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
{jingtangzhang, kui.liu, zhe.liu}@nuaa.edu.cn

†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, darkrsw@knu.ac.kr

¶Monash University, Melbourne, Australia, li.li@monash.edu

‖Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{jacques.klein, tegawende.bissyande}@uni.lu

***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.**

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

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 ❶ assess that the software behavior matches expected requirements and ❷ 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.

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 explored 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].

Spectrum-based fault localization leverages the execution traces of passing and failing tests to estimate the suspiciousness of each statement with a specific ranking formula [23] (e.g., Ochiai [24]). When a statement is executed by more failing tests and fewer passing tests, it is suspected to be faulty [25]. If the faulty statement can be spotted by the fault localization technique with a higher suspiciousness than other statements, APR tools can be more effective in addressing the fault. Recently, there have been studies investigating the impacts of fault localization on APR performance, but they have a limited scope. For example, Liu et al. [18] have recently demonstrated how localization failures hinder the performance of APR. Their empirical study, however, does not provide

DOI 10.1109/ICSME52107.2021.00010

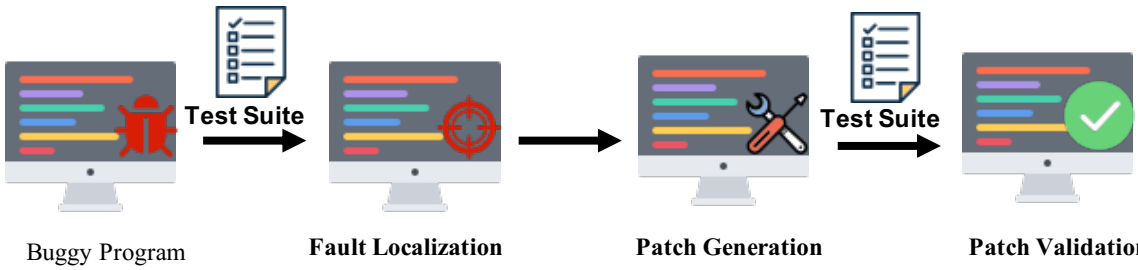

Fig. 1. Overview of APR process.

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 `NullPointerException` bugs. Resolving developer-defined exceptions remains however the most challenging.
5) The information in stack traces produced by bug-triggering test cases[1] 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.

[1]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.

II. Background

A. SBFL — Spectrum-Based Fault Localization

SBFL is a fault localization technique widely used in the APR community [26], [27], [28], [29], [30]. This technique takes as input a buggy program and its passing test cases as well as its failing test cases, and applies a ranking formula to the execution traces of all test cases to localize buggy code at the statement level by calculating the suspiciousness of each statement [23], [25]. In the APR literature [31], [32], [33], [34], [35], [36], [37], Ochiai [24] is the widely-used ranking formula of SBFL, which calculates the suspiciousness S_{ochiai} for a program statement s with the formula presented as below:

$$S_{ochiai}(s) = \frac{s_f}{\sqrt{(s_f + s_p) * (s_f + s'_f)}} \quad (1)$$

where s_f and s_p respectively denote the number of failing and passing test cases that executed the statement s, while s'_f is the number of failing test cases that did not execute the statement s. It shows that the precision of locating the faulty statement is relevant to the number of failing and passing test cases. To precisely rank the faulty statement to the top location that will be preferentially resolved by APR systems to generate patches, the buggy program should have more failing test cases to execute the faulty statement than other statements. If the faulty statement is not prioritized to the top, APR systems will have a large space of patch candidates for fixing bugs, as they will conduct lots of attempts on suspicious but non-faulty statements before the faulty one. If the faulty statement cannot be detected, all of the patch candidates generated by APR systems for non-faulty statements are useless.

B. Generate-and-Validate Automated Program Repair

Generate-and-validate automated program repair (APR) aims to shift the burden of manual debugging by automatically generate patches for fixing faults located in programs [7]. As illustrated in Figure 1, the APR system starts with a set of test cases, at least one of them will help to expose the defect of the buggy program in the fault localization step. In the next step, the APR system applies adequate modifications to the buggy program to generate patch candidates for the bug. As the final step, patch validation is dedicated to validating generated patch candidates to check whether it can make the patched program pass all tests or not. If the patched program passes all tests, the patch candidate is considered as a plausible patch for the buggy program, which produces correct outputs for all inputs in the test suite of the buggy program [33]. The plausible

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[2] 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[3] 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].

