Program Repair: Automated vs. Manual

Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang*, Ziyuan Wang*, Lingming Zhang

Abstract—Various automated program repair (APR) techniques have been proposed to fix bugs automatically in the last decade. Although recent researches have made significant progress on the effectiveness and efficiency, it is still unclear how APR techniques perform with human intervention in a real debugging scenario. To bridge this gap, we conduct an extensive study to compare three state-of-the-art APR tools with manual program repair, and further investigate whether the assistance of APR tools (i.e., repair reports) can improve manual program repair. To that end, we recruit 20 participants for a controlled experiment, resulting in a total of 160 manual repair tasks and a questionnaire survey. The experiment reveals several notable observations that (1) manual program repair may be influenced by the frequency of repair actions sometimes; (2) APR tools are more efficient in terms of debugging time, while manual program repair tends to generate a correct patch with fewer attempts; (3) APR tools can further improve manual program repair regarding the number of correctly-fixed bugs, while there exists a negative impact on the patch correctness; (4) participants are used to consuming more time to identify incorrect patches, while they are still misguided easily; (5) participants are positive about the tools’ repair performance, while they generally lack confidence about the usability in practice. Besides, we provide some guidelines for improving the usability of APR tools (e.g., the misleading information in reports and the observation of feedback).

Index Terms—automated debugging, automated program repair, human study, manual program repair

1 INTRODUCTION

Modern software systems continuously evolve with prevalent bugs, which have been widely recognized as notoriously costly and disastrous [1]. Manual debugging can be an extremely time-consuming and error-prone task due to the increasing size and complexity of software systems [2]–[4]. For example, shown in a prior report, software debugging often accounts for over 50% of the development cost of a software product [5], and consumes billions of dollars globally every year [6]. Therefore, a vast body of research effort has been dedicated to automated debugging, such as automated fault localization [7] and automated program repair (APR) [8], [9]. The former aims to directly localize software buggy elements to alleviate manual effort, while the latter aims to automatically fix software bugs without human intervention.

Despite an emerging research area, APR has been extensively studied in the literature and recent researches have made significant progress on effectiveness and efficiency [10]. According to a living review of APR research [11], multiple papers get published each year, introducing various delicately implemented APR tools. Among them, numerous studies evaluate APR tools on the effectiveness, in terms of the number of correctly-fixed bugs [8], [12]–[14]. For example, Ghanbari et al. [8] propose a novel tool PraPR based on mutation testing, which can successfully fix 43 real bugs from Defects4J [15]. Other advanced tools, such as ACS [12] and SimFix [13], have also produced promising results in terms of the number of bugs that can be fixed. Another performance aspect that deserves investigation is the efficiency, in terms of repair time and the number of patch candidates [8], [16]. Recently, Kui et al. [16] calculate the number of generated patch candidates before fixing a given bug to assess the efficiency.

Existing studies usually focus on the performance of the repair approaches regarding some criteria (e.g., repairability, correctness, and repair time) [16], [17]. APR is indeed a growing field, and it is essential to understand how it can be applied in practice when developers get involved. A natural question thus arises from the above scenario: How do state-of-the-art APR tools perform comparing with manual program repair? However, although APR has been extensively studied and even has drawn attention from industry (e.g., Facebook [18], [19] and Google [20]), direct deployment of APR tools in the industry seems to consume constant research effort. For example, research has identified that generated patches might be less readable and maintainable, even meaningless sometimes [21], [22]. Meanwhile, it is fundamentally difficult to achieve high precision for generated patches due to the weak test suites [10], [23], [24]. Besides, as repair reports generated by APR tools may provide suspicious elements and corresponding candidate fixes, developers could reduce repair effort with the assistance of these reports. Rather than direct deployment, such a semi-automatic approach may be a more feasible application of APR tools at this point. As such, are state-of-the-art APR tools beneficial to manual program repair as debugging aids?

