Revisiting Test Cases to Boost Generate-and-Validate Program Repair

Jingtang Zhang*, Kui Liu†‡, Dongsun Kim§, Li Li*, Zhe Liu*, Jacques Klein|| Tegawendé F. Bissyandé||
*Nanjing University of Aeronautics and Astronautics, Nanjing, China
†Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Nanjing, China
‡State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
§Kyungpook National University, Daegu, South Korea, darkrs@knu.ac.kr
||Monash University, Melbourne, Australia, li.li@monash.edu

Index Terms—Automated program repair, test case, fault localization.

Abstract—Fault localization produces bug positions as the basic input for many automated program repair (APR) systems. Given that test cases are the common means that automatic fault localization techniques leverage, we investigate the impact of their characteristics (in terms of quality and quantity) on APR. In particular, we analyze the statements that appear in crash stack traces when test cases fail (note that stack traces are available when an ordinary test case fails since its verdict is often made by assertions that produce errors such as AssertError in Java and JUnit), and explore the possibility of using some relevant crash information to enhance fault localization; this ultimately improves the effectiveness of APR tools. Our study reveals that the considered state-of-the-art APR systems achieve the best performance when fixing bugs associated with boolean type expected values (e.g., assertTrue() or assertFalse()). In contrast, they achieve their worst performance when addressing bugs related to null check assertions. Meanwhile, null check bugs as well as the bugs associated with boolean and string type expected values are still the main challenge that should be addressed by the future APR. For exception throwing bugs, existing APR systems present the best performance on fixing NullPointerException bugs, while the tough task of them is to resolve the bugs throwing developer-defined exceptions. The information in stack traces after executing the bug-triggering test cases can be used to effectively improve the performance on fault localization and program repair.

Unfortunately, while testing is now largely automated for identifying defects, fixing programs remains challenging as it resource-intensive w.r.t.time and manual effort [5]. Therefore, the promise of automated program repair (APR) to alleviate the manual burden is appealing [6]. APR has thus been a prolific research field in the last decade [7], [8], [9]. Among the various approaches that are proposed, many fall under the category of generate-and-validate APR [10], [11], [12], [13], [14], [15], [16], [17] where program faults are localized to drive patch generation before patch validation. Figure 1 presents the common steps of APR, in which fault localization tries to spot the faulty statements that should be changed by APR tools [18]. In most APR tools, fault localization is implemented with spectrum-based techniques that highly rely on test case execution coverage [19].

Test cases are important for APR. In the patch generation process, they have been largely exploited to improve repair performance. For example, ACS [20] leverages the expected return values of the failed tests as constraints to synthesize patches. Xin and Reiss [21] also explored statements appearing in crash stack traces for improving program repair. Patch validation further relies on the regression tests to check whether the patch can make the patched buggy program pass all tests or not. Test cases have been further exploited to validate the correctness of APR-generated patches by investigating the behavior similarity of test case executions [22].

I. INTRODUCTION

With increases in scale and complexity, the software is prone to defects. Such defects, however, can incur huge losses [1], [2], [3]. To identify software defects, developers often rely on software testing campaigns, where each software functionality is executed to (1) assess that the software behavior matches expected requirements and (2) ensure that the software is defect-free [4]. In general, developers specify some inputs and their corresponding expected outputs as test cases, which will form the test suites be executed in a testing campaign. When software is adequately tested, it provides more guarantees for reliability, security, and high performance [4].

*Kui Liu and Zhe Liu are the corresponding authors.
insights into the test cases that are leveraged to perform fault localization, nor does it attempt to enhance the fault localization results towards improving program repair.

This paper. Our work aims to revisit test cases (and their execution output) to improve generate-and-validate program repair through enhancing fault localization. After investigating the relationship between the fault localization performance and the bug-fixing performance of state-of-the-art APR systems, we provide a characterization (both qualitative and quantitative) of test cases with respect to fault localization. We then propose to leverage information on statements listed in crash stack traces to improve localization results and eventually boost the APR performance.