[2]When using JUnit, normal failing test cases always result in crashes since JUnit's assertions produce `Exceptions` if its condition is not satisfied.

[3]https://github.com/rjust/defects4j/releases/tag/v1.5.0

TABLE I
Defects4J-v1.5.0 dataset information.

Project	Bugs	kLoC	Tests
JFreeChart (Chart)	26	96	2,205
Closure compiler (Closure)	131	90	7,927
Apache commons-lang (Lang)	64	22	2,245
Apache commons-math (Math)	106	85	3,602
Mockito	38	11	1,457
Joda-Time (Time)	26	28	4,130
Total	391	332	21,566

*All information is excerpted from the Defects4J paper [45] and [51]. Four deprecated bugs[5](i.e., Closure-63, Closure-93, Lang-2 and Time-21) are not considered in this study.

TABLE II
APR tools for Java bugs studied in this work.

jGenProg [27], jKali [27], jMutRepair [27], HDRepair [32], Nopol [28], ACS [20], ssFix [21], ELIXIR [36], JAID [35], SimFix [26], CapGen [30], SketchFix [29], LSRepair [37], SOFix [52], ARJA [53], 3sFix [54], kPAR [18], AVATAR [49], TBar [48], PraPR [55], VFix [44], Hercules [50], FixMiner [17], ConFix [56], GenPat [57], DLFix [58].

For fault localization, we consider GZoltar[4] with Ochiai to expose 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.

To answer RQ1 and RQ2, we collect patches for Defects4J bugs that are generated by 26 state-of-the-art APR tools shown in Table II. The patches generated with the assumption of perfect fault localization are not considered, as the fault localization process is not actually included in the pipeline of generating such patches [49], [48], [16]. The collected patches are identified as *correct* or *incorrect*, where *correct* patches denote that the APR-generated patches are syntactically/semantically similar to developer-provided patches, while the *incorrect* patches are the plausible patches that overfit the test cases of buggy programs but do not actually fix the bugs. In this work, we follow the criteria provided by Liu et al. [46] to identify the correctness of APR-generated patches.

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, ❶ 214

[4]https://github.com/GZoltar/gzoltar

[5]https://github.com/rjust/defects4j

```
--- a/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
+++ b/source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java
@@ -1794,7 +1794,7 @@ public abstract class AbstractCategoryItemRenderer extends AbstractRenderer
       }
       int index = this.plot.getIndexOf(this);
       CategoryDataset dataset = this.plot.getDataset(index);
-      if (dataset != null) {
+      if (dataset == null) {
           return result;
       }
       int seriesCount = dataset.getRowCount();
```

Fig. 2. Patch diff for fixing bug *Chart-1*.

bugs are not fixed by any APR tools (*unfixed bugs*), ❷ 53 bugs are fixed by at least one APR tool with plausible but incorrect patches only (plausibly fixed bugs, denoted as *p-fixed bugs*), ❸ 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, denoted as *p&c-fixed bugs*), and ❹ 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 III
STATISTICS ON FIXED/UNFIXED DEFECTS4J BUGS.

Category	# bugs	# bugs′
unfixed bugs	214	245
only plausibly-fixed bugs	53	41
plausibly/correctly-fixed bugs	78	42
only correctly-fixed bugs	46	63