In this paper, we perform a large-scale human study to bridge the current gap. Specifically, 8 real bugs with varied symptoms from the widely studied Defects4J benchmark and 3 state-of-the-art APR tools from all possible repair categories are randomly selected based on some well-designed criteria. As far as we are aware, this is the largest user study in APR field ever (e.g., there exist 6 bugs and 2 APR tools in
Our work reveals several notable observations. Firstly, manual program repair may be influenced by the frequency of repair actions (i.e., addition, removal, and modification over code elements), while APR tools are effective for the bugs with certain types due to their design mechanisms. Moreover, patches generated by manual program repair usually have a higher correctness ratio and code quality than those generated by APR tools. Meanwhile, for successful repair tasks, APR tools are more efficient in terms of repair time, while participants need fewer attempts to generate a correct patch.

Secondly, we analyze whether manual program repair benefits from APR tools. Our analysis confirms that APR tools indeed improve manual program repair regarding the number of correctly-fixed bugs, while the tools have a negative impact on the patch correctness and may not help reduce debugging time. Besides, the types of repair tools (i.e., heuristic-based, constraint-based and template-based) have little influence on the patch correctness and debugging time, while the types of repair reports (log report, correct report and incorrect report) have a more significant impact on the patch correctness. Further, when provided with incorrect patches, participants tend to consuming more debugging time, while they are still easily misguided.

Finally, we qualitatively investigate participants’ opinions on the assistance of APR tools. There is a huge gap between the repair performance in the benchmark and the usability in practice for state-of-the-art APR tools. Besides, the feedback in log output should be valued more in the future. Moreover, participants are positive about the quick identification of buggy elements and candidate fixes, while they generally lack confidence in the accuracy and format of repair reports.

Based on these observations, we provide various practical guidelines on how to improve the usability of APR tools so as to benefit manual program repair further. For example, we discuss how to present repair reports in a more user-friendly way and reduce misleading repair information. Above all, we suggest that APR tools should pay more attention to the feedback in the log output, which can provide useful guidelines for all possible bugs. Overall, our contributions are summarized as follows.

- We conduct the first systematic large-scale human study to compare manual program repair against three state-of-the-art APR tools, and analyze whether the assistance of APR tools can improve manual program repair.
- We perform a questionnaire survey to analyze participants’ attitudes toward the assistance of APR tools, and further provide various practical guidelines on how to improve the usability of APR tools in practice.
- We release all experimental data (including all raw data, experimental script, and result analysis) for replication and future research on APR at https://github.com/AnonymousAPR/Data.

## 2 RELATED WORK AND MOTIVATION

### 2.1 Automated Program Repair

#### 2.1.1 Typical repair steps

The primary objective of APR techniques is to identify and fix the bug automatically. A typical repair technique is usually composed of three steps: (1) off-the-shelf fault localization techniques are used to recognize the suspicious code elements [26]–[28], (2) these elements are then modified based on a set of transformation rules to generate various new program variants (i.e., candidate patches); (3) the original test suite is adopted as the oracle to verify all candidate patches. Specifically, a candidate patch passing the original test suite is called a plausible patch. A plausible patch, which is also semantically equivalent to the developer patch, denotes a correct patch.

In practice, a typical repair report generated by an APR tool can be found in Fig. 1. Specifically, the report is usually divided into two parts (i.e., the log output and patch output), where the former contains the log information returned by the first two steps (i.e., the suspicious statements and candidate patches already tried), and the latter contains the patch information returned by the last step (i.e., the plausible patch passing all the available test cases). As a reminder, the repair report will only contain the log output if the APR tool cannot generate a plausible patch for a given bug.

![Fig. 1: A typical repair report generated by SimFix](https://example.com/simfix_report.png)

#### 2.1.2 Patch generation technique

In the literature, a variety of techniques [29]–[36] are adopted to generate patches based on different mechanisms. Following recent APR work [16], [37], [38], we categorize them into four main categories: heuristic-based, constraint-based, template-based and learning-based repair techniques.
• **Heuristic-based repair techniques.** These techniques usually use a heuristic algorithm to find a valid patch by iteratively exploring a search space of syntactic program modifications [39]–[41]. Among them, GenProg [39] proposed in the early days has been considered a seminal work in this field, which performs delete and replace operations at the abstract syntax tree (AST) level. The recent SimFix technique [13] utilizes code change operations from existing patches and similar code snippets to build two search spaces. Then, the intersection of the above two search spaces is further used to search the final patch by basic heuristics.

