spoon: Program Analysis and Transformation in Java. Rapport de recherche RR-5901, INRIA, 2006.

[28] 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.

[29] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *IEEE 31st International Conference on Software Engineering, 2009*, pages 56–66. IEEE, 2009.

[30] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130. IEEE, 2012.

[31] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.

[32] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[33] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[34] J. Xu, Z. Zhang, W. Chan, T. Tse, and S. Li. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, 55(5):880–896, 2013.

[35] J. Xuan, M. Monperrus, et al. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. IEEE, 2014.

[36] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19, 2013.

[37] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 765–784. ACM, 2013.

[38] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.

[39] Z. Zhang, W. K. Chan, and T. Tse. Fault localization based only on failed runs. *IEEE Computer*, 45(6):64–71, 2012.

[40] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.