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

```
01: org.jfree.chart.renderer.category.AbstractCategoryItem
    Renderer@1793
02: org.jfree.chart.plot.CategoryPlot@1614
03: org.jfree.chart.plot.CategoryPlot@1613
04: org.jfree.chart.plot.CategoryPlot@1684
05: org.jfree.chart.plot.CategoryPlot@1682
06: org.jfree.chart.plot.CategoryPlot@1681
07: org.jfree.chart.plot.CategoryPlot@1679
08: org.jfree.chart.plot.CategoryPlot@1678
09: org.jfree.chart.plot.CategoryPlot@1675
10: org.jfree.chart.plot.CategoryPlot@1674
11: org.jfree.chart.plot.CategoryPlot@1673
12: org.jfree.chart.plot.CategoryPlot@1672
13: org.jfree.chart.plot.CategoryPlot@1667
14: org.jfree.chart.plot.CategoryPlot@1665
15: org.jfree.chart.plot.CategoryPlot@1340
16: org.jfree.chart.plot.CategoryPlot@1339
17: org.jfree.chart.plot.CategoryPlot@1358
18: org.jfree.chart.plot.CategoryPlot@1367
19: org.jfree.chart.plot.CategoryPlot@1365
20: org.jfree.chart.plot.CategoryPlot@1362
21: org.jfree.chart.plot.CategoryPlot@1357
22: org.jfree.chart.plot.CategoryPlot@1356
23: org.jfree.chart.plot.CategoryPlot@1353
24: org.jfree.chart.plot.CategoryPlot@1352
25: org.jfree.chart.plot.CategoryPlot@1046
26: org.jfree.chart.plot.CategoryPlot@1045
27: org.jfree.chart.renderer.category.AbstractCategoryItem
    Renderer@1798
28: org.jfree.chart.renderer.category.AbstractCategoryItem
    Renderer@1797
```

Fig. 3. Ranked list of suspicious statements for exposing bug *Chart-1*.

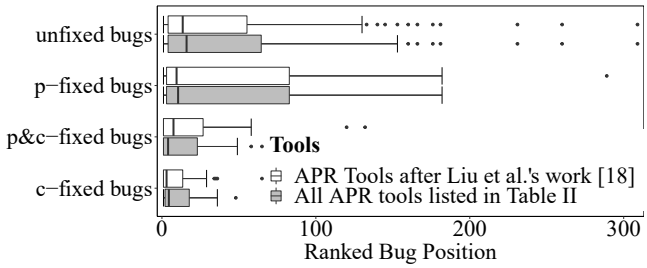

Fig. 4. Distribution of spotted bug locations for fixed/unfixed bugs (the smaller the better).

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

TABLE IV
DISTRIBUTION OF THE NUMBER OF FAILING TEST CASES OVER THE NUMBER OF FIXED/UNFIXED BUGS.

# failing test cases	# unfixed bugs	# p-fixed bugs	# p/c-fixed bugs & c-fixed bugs	Total
1	124	37	57+31 = 81 (32.5%)	249
2	41	8	10+11 = 21 (30%)	70
3	19	1	2 + 1 = 3 (13%)	23
4	7	2	3 + 0 = 3 (25%)	12
5	3	1	2 + 0 = 2 (33.3%)	6
6	3	1	1 + 0 = 1 (20%)	5
7	5	0	0 + 1 = 1 (16.7%)	6
8	4	2	0 + 2 = 2 (25%)	8
9	2	0	0 + 0 = 0	2
10	2	0	0 + 0 = 0	2
>10	4	1	3 + 0= 3 (37.5%)	8
Total	214 (54.7%)	53 (13.6%)	78+46=124 (31.7%)	391

way (i.e., Item 1) would be to efficiently filter out, with new patch validation methods, the plausible but incorrect patches generated by APR tools.

In addition, we inspect the impact of the number of failing test cases available when locating suspicious locations. As shown in Equation 1 of Section II, the fault localization performance is relevant to the number of failing test cases. We thus investigate whether the bug-fixing performance of APR tools is associated with the number of failing test cases. Table IV presents the distribution of the number of failing test cases over the number of fixed/unfixed bugs. From the quantitative aspect, most correctly fixed bugs have one or two failing test cases, while the ratio of correctly fixed bugs with one or two failing test cases is a bit higher than the ratio of correctly fixed bugs with at least three failing test cases. *From the aspect of quantitative test cases, it is difficult to explicitly assess the relationship between the number of failing test cases and the bug-fixing performance of APR tools.*

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

The failing test case of bug Chart-1:

```
396:  public void test2947660() {
          ......
407:    dataset.addValue(1.0, "S1", "C1");
408:    LegendItemCollection lic = r.getLegendItems();
409:    assertEquals(1, lic.getItemCount());
410:    assertEquals("S1", lic.get(0).getLabel());
411:  }
```