• **Constraint-based repair techniques.** These techniques mainly focus on conditional statement repair, which can repair more than half of the bugs in existing approaches [42]–[44]. In detail, these techniques transform the patch generation into a constraint solving problem, and use a solver to obtain a feasible solution. For example, Nopol [44] relies on an SMT solver to solve the condition synthesis problem after identifying potential locations of patches by angelic fix localization and collecting test execution traces of the program. Among them, ACS [12] proposed to refine the ranking of ingredients for condition synthesis is considered as one of the most advanced constraint-based repair techniques [16].

• **Template-based repair techniques.** These techniques generate patches by mutating a buggy program, similar to mutation testing [45]. In detail, these techniques generate patches by following fix patterns to mutate buggy code entities with retrieved donor code [46]–[48]. For example, Liu et al. [47] revisit the repair performance of fix patterns via a systematic study assessing the effectiveness of a variety of fix patterns summarized from the literature. Among them, the most recent technique PraPR [8] is able to generate plausible and correct patches for 443 and 43 real bugs, respectively, which is the largest number of bugs reported as fixed for Defects4J when published.

• **Learning-based repair techniques.** These techniques attempt to fix bugs enhanced by machine learning techniques [13], [49]–[53]. For example, Tufano et al. [14] extensively evaluate the ability of adopting neural machine translation (NMT) techniques to generate patches from bug-fixes commits in the wild. Furthermore, Lutellier et al. [51] propose a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately, to automatically fix bugs in multiple programming languages. Unlike techniques in above three categories, learning-based techniques generally require extra training data (i.e., the tuples of buggy, context, and fixed lines of code) to capture complex relations between buggy and fixed code.

In our work, we initially aim to consider all existing APR tools. However, due to the fact that our human study involving too many tools may be unaffordable, we select several representative APR tools from the above categories based on the criteria (Section 4.1) for our evaluation.

### 2.2 Automated debugging with human intervention

Various automated debugging techniques have been proposed over the past two decades [27], [28]. Furthermore, various empirical studies involving human intervention are also conducted to evaluate the usefulness of automated debugging techniques [51]–[55].

In similar areas of fault localization, some studies have been conducted to analyze the performance of program debugging with or without human intervention [51]–[55]. Among them, SBFL utilizes testing outcomes to evaluate the suspiciousness for each program unit, and then they are ranked in descending order and provided to programmers as the suggested fault location. Besides, Parnin et al. [54] require programmers (i.e., students) to perform debugging tasks with or without suspicious statements given by spectrum-based fault localization (SBFL) techniques. By addressing some significant problems that are not resolved in the above study, Xie et al. [57] focus on the reasons behind the observed results via a quantitative focus-tracking analysis to revisit the actual helpfulness of SBFL.

In the field of APR, Tao et al. [25] investigate the usefulness of automatically generated patches as debugging aids, where ten plausible patches (i.e., passing all the corresponding test cases) are selected from GenProg [39] and PAR [59]. However, they focus on the impact of selected patches’ quality and do not concern other feedback (e.g., the fixes already tried and patch execution information) in the repair reports. To date, even state-of-the-art APR tools can only fix a small ratio of real bugs (i.e., < 20% for Defects4J) [7], [8], [12]), and such tools seem useless for the vast majority of unixed bugs. Recently, Lou et al. [7] confirm that the feedback (i.e., the patch execution information) can provide useful guidelines for powerful fault localization, even when the APR tool fails to return a plausible patch for a bug.