We make the following findings in our study:

1) APR tools in the literature are still challenged to repair bugs that cannot be precisely localized by spectrum-based fault localization.
2) From a quantitative perspective, the number of failing test cases does not seem to affect the performance of APR.
3) From a qualitative perspective, test case characteristics correlate with the performance of APR: the state-of-the-art APR presents the best performance on fixing bugs associated with expected output values of boolean, but achieves the worst performance on fixing non-null asserting bugs. Meanwhile, non-null asserting bugs, as well as the bugs associated with boolean and string type expected values are still the main challenge that should be addressed by the future APR.
4) Among exception-related bugs, existing APR systems presented the best performance on fixing NullPointer Exception bugs. Resolving developer-defined exceptions remains however the most challenging.
5) The information in stack traces produced by bug-triggering test cases can be used to effectively improve the performance of fault localization (and eventually of program repair). Furthermore, the information in stack traces can be leveraged to guide the selection of fix patterns towards producing more correct patches.

Note that, when using JUnit, stack traces are available for any ordinary failing test case as developers often check test verdict by using assertions (e.g., `assertTrue()`), which produce errors with stack traces if the test condition is not satisfied.
patch could just overfit the test suite [33], but does not fix the bug as correctly as developers expected. To address this issue, practitioners have been exploring to validate the correctness of APR-generated patches [38], [39], [40], [41], [42]. For example, Xiong et al. [22] analyzed the behavior similarity of test case executions to determine patch correctness.

III. STUDY DESIGN

This section first overviews the research questions that we investigate in this work. Then, we present the experimental setup to answer these research questions.

A. Research Questions

• RQ1: Do the state-of-the-art APR systems only focus on addressing the bugs that are easily detected by fault localization techniques? Spectrum-based fault localization frameworks (e.g., GZoltar [19]) provide a ranked list of suspicious statements. Our research question aims at investigating whether state-of-the-art APR systems are prone to correctly fix such bugs that current localization techniques can readily localize (i.e., they rank buggy statements in top positions).

• RQ2: Do the characteristics of test cases impact the bug-fixing performance of state-of-the-art APR systems? Test cases play an important role in the process of fault localization as well as patch validation, and can be used in patch generation since test cases can be used to categorize bugs [9]. In previous studies, a few APR systems have been proposed for addressing the specific types of bugs (e.g., NPEFix [43] and VFix [44] for null pointer dereferences). Our research question checks out whether the state-of-the-art APR systems focus on specific types of bugs or intentionally address common bugs.

• RQ3: To what extent stack trace information (when test cases lead to program failures) can be leveraged to improve the bug-fixing effectiveness of APR systems? The failing execution of some test cases can lead to crashes with exceptions being thrown, where crashed statements will be enumerated in the corresponding stack trace. When developers fix such bugs manually, they will firstly attempt those crashed statements before others. Therefore, we explore the possibility of improving the bug-fixing performance of APR systems with the stack trace information from failed executions of test cases.

B. Experimental Setup

In this study, we focus on the APR systems targeting Java program bugs. To answer the aforementioned research questions, we select the benchmark Defects4J [45] as it contains test cases for buggy Java programs with the associated developer patches. Table I presents statistics on the number of bugs and test cases available in version 1.5.0 of Defects4J that we use in this paper, since this version has been widely used by state-of-the-art APR systems targeting Java programs [46].

For fault localization, we consider GZoltar with Ochiai as the ranking formula to prioritize the bug positions in Defects4J. GZoltar [19] is an on-hand test automation framework for automatic debugging, and has been widely used in the APR community [20], [47], [48], [49] as well. And the majority of the APR systems [30], [36], [50], [29] considered Ochiai as the ranking formula to prioritize the suspicious statements.

Table II presents the corresponding research questions that are considered in this study.

IV. STUDY RESULTS

This section provides experiment results as well as the key findings of the corresponding research questions that are investigated in this work.