Excerpted stack trace after executing the test cases of bug Chart-1:

```
--- org.jfree.chart.renderer.category.junit.
     AbstractCategoryItemRendererTests::test2947660
junit.framework.AssertionFailedError: expected:<1> but
     was:<0>
    at junit.framework.Assert.fail(Assert.java:57)
    at junit.framework.Assert.failNotEquals(Assert.java:329)
    at junit.framework.Assert.assertEquals(Assert.java:78)
    at junit.framework.Assert.assertEquals(Assert.java:234)
    at junit.framework.Assert.assertEquals(Assert.java:241)
    at junit.framework.TestCase.assertEquals(TestCase.java
      :409)
    at org.jfree.chart.renderer.category.junit.AbstractCatego
      ryItemRendererTests.test2947660(AbstractCategoryItemRen
      dererTests.java:409)
     ......
```

Fig. 5. The failing test case and the excerpted stack trace after executing the test cases of bug *Chart-1*.

```
--- org.jfree.chart.axis.junit.LogAxisTests::
     testXYAutoRange1
java.lang.NullPointerException
    at org.jfree.chart.plot.XYPlot.getDataRange(XYPlot.java
      :4493)
    at org.jfree.chart.axis.NumberAxis.autoAdjustRange(
      NumberAxis.java:434)
    at org.jfree.chart.axis.NumberAxis.configure(NumberAxis.
      java:417)
    at org.jfree.chart.axis.Axis.setPlot(Axis.java:1044)
    at org.jfree.chart.plot.XYPlot.<init>(XYPlot.java:660)
    at org.jfree.chart.ChartFactory.createScatterPlot(
      ChartFactory.java:1490)
    at org.jfree.chart.axis.junit.LogAxisTests.testXYAuto
      Range1(LogAxisTests.java:260)
     ......
```

Fig. 6. Excerpted stack trace after executing the test cases of bug *Chart-4*.

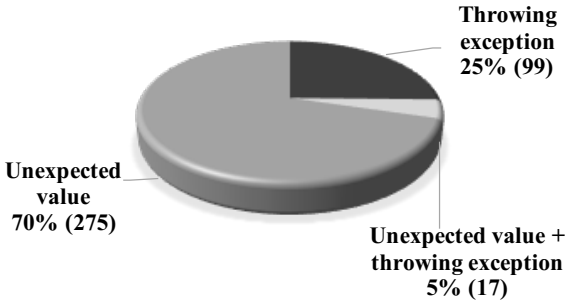

Fig. 7. Category of bugs with their failing behaviors.

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% $\approx \frac{33+51}{124}$) 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

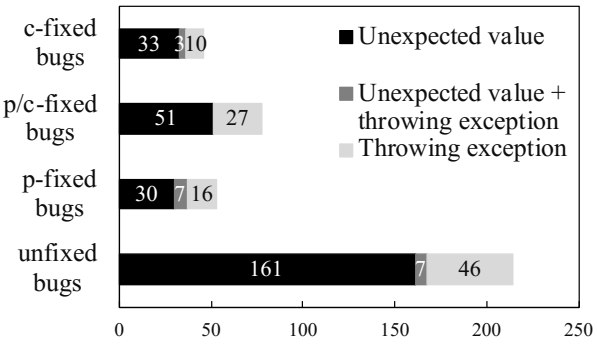

Fig. 8. Fixed/unfixed bugs with respect to the failing behaviors.

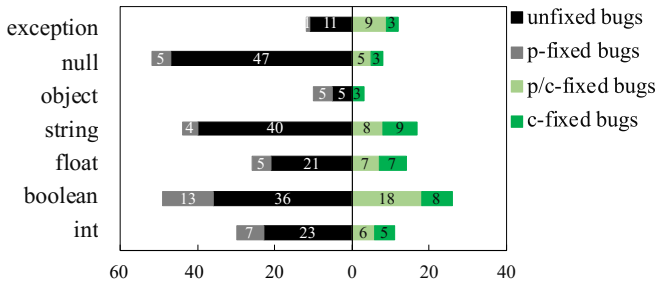

Fig. 9. Dissection of the fixed/unfixed bugs in terms of the data types of the expected values.