Our insight is that such repair feedback can also provide useful debugging hints for repair tasks, which can be adopted by developers directly. For example, when we apply state-of-the-art APR technique, SimFix, on a real bug Time-15 (i.e., denotes the 15th buggy version of Joda-Time project [60] from Defects4J, no plausible patch is produced. Table 1 lists the top-5 most suspicious statements and the number of patch candidates already tried. SimFix generates only one candidate patch for the buggy statement $s_2$, while more than 80 attempts are made for the statement $s_4$. Although no plausible patch is returned, we still observe other useful feedback for repairing the bug in the repair report. For example, there exist 13 candidate patches for $s_4$ containing the same code snippet in Listing 1 which is very similar to the correct patch in Table 2. Thus instead of focusing on the limited selected patches, our study considering the whole repair report, extends the evaluation scope of APR to all possible bugs.

```java
if(total/val1!=val1||val1==Long.MIN_VALUE&&val2==-1||val2==Long.MIN_VALUE|val1sh1==-1){
    throw new ArithmeticException("Multiplication overflows a long: "+val1);
}
```
Besides, different APR tools vary dramatically in settings (i.e., the patch generation strategy and report format), which may affect programmers’ repair performance. For example, ACS and SimFix return only one most plausible patch in source-code level while PraPR returns all plausible patches in bytecode level for each bug. Thus multiple state-of-the-art APR tools from different categories are selected in our evaluation. Based on the above insights, we present an extensive study to compare the performance of manual program repair with state-of-the-art APR tools and investigate whether the assistance of such tools benefits manual program repair.

3 RESEARCH QUESTIONS

In this study, we aim to investigate the following three research questions.

RQ1. How do state-of-the-art APR tools perform comparing with manual program repair?

To answer RQ1, we compare the number and quality of patches generated by manual program repair and several state-of-the-art APR tools. We also distinguish how the characteristics and type of bugs impact the two repair scenarios. Further, we analyze the repairing efficiency in terms of repair time and the number of candidate patches.

RQ2. Are state-of-the-art APR tools beneficial to manual program repair as debugging aids?

To answer RQ2, we discuss whether the assistance of APR tools (i.e., repair reports) can further boost the performance of manual program repair. Specifically, we also investigate how manual program repair is influenced by different factors (e.g., patch generation techniques, report formats)

RQ3. What are participants’ opinions on the use of state-of-the-art APR tools?

To answer RQ3, we perform a questionnaire survey to qualitatively analyze participants’ opinions on the assistance of APR tools, such as the positive and negative thoughts.

4 EMPIRICAL STUDY

In this section, we enumerate the subject programs, participants and state our study design in detail.

4.1 APR tools

Although we initially plan to consider all repair tools proposed in the last decade [37], we are limited by the fact that the size of selected tools may lead to endless human study. Thus, following prior work [17], [25], [61], we systematically consider several representative tools in our experiment based on the following criteria.

1) All selected tools are required to be publicly available and possible to run, as we need to run the tools in the experiment.

2) The number of selected tools should be proper, as our human study involving too many tools may be unaffordable (e.g., two tools are used in [25]).

3) The selected tools can represent all possible categories, as we want to consider all repair techniques in the experiment.

4) The tool should require only the source code of buggy program and its corresponding test suite, as the two elements are the two inputs specified in the problem statement of APR [17]. Thus, according to recent empirical studies [16], [37], all learning-based tools are excluded (e.g., [51], [53]).

Accordingly, we select three repair tools, each representing the state-of-the-art technique in the corresponding category. Specifically, the selection configuration is the same as a recent APR work [61] except PraPR, because only source-level APR tools are considered in that work. The first one is a template-based tool, PraPR, which is proposed recently and fixes bugs at the bytecode level. Compared against other state-of-the-art tools, PraPR is able to fix more bugs with a much lower overload and is considered as one of the most advanced repair tools. The rest two are a heuristic-based tool SimFix and a constraint-based tool ACS.

4.2 Subject Dataset and Bugs Selection

There exist several benchmarks in recent APR literature [62]–[64]. After searching the literature for benchmarks, we adopt Defects4J [15], as it has been continuously developed for a long time and has become the most widely studied dataset in APR studies [7], [8], [16], [37], or even other software engineering research (e.g., fault localization [65]–[68] and test case prioritization [69], [70], etc.) in general. Defects4J consists hundreds of known and reproducible real-world bugs from a collection of 16 real-world Java programs.