A. RQ1: Quantitative Bug-Fixing Performance with Fault Localization and Failing Tests

To answer this question, we first review the APR tools listed in Table II and collect the bugs that are correctly/plausibly fixed by them (shown in Table III). The correctness and plausibility of fixed bugs are directly excerpted from the reported results of the related research work presented in [18]. Among the 391 bugs in the Defects4J benchmark, 214
bugs are not fixed by any APR tools (unfixed bugs), 53 bugs are fixed by at least one APR tool with plausible but incorrect patches only (plausibly fixed bugs), 78 bugs are fixed with correct patches by some APR tools and plausible but incorrect patches by other APR tools (plausibly or correctly fixed bugs), and 46 bugs are fixed with patches that were all correct (correctly fixed bugs), denoted as c-fixed bugs: none of the tools generated a plausible but incorrect patch for them.

<table>
<thead>
<tr>
<th>Category</th>
<th># bugs</th>
<th># bugs’</th>
</tr>
</thead>
<tbody>
<tr>
<td>unfixed bugs</td>
<td>214</td>
<td>245</td>
</tr>
<tr>
<td>only plausibly-fixed bugs</td>
<td>53</td>
<td>41</td>
</tr>
<tr>
<td>plausibly/correctly-fixed bugs</td>
<td>78</td>
<td>42</td>
</tr>
<tr>
<td>only correctly-fixed bugs</td>
<td>46</td>
<td>63</td>
</tr>
</tbody>
</table>

**“# bugs”** denotes the bugs that are fixed by the APR systems, studied after the investigation by Liu et al. [18] in which the impact of fault localization on program repair has been explored.

As reported by Liu et al. [18], the fault localization performance can impact the bug-fixing performance of APR tools, which are prone to fix the subset of Defects4J bugs that can be accurately localized. We further investigate whether the bug-fixing performance of new proposed APR tools has been improved after Liu et al.’s finding is reported. To that end, we first leverage GZoltar + Ochiai to expose the bug positions related to Defects4J bugs within a ranked list of suspicious statements. We note that the majority of the ranked suspicious statements are actually not faulty. Concretely, we define the index of the actual faulty statement spotted in the ranked list of suspicious statements as the spotted bug location. For example, the bug Chart-1 shown in Figure 2 is located at line 1,797 in the class org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java. Its spotted bug location is 28 (highlighted in red) exposed by GZoltar + Ochiai, as shown in Figure 3.

Figure 4 illustrates the distribution of spotted bug locations concerning the bug-fixing results of the state-of-the-art APR tools. We observe that for all state-of-the-art APR tools listed in Table II, the unfixed bugs and the plausibly but incorrectly fixed bugs have lower spotted bug locations (i.e., bigger values) than the correctly fixed ones. When only considering the APR tools proposed after Liu et al.’s “fault localization bias” work [18], the spotted bug locations of fixed and unfixed bugs have a similar distribution with all APR tools. This indicates that the newly proposed state-of-the-art APR tools

![Fig. 2. Patch diff for fixing bug Chart-1.](image1)

![Fig. 3. Ranked list of suspicious statements for exposing bug Chart-1.](image2)

![Fig. 4. Distribution of spotted bug locations for fixed/unfixed bugs (the smaller the better).](image3)

still face the challenge on effectively resolving the bugs which cannot be precisely localized. To address this challenge, two ways could be explored:

1) Improving the fault localization performance for APR. APR tools make trials on modifying the suspicious statements reported by fault localization techniques. The straightforward way is to improve the precision of locating bug positions by ranking high the faulty statements.