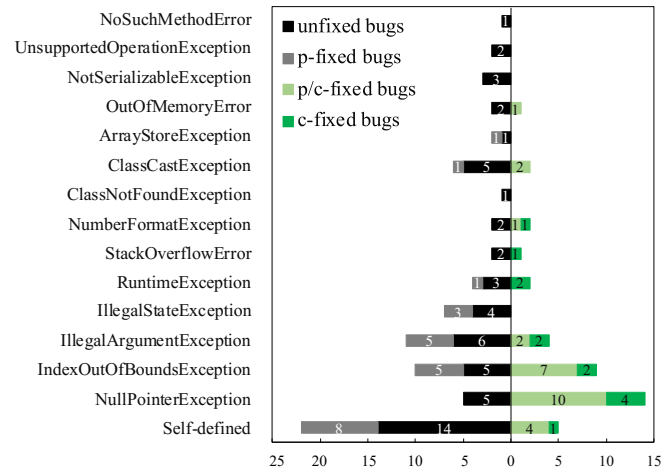

Fig. 10. Dissection of the fixed/unfixed bugs in terms of the types of throwing exceptions.

the best performance on fixing the bugs associated with the `boolean` type expected values (28.6% of correctly fixed bugs), followed by `string` (18.7%), `float` (15.4%), `exception` (13.2%), and `int` (12.1%). For the unfixed/incorrectly-fixed bugs, the non-null asserting bugs (i.e., `null` in Figure 9, asserting the returned value is not `null`) are the bugs that are most difficult to be resolved by APR systems, which is followed by the bugs expecting the `boolean` and `string` data type. ***The state-of-the-art APR systems presented the best performance on fixing bugs associated with*** `boolean` ***type expected values, while achieved 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.***

For the bugs throwing exceptions, the existing APR systems achieve the best performance on fixing `NullPointerException` bugs. 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`, and `IllegalStateException` exceptions 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

```
01: --- org.apache.commons.lang3.StringUtilsTest::testEscapeSurrogatePairs
02: java.lang.StringIndexOutOfBoundsException: String index out of range: 2
03:   at java.lang.String.charAt(String.java:658)
04:   at java.lang.Character.codePointAt(Character.java:4884)
05:   at org.apache.commons.lang3.text.translate.CharSequenceTranslator.translate(CharSequenceTranslator.java:95)
06:   at org.apache.commons.lang3.text.translate.CharSequenceTranslator.translate(CharSequenceTranslator.java:59)
07:   at org.apache.commons.lang3.StringEscapeUtils.escapeCsv(StringEscapeUtils.java:556)
08:   at org.apache.commons.lang3.StringUtilsTest.testEscapeSurrogatePairs(StringUtilsTest.java:2187)
09:   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
10:   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
11:   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
12:   at java.lang.reflect.Method.invoke(Method.java:498)
13:   at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:47)
14:   at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
15:   at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:44)
16:   at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
   ......
```

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

```
diff --git a/src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java b/src/main/java/org/apache/
    commons/lang3/text/translate/CharSequenceTranslator.java
index 0500460..4d010ea 100644
--- a/src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java
+++ b/src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java
@@ -92,7 +92,7 @@ public abstract class CharSequenceTranslator {
 //          // contract with translators is that they have to understand codepoints
 //          // and they just took care of a surrogate pair
             for (int pt = 0; pt < consumed; pt++) {
-                pos += Character.charCount(Character.codePointAt(input, pos));
+                pos += Character.charCount(Character.codePointAt(input, pt));
             }
         }
     }
```

Fig. 12. Developer's patch for fixing bug *Lang-6*.