Note that we are unable to successfully apply ACS and SimFix to the new subjects (e.g., Mockito project) beyond the original paper [12], [13]. SimFix is unable to locate reusable code snippets in new subjects and ACS is no longer allowed to use programmed queries in GitHub [1] due to the changed interface. Thus, according to the recent APR study [37], the remaining four subjects (i.e., Chart, Time, Lang, and Math projects) are used in our experiment.

In recent work, Durieux et al. [17] show that APR tool may overfit Defects4J dataset in terms of repairability. Thus, instead of adopting the entire dataset, we carefully select several representative bugs to mitigate the overfitting problem that satisfy the following criteria:

1. https://github.com/.
1) The number of total selected bugs should be proper, to control the scale of human study (e.g., 5 bugs are used in [23]).

2) The diversity of bug types is preferred, to represent the whole dataset [71].

3) The performance distribution of APR tools on the whole selected bugs should be balanced, to avoid single APR tool may overfit the whole selected bugs.

4) The performance distribution of APR tools on each single selected bug should be varied, to avoid single bug type may overfit the whole selected tools.

Accordingly, we randomly select two bugs for each subject, resulting in a total of eight real bugs. The detailed information is summarized in Table 2 where column “BugID” presents the version of the buggy subject, and the remaining two columns present the developer patch and corresponding description of the bug. Specifically, these selected bugs manifest different symptoms and cover various bug types. There exist four bugs involving conditional statements (i.e., Time-7, Chart-7), and two bugs involving assignments and return statements (i.e., Math-5, Chart-12). The distribution of bug types is consistent with that of Defects4J [71], which indicates that they can represent the whole dataset well. Meanwhile, we conform all APR tools perform similarly for the whole selected bugs (e.g., each tool can correctly, plausibly and abortively fix 1/3 bugs approximately). Besides, APR tools perform diversely for each bug (e.g., each bug is correctly-fixed, plausibly-fixed and abortively-fixed by the three tools respectively) , to ensure each bug type can provide various report outcomes.

| Buggy code snippet and Developer patch | Symptom and illustration |
|----------------------------------------|--------------------------|
| 1. if (str.startsWith("--")) { 2. return null; 3. } 4. ... 5. if (str.trim().startsWith("--")) { 6. throw new NumberFormatException("str + " is not a valid number."); 7. } | - The buggy method createBigDecimal() would return null when the string starting with "--" is given, which is contrary to the behaviour of other methods and would throw NumberFormatException. - The developer patch fixes the bug by throwing an appropriate exception in the method createNumber() when the above string is given. |
| 1. boolean wasWhite= false; ... 2. if (Character.isWhitespace(c)) { 3. if ("wasWhite") { 4. wasWhite= true; 5. } 6. regex.append("\s+"); 7. } 8. continue; 9. } 9. wasWhite= false; | - The buggy code treats white-space specially, which is in conflict with other methods in the program. - The developer patch fixes the bug by commenting the relevant code. |
| 1. if (real == 0.0 && imaginary == 0.0) { 2. return NaN; 3. return INF; 4. } | - The buggy code returns the wrong value, and throws AssertionFailedError in line 2. - The developer patch fixes the bug by returning correct value. |
| 1. if (yMin * yMax > 0) { 2. throw MathRuntimeException.createIllegalArgumentException(NON_BRACKETING_MESSAGE, min, max, yMin, yMax); 4. } | - The buggy code fails to throw an illegalArgumentException if the values of the method at the three points have the same sign. - The developer patch fixes the bug by inserting conditional statements to checking the values. |
| 1. long s = getDataItem(this.minMiddleIndex).getPeriod().getStart() 2. long s = getDataItem(this.maxMiddleIndex).getPeriod().getStart() 3. .getTime(); 4. long e = getDataItem(this.minMiddleIndex).getPeriod().getEnd() 5. .getTime(); 6. .getTime(); | - The buggy code returns the wrong index of the start (end) time and throws an AssertionError-FailedError. - The developer patch fixes the bug by changing related parameters in called methods. |
| 1. super(); 2. this.dataset = dataset; 3. setDataset(dataset); | - The dataset is not wired to a listener, when dataset is passed into constructor for MultiplePiePlot. - The developer patch fixes the bug by setting the new dataset, and registering the chart as a change listener. |

TABLE 2: The details of selected bugs
4.3 Participants

There are 20 participants involved in our experiment. All of them are carefully selected from graduate-level software engineering classes. Besides, as a part of their course work, they are familiar with software testing and debugging. It is noteworthy that we confirm the selected students’ backgrounds represented a full range of experiences. Specifically, some students are professional developers with rich experience in industry (and return to school), others have one or more internships in companies, and yet others have limited programming experiences outside of school. As such, we are comfortable with this variance in experiences, which can help to provide reliable results in our analysis.