2) Improving the patch validation for APR. Many bugs are fixed with plausible but incorrect patches that are generated by modifying the non-faulty statements, which are, unfortunately, ranked before the faulty statement. Thus, to boost APR, an alternative or complement to the first
B. RQ2: Dissection of Failing Test Cases

When the test cases of buggy programs failed to be executed, we observe that the failing behavior can be summarized into two main categories:

1) **Unexpected value**: The actual executed results do not satisfy the expected results of programs specified in test cases. For example, the bug *Chart-1* shown in Figure 5, its assertion code expects that the returned value of `lic.getItemCount()` should be 1 (cf. the code at line 409 highlighted with red background), while the actual returned value is 0 after executing the test case (cf. the information presented in the excerpted stack trace). So the executed result does not satisfy the expected result, which leads to the failing test.

2) **Throwing exception**: Throwing exceptions when the failing test cases are executed. For example, after executing the test cases of bug *Chart-4*, it throws a `java.lang.NullPointerException`, shown in Figure 6.

As shown in Figure 7, the 391 Defects4J bugs can be grouped into three categories with their failing behavior: 70% of them present the unexpected values, 25% of them throw exceptions, and the remaining 5% bugs return the unexpected values as well as throw exceptions. As presented in Figure 8, from a quantitative perspective (i.e., but looking at the raw numbers of fixed bugs only), most correctly fixed bugs (68% ≈ \( \frac{331}{489} \)) belong to the bugs with unexpected values. While if we consider the ratio of bugs, it seems that the correctly fixed bugs are even distributed in the three categories of bugs (i.e., \{68%, 30%, 2%\} → \{70%, 25%, 5%\}).

We further dissect the bugs associated with the unexpected values and throwing exceptions with the data types of expected values and the exception types, respectively. The related results are presented in Figure 9 and Figure 10. Among the bugs with unexpected values, the state-of-the-art APR systems present
For the bugs throwing exceptions, the existing APR systems achieve the best performance on fixing `NullPointerException` exception. This result can be explained by the existence of 1) APR tools specifically designed to address such exception type (e.g., NPEFix [43] and VFix [44]) and 2) specific targeted fix patterns [31, 48] for `NullPointerException` bugs. The bugs concerning `Self-defined` exceptions (i.e., exceptions specifically defined by developers for the programs) are not easily resolved by existing APR systems. The following toughly resolved bugs are related to the `IllegalStateException` and `IndexOutOfBoundsException`. Furthermore, the bugs about `NoSuchMethodError`, `UnsupportedOperationException`, `NotSerializableException`, `ArrayStoreException`, `ClassNotFoundException`, `NullPointerException`, and `StackOverflowError` cannot be fixed by any state-of-the-art APR systems. For throwing exception bugs, the existing APR systems presented the best performance on fixing `NullPointerException` bugs, and the tough task of them is to resolve the bugs throwing developer-defined exceptions.

C. RQ3: Stack Trace Information from Failing Tests

In the spectrum-based fault localization process of generate-and-validate program repair, the passing/failing test cases are used to calculate the suspiciousness values for ranking suspicious statements to spot bug positions (cf. Formula 1 Section II). APR systems will revise each suspicious statement in the ranked list one by one to generate patch candidates until one valid patch (that makes the patched program pass all tests [46]) is found. It is somehow different from the manually debugging process, especially for those bugs with the crashed statements in stack trace after executing test cases.

In practice, when a bug arises with the crashed statements in a stack trace (e.g., the crashed statements of bug Lang-6 shown in Figure 11), developers are prone to address these crashed statements than others. In the APR literature, only ssFix [21] leverages those crashed statements to purify the exposed bug positions. To the best of our knowledge, we are the first to investigate to what extent the crashed statements after executing the failing test cases can impact the fault localization and bug-fixing performance of an APR system.