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[6]) 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

[6]https://github.com/rjust/defects4j

```
01: --- org.joda.time.format.TestDateTimeFormatter::testParseInto_monthDay_feb29_newYork_startOfYear
02: org.joda.time.IllegalFieldValueException: Cannot parse "2 29": Value 29 for dayOfMonth must be in the range [1,28]
03:         at org.joda.time.field.FieldUtils.verifyValueBounds(FieldUtils.java:220)
04:         at org.joda.time.field.PreciseDurationDateTimeField.set(PreciseDurationDateTimeField.java:78)
05:         at org.joda.time.format.DateTimeParserBucket$SavedField.set(DateTimeParserBucket.java:483)
06:         at org.joda.time.format.DateTimeParserBucket.computeMillis(DateTimeParserBucket.java:366)
07:         at org.joda.time.format.DateTimeParserBucket.computeMillis(DateTimeParserBucket.java:359)
08:         at org.joda.time.format.DateTimeFormatter.parseInto(DateTimeFormatter.java:715)
09:         at org.joda.time.format.TestDateTimeFormatter.testParseInto_monthDay_feb29_newYork_startOfYear
            ......
```

Fig. 13. Excerpted stack trace after executing test cases of bug *Time-7*.

```
diff --git a/src/main/java/org/joda/time/format/DateTimeFormatter.java b/src/main/java/org/joda/time/format/
    DateTimeFormatter.java
index 913d036..447674a 100644
--- a/src/main/java/org/joda/time/format/DateTimeFormatter.java
+++ b/src/main/java/org/joda/time/format/DateTimeFormatter.java
@@ -700,14 +700,14 @@ public class DateTimeFormatter {
     public int parseInto(ReadWritableInstant instant, String text, int position) {
         DateTimeParser parser = requireParser();
         if (instant == null) {
             throw new IllegalArgumentException("Instant must not be null");
         }

         long instantMillis = instant.getMillis();
         Chronology chrono = instant.getChronology();
+        int defaultYear = DateTimeUtils.getChronology(chrono).year().get(instantMillis);
         long instantLocal = instantMillis + chrono.getZone().getOffset(instantMillis);
         chrono = selectChronology(chrono);
-        int defaultYear = chrono.year().get(instantLocal);

         DateTimeParserBucket bucket = new DateTimeParserBucket(
             instantLocal, chrono, iLocale, iPivotYear, defaultYear);
```

Fig. 14. Developer's patch for fixing bug *Time-7*.

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 methods in source code. 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