4.4 Experiment platform

We have developed Mooctest, an online platform for a Coursera course on software testing, which can provide all supportive features and consists of three major modules:

1) Arrangement module is able to arrange bugs and corresponding test suite for all participants automatically.
2) Submission module is able to record all patches submitted by participants.
3) Evaluation module is able to identify whether the submitted patch can pass the available test cases or not.

Unlike the study design in [25], we schedule a repair task for one bug individually, as putting all bugs together may influence participants’ debugging behaviour. They may be free to skip any bugs, which is inconsistent with a practical debugging scenario [57].

4.5 Evaluation metrics

We measure the repair performance by the effectiveness (i.e., repairability and correctness) and efficiency (i.e., debugging time and the number of generated patch candidates). We also qualitatively investigate participants’ opinions on using APR tools as debugging aids by the survey feedback.

Repairability focuses on the number of correct patches. A patch is considered to be correct if it can pass all test cases and be semantically equivalent or similar to the corresponding developer patch (described in Table 2).

Thus for the plausible patches (checked by our platform automatically), two volunteers with 7-year Java programming experience conduct three rounds of manual inspection to identify whether the patches are correct or not, according to the previous work [25].

Correctness indicates whether a generated patch is correct or not. It is widely considered that patches generated by an APR tool should have a high probability to be correct, while the overfitting problem (i.e., the generated patch is plausible but overfitting) is still a major and long-standing challenge [1].

Debugging time is recorded automatically. For manual program repair, debugging time repair is calculated by the time elapsed from each repair task beginning until the patch submission. For automated program repair, debugging time is calculated by the repair report (e.g., Fig. 1).

Fig. 2: Overview of the experimental process

The number of patch candidates indicates how many attempts automated (manual) program repair makes before fixing the bug. This metric is independent of a few redundant variables (i.e., machine configurations) that are unrelated to the approach implemented in APR [16], and can provide reliable results to supplement efficiency evaluation in our study.

Feedback is collected from the questionnaire survey. The survey is conducted online and all participants can submit their free-form answers on condition of anonymity to mitigate some risks, which could bias the results (e.g., participants might feel that the answers could affect their performance).

4.6 Experimental procedure

Firstly, an introduction to APR tools and our experimental platform in the form of document and presentation video is given to all participants, to ensure they fully understand the experimental procedure. Then, to mitigate the effect of our platform’s usability, we conduct a pre-test involving an additional real bug from Defects4J. Finally, participants are required to fix all eight real bugs independently within four hours. Fig. 2 presents an overview of the experimental procedure. As a reminder, the restriction on repair hours is imposed to avoid endless debugging. Based on participants’ performance in the pre-test and study designs in previous studies [25], [57], 4 hours should be adequate for participants to complete repair tasks at a reasonably comfortable pace.

Specifically, to answer RQ1, we compare the repair performance of the participants with no aid against three state-of-the-art APR tools. To answer RQ2, we compare the repair performance of the participants with no aid against the participants with APR aids (i.e., repair reports produced by the tools). However, as repair reports already reveal suspicious code elements, for a fair comparison, we provide buggy location reports to the control group instead of leaving it completely unaided. The buggy location report is generated by an off-the-shelf fault localization technique Ochiai [26], which is widely adopted in recent APR studies (e.g., PraPR, SimFix and ACS) [8], [9], [13]. For simplicity, participants are rated in four levels based on programming experience, and then participants within the same level are randomly divided into five groups: None group (unaided), PraPR group (PraPR aided), ACS group (ACS aided), SimFix group (SimFix aided), and Location group (buggy location aided), resulting in a total of 160 manual program repair tasks