Regarding the information contained in the stack trace, the first line indicates the failing executed test case (e.g., `org.apache.commons.lang3.StringUtilsTest::testEscapeSurrogatePairs` in Figure 11). The second line indicates the concrete error which causes the program to terminate abnormally (e.g., exception or assertion failure), which may provide a detailed specification of the exception, such as the `java.lang.StringIndexOutOfBoundsException` with the potential reason “String index out of range: 2” shown in Figure 11. Then, the remaining contents describe the full stack trace with the crashed statements that are presented with the related classes as well as their line numbers. These crashed statements are presented reversely in terms of their execution order. For some cases, the latest...
executed statements are not from the buggy program but, e.g., from APIs. For example, in Figure 11, the first two crashed statements (i.e., lines 03 and 04) are the last executed two statements of the failing test case, but they come from the Java framework API and not from the buggy program Lang-6. Therefore, in the manual debugging way, developers will make trials on the remaining crashed statements one by one until the bug is fixed. For bug Lang-6, the line-95 statement in the class org.apache.commons.lang3.text.translate.CharSequenceTranslator highlighted in red at the line-05 in Figure 11 will be first addressed by manual debugging. Indeed, the line-95 statement is the buggy statement of Lang-6 (cf. the developer’s patch of fixing Lang-6 shown in Figure 12). However, GZoltar+Ochiai fails to expose the bug position of Lang-6.

After manual review, we observe that all Defects4J bugs (391 bugs in version 1.5.0 and 835 bugs in the latest version 2.0.0) will print the stack trace information after executing their test cases. With the observation, we infer that the stack trace information can be used to improve the performance of fault localization and generate-and-validate program repair. With this hypothesis, we propose two straightforward heuristic principles to rerank the suspicious statement list of spectrum-based fault localization, and feed 4 APR tools (AVATAR, TBar, SimFix, ACS) with the reranked suspicious statements.

1) **Principle 1: prioritize the source code statements in a stack trace.** This principle straightforwardly considers the source code statements in stack trace have a higher suspiciousness value than other statements. The stack trace contains four kinds of statements: (1) the statements from the dependent external libraries (e.g., the third-party libraries for the building or testing framework) of the buggy program, (2) the statements in the source code of Java Development Kit (JDK), (3) the statements in the test cases of the buggy program, and (4) the statements in the source code of the buggy program. APR tools aim to solve the bugs in source code. So we consider prioritizing the fourth kind of statements located in the source code of the buggy program and the other three kinds of statements will be discarded. For example, in Figure 11, the statements presented in Lines 05 to 08 will be considered while others will be discarded.

2) **Principle 2: prioritize the code methods and classes targeted by test cases.** This principle aims to figure out the source code range that is targeted by test cases. In practice, for the convenience of maintenance and the high readability of program code, developers often write test code for their programs following a canonical naming convention that names the test classes and test methods with their targeting class and method names (e.g., Test\*\* or ***Test). According to this naming convention, we suppose that the failing test cases are always associated with the related source code. Therefore, we propose that the statements in the scope of the methods and classes tested by the failing test cases have a higher possibility to be faulty than other statements. For example, the failing test case of bug Time-7 is org.joda.time.format.TestDateTimeFormatter::testParseInto_monthDay_feb29_newYork_startOfYear (shown in Figure 13). The bug

---

Fig. 11. Excerpted stack trace after executing the test cases of bug Lang-6.

Fig. 12. Developer’s patch for fixing bug Lang-6.
position indeed is located in the method `parseInto` of the class `org.joda.time.format.DateTimeFormatter`, as shown in Figure 14.

With the two principles, we propose a straightforward algorithm (i.e., Algorithm 1) to rerank the list of suspicious statements exposed by GZoltar + Ochiai. The source code statements of a buggy program in the stack trace are assigned with the highest priority among all suspicious statements (cf. line 9 and line 20 in Algorithm 1). The failing test cases are always designed for validating the functionality of the related code classes concerning the associated failing test classes. Thus, the suspicious statements belonging to failing-test-related methods are assigned with a second higher priority than other suspicious statements (cf. lines 14 and 20). The testing classes of failing test cases have a wider range than the corresponding test cases on exposing faulty positions, so the suspicious statements in the source code classes concerning the associated failing test classes are assigned with lower priority than the failing test cases (cf. lines 17 and 20). Finally, the remaining suspicious statements are ranked with the original order in the lowest priority (cf. lines 8, 18 and 20).