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 := ∅;
4      /* Initialize the suspicious lines L_m in M_p2.   */
5      L_m := ∅;
6      foreach l ∈ L do
7          if l ∈ S_p1 then
8              L.remove(l);
9              Continue;
10         foreach C ∈ C_p2 do
11             if l ∈ C then
12                 foreach M ∈ M_p2 do
13                     if l ∈ M then
14                         L_m.add(l);
15                         Break;
16                 if !(l ∈ L_m) then
17                     L_c.add(l);
18                 L.remove(l);
19                 Break;
20     L_r.addAll(S_p1).addAll(L_m).addAll(L_c).addAll(L);
21     Return(L_r);
```

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, 4, 65, and 53, 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.***

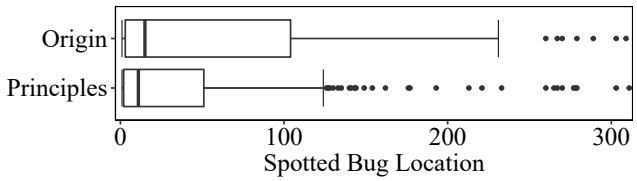

Fig. 15. Comparison on the fault localization performance.

2) Bug Fixing Performance: We further assess whether our proposed two principles can be used to improve 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)8 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 avoides 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 correct 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 the faulty positions refined with proposed principles.

TABLE V
DETAILS ON IMPROVED BUG-FIXING PERFORMANCE.

APR tools	GZoltar + Ochiai		GZoltar + Ochiai + Principles	
	# fixed bugs	CR (%)	# fixed bugs	CR (%)
AVATAR	26/82	31.7	(26+2+6)/(82-0+8)	+6.1
TBar	22/50	44.0	(22+2+11)/(50-10+12)	+23.3
SimFix	17/27	63.0	(17+(-1)+3)/(27-5*+3)	+13.0
ACS	10/20	50.0	(10+0+0)/(20-6+0)	+21.4

The data in the 4th column are presented in the format $(x+y+z)/(X-Y+Z)$, where (x)X represents the number of bugs (correctly) fixed by the APR tool with GZoltar + Ochiai. y and Y represent the number of bugs (plausibly fixed by the APR tool with GZoltar + Ochiai) correct fixed / unfixed by the APR tool with our proposed prinpicles, respectively. **CR** denotes the correctness ratio of generated patches.

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

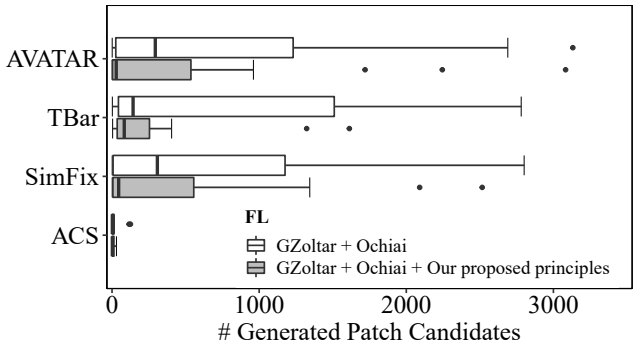

Fig. 16. Comparison on the number of patch candidates generated by AVATAR, TBar, SimFix and ACS.

observe that the efficiency of ACS is improved with a few increases as ① ACS produces far fewer candidate patches than other tools, so the improvement is not obvious; and ② 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 (suspicious) 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).

REFERENCES

[1] N. N. Release, "Software errors cost u.s. economy $59.5 billion annually," http://www.abeacha.com/NIST_press_release_bugs_cost.htm.

[2] R. Cohance, "Financial cost of software bugs," https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107.

[3] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009. [Online]. Available: https://doi.org/10.1109/MSP.2009.56

[4] G. J. Myers, T. Badgett, T. M. Thomas, and SandlerCorey, *The art of software testing*. Chichester: John Wiley & Sons., 2004, vol. 2.

[5] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Business School, University of Cambridge, Cambridge, UK, Technical Report*, 2013.

[6] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 3–13.

[7] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[8] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 17:1–17:24, 2018.

[9] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110817

[10] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[11] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2013, pp. 356–366.

[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[13] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 772–781.

[14] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016, pp. 702–713.

[15] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "iFixR: Bug report driven program repair," in *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 314–325.

[16] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[17] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "FixMiner: mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[18] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2019, pp. 102–113.

[19] 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*. ACM, 2012, pp. 378–381.

[20] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 2017, pp. 416–426.

[21] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 660–670.

[22] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 789–799.

[23] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[24] 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*. IEEE, 2007, pp. 89–98.

[25] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. ACM, 2017, pp. 609–620.

[26] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 298–309.

[27] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[28] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[29] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 12–23.

[30] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1–11.

[31] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 802–811.

[32] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 213–224.

[33] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.

[34] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 166–178.

[35] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 637–647.

[36] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 648–659.

[37] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*. IEEE, 2018, pp. 658–662.

[38] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 226–236.

[39] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 831–841.

[40] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

[41] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 524–535.

[42] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020, pp. 981–992. [Online]. Available: https://ieeexplore.ieee.org/document/9286101

[43] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, pp. 349–358.

[44] X. Xu, Y. Sui, H. Yan, and J. Xue, "VFix: value-flow-guided precise program repair for null pointer dereferences," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 512–523.

[45] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[46] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 625–627.

[47] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 135–146.

[48] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 31–42.

[49] ——, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 456–467.

[50] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 13–24.

[51] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, "Comparing developer-provided to user-provided tests for fault localization and automated program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 287–297.

[52] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 118–129.

[53] Y. Yuan and W. Banzhaf, "ARJA: automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, 2018.

[54] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *CoRR*, vol. abs/1811.05703, 2018. [Online]. Available: http://arxiv.org/abs/1811.05703

[55] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 19–30.

[56] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[57] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2019, pp. 255–266.

[58] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[59] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *CoRR*, vol. abs/1707.05172, 2017. [Online]. Available: http://arxiv.org/abs/1707.05172

[60] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "On the impact of flaky tests in automated program repair," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021, pp. 295–306. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00035

[61] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Sci. China Inf. Sci.*, vol. 62, no. 10, pp. 200 102:1–200 102:16, 2019. [Online]. Available: https://doi.org/10.1007/s11432-018-1465-6