1) Fault Localization: In this experiment, all 835 bugs of Defects4J version 2.0.0 are considered. The fault localization is implemented with the spectrum-based fault localization technique (i.e., GZoltar + Ochiai) widely used in the APR community. Figure 15 presents the fault localization results refined with the aforementioned two principles comparing against the spectrum-based fault localization. Overall, it explicitly shows that fault localization performance has been improved by ranking the faulty statements in higher ranked locations with our proposed two principles than the original spectrum-based fault localization technique. More specifically, as presented in Figure 15, after considering the two principles for fault localization, the 1st quartile, middle, mean, and 3rd quartile values of spotted bug locations are improved with 1, 10, 20, and 30, respectively. Furthermore, 16 bugs are newly exposed with the two principles that cannot be localized before. To sum up, our proposed two straightforward principles can effectively improve the fault localization performance for the spectrum-based fault localization technique.

```
Algorithm 1: Reranking the suspicious statements.
Input : $L$, a list of suspicious statements for a buggy program.
Input : $S_{p1}$, a sorted list of statements with principle 1.
Input : $M_{p2}$, a set of methods with principle 2.
Input : $C_{p2}$, a set of classes with principle 2.
Output : $L_r$, the reranked list of suspicious statements.

1 Function rerank()
2     /* Initialize the suspicious lines $L_c$ in $C_{p2}$. */
3     $L_c := \emptyset$;
4     /* Initialize the suspicious lines $L_m$ in $M_{p2}$. */
5     $L_m := \emptyset$;
6     foreach $l \in L$ do
7         if $l \in S_{p1}$ then
8             $L$.remove($l$);
9                 Continue;
10     endforeach
11     foreach $C \in C_{p2}$ do
12         if $C \in C$ then
13             foreach $M \in M_{p2}$ do
14                 if $l \in L_m$ then
15                     $L_m$.remove($l$);
16                     Break;
17                 else
18                     if $l \in L_c$ then
19                         $L_c$.add($l$);
20                         Break;
21                     else
22                         $L_r$.addAll($S_{p1}$).addAll($L_m$).addAll($L_c$).addAll($L$);
23             Return $L_r$;
```

2) Bug Fixing Performance: We further assess whether our proposed two principles can be improved to use the bug-fixing performance of generate-and-validate APR tools. To that end, we apply the improved bug-spotting results to four state-of-the-art APR tools (namely AVATAR, TBar, SimFix, and ACS) to re-evaluate their bug-fixing performance. Other APR tools are not considered as we failed to re-execute them because of several reasons (i.e., unavailable, unconfigurable, specifically fault localization technique, and specific settings [18], [46]).

As presented in Table V, with our proposed principles for fault localization, all of the 26 bugs, that are correctly fixed by AVATAR with the normal fault localization (normal FL, i.e., GZoltar + Ochiai), still can be correctly fixed by AVATAR with our proposed two principles. And AVATAR can (correctly) fix 6 more bugs, as they are correctly located by our proposed principles but are failed to be located by normal FL. AVATAR also correctly fixes 2 bugs plausibly fixed before. TBar correctly fixes 35 bugs, 11 of them are newly fixed by it, 2 of them are plausibly fixed before. And TBar avoids plausibly fixing 10 bugs. SimFix correctly fixes 3 previously unfixed bugs and avoids generating plausible patches for 4 bugs, but it fails to fix one previously correctly fixed bug. ACS successfully reproduces all previously correctness fixed patches without fixing any new bugs, and avoids generating plausible patches for 6 bugs. To sum up, the correct ratios of patches generated by the four APR tools are improved when they are fed with our proposed principles but are failed to be located by normal FL, i.e., GZoltar + Ochiai.

### Table V

<table>
<thead>
<tr>
<th>APR tools</th>
<th># fixed bugs</th>
<th>CR (%)</th>
<th># fixed bugs</th>
<th>CR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVATAR</td>
<td>26/82</td>
<td>31.7</td>
<td>(26+2+6)/(82+0+8)</td>
<td>+6.1</td>
</tr>
<tr>
<td>TBar</td>
<td>22/50</td>
<td>44.0</td>
<td>(22+2+11)/(50+10+12)</td>
<td>+23.3</td>
</tr>
<tr>
<td>SimFix</td>
<td>17/27</td>
<td>63.0</td>
<td>(17+1+3)/(27+5+3)</td>
<td>+13.0</td>
</tr>
<tr>
<td>ACS</td>
<td>10/20</td>
<td>50.0</td>
<td>(10+0+0)/(20+6+0)</td>
<td>+21.4</td>
</tr>
</tbody>
</table>

The data in the 4th column are presented in the format \((x+y+z)/(X+Y+Z)\), where \(x\) represents the number of bugs (correctly) fixed by the APR tool, \(y\) represents the number of bugs plausibly fixed by the APR tool, and \(z\) represents the number of bugs plausibly fixed by the APR tool with GZoltar + Ochiai.

Looking at the efficiency of fixing bugs in terms of the number of generated patch candidates [46], shown in Figure 16, the efficiency of AVATAR, TBar, and SimFix are dramatically improved by generating fewer patch candidates for fixing bugs with our proposed principles than the normal fault localization, since fewer non-faulty statements will be mutated by them to generate the nonsensical/plausible patch candidates. Fewer patch candidates will spend less source (e.g., time) for compiling and testing the patched programs. We also observe that the efficiency of ACS is improved with a few increases as ACS produces far fewer candidate patches than other tools, so the improvement is not obvious; and most of the correctly fixed bugs are precisely localized with the normal fault localization. Overall, these results indicate that our proposed two straightforward principles can be used to improve the bug fixing performance of the generate-and-validate program repair tools by reducing the trials on non-faulty statements.

In the APR community, some APR tools (e.g., NPEFix [43] and VFix [44] and fix patterns [31], [48] have been specifically proposed for null pointer exception bugs. Indeed, some bugs can be emerged by throwing null pointer exceptions or other exceptions (cf. Section IV-B and Figure 10). We thus explore whether the stack trace information can be used to match fix patterns in pattern-based APR. To this end, we preferentially match the null pointer related fix patterns over other fix patterns of AVATAR for the 12 null pointer exceptions bugs that are fixed by AVATAR. The experimental results show that, according to matching the null pointer related fix patterns for the bugs throwing null pointer exception, the numbers of generated patch candidates for fixing each of the 12 bugs are decreased. Overall, the average number of generated patch candidates is decreased by 746. The efficiency of fixing the null pointer bugs is improved effectively by matching the related fix patterns with the throwing exceptions in the stack trace.

**The information in stack trace after executing the bug-triggering test cases can be used to effectively improve the performance on fault localization and program repair. And the information in stack trace shows the potential of matching adequate fix patterns for bugs.**

### V. Threats to Validity

One of the threats to external validity is the target language of bugs. Only Java bugs are considered in this study. Although the format of test cases and stack traces are different in other languages, we believe that the conclusion and methodology can be applied to other languages since the function of test cases and information in stack traces are similar among different languages. Another external threat to validity is that only one Java bug dataset, namely Defects4J, is considered. However, our study mainly focuses on test cases and stack
traces, which are common elements among Java projects independent from a specific dataset. The other external threat to validity is that we only consider four APR tools, namely AVATAR, TBar, SimFix and ACS, to evaluate the improvement of fault localization for APR tools. Since our optimization principles are directly applied to the fault localization result of the GZoltar/Ochiai framework, instead of APR tools, the improvement for fault localization given by this study can benefit other APR tools as well.

The internal threat is that more semantic information of test cases should be explored. In this study, we only consider the number and type of test cases, ignoring the detailed semantic information in testing code. We leave this point as future work.

The construct threats to the validity include the stack trace availability for failing test cases. Java projects with the JUnit framework often write test cases with assertions, which may throw exceptions if the test cases fail. Thus, our principles can be applied to Java projects without additional efforts. However, this may not apply to other programming languages or testing frameworks. The threat can be mitigated by augmenting or converting given test cases in a systematic way. Making test cases throw stack traces is not a complicated strategy; it can be readily applicable to other programming contexts.

VI. RELATED WORK

Fault localization in APR: Locating a (susicious) buggy element(s) in a source code is the first step of APR pipelines. Thus, many APR techniques rely on existing fault localization (FL) tools. There are two main lines of FL research utilized by APR tools: (1) spectrum-based fault localization (SBFL) and (2) information retrieval (IR)-based bug localization (IRBL). While some recent studies leverage the latter (such as iFixR [15]), most APR tools resort to the former as it can specify more precise locations (such as statements), even though it has a fundamental limitation that SBFL requires multiple passing and failing test cases with test oracles.

There have been investigations to identify the impact of FL techniques on the performance of the APR pipelines. These studies inspected how different FL settings affect fault localization accuracy as well as bug-fixing effectiveness. Liu [18] et al. studied to what extent FL techniques impact the performance of the APR pipeline. They investigated the bias on the performance comparison among APR tools caused by fault localization. Based on their findings, they called for new guidelines for assessing and reporting on the performance of APR systems. Our study is orthogonal to their work but proves that the bias of fault localization results can be reduced by optimizing the fault localization process with proposed principles. Wen et al. [59] examined the influence of the fault space on the success of finding correct patches by the APR tool, where fault space is defined as a ranked list of suspicious entities in a program. Our study directly considers the exact location of faults and their correlation with the success of fixing bugs.

Test cases for generate-and-validate APR: The quality of test cases is a critical factor to achieve a better performance in automated program repair. Most APR techniques (of course, including generate-and-validate APR), rely on test cases given by developers or generated by automatic test generation tools. For example, in generate-and-validate APR, test cases are leveraged to locate suspicious buggy statements, and also to validate that a patch candidate generated by an APR technique has actually fixed the given bug. In addition, test cases can be re-utilized when automatically determining whether a validated patch is correct or not [22], [60].

Recent studies examined the impact of test cases on the performance of program repair. Jiang et al. [61] pointed out that the weakness of real-world program test suites is a possible reason for the low performance of APR systems. They manually analyzed 50 real-world defects in the Defects4J benchmark, summarizing 7 fault localization strategies and 7 patch generation strategies to benefit defect localization and fixing without detailed evaluation of these strategies. However, our study concretely investigates to what extent our proposed principles can improve the performance of APR tools.

VII. CONCLUSION

To alleviate the burden of manual debugging, automated program repair (APR) tools have been explored in the latest decade. Generate-and-validate APR is one of the widely studied domains of APR that relies on the execution of test cases to spot bug positions and validate the correctness of generated patch candidates. In this study, we revisit the test cases in the process of fault localization and the process of program repair to boost the performance of APR. To this end, we first investigate the relationship between the bug-fixing performance of state-of-the-art APR tools and the spectrum-based fault localization as well as the quantitative perspective of failing test cases. We then dissect the characteristics of test cases to understand the achieved bug-fixing performance of state-of-the-art APR systems. Eventually, analyzing the information in stack traces after executing the bug-triggering test cases, we propose two principles to improve the performance of both fault localization and program repair. Our experimental results also confirm that the information in stack traces shows the potential of matching adequate fix patterns for bugs. Our replication package is publicly available at:

https://github.com/mrdrivingduck/TestCases4APR

ACKNOWLEDGEMENTS

This research was supported by the National Key R&D Program of China (No. 2020AAA0107704), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06) and the Open Project Program of the Key Laboratory of Safety-Critical Software (NUAA), Ministry of Industry and Information Technology (No. XCA20026), the National Research Foundation of Korea grant funded by the Korea government (No. 2021R1A5A1021944), as well as the funding from the European Research Council under the European Union’s Horizon 2020 research and innovation programme (No. 949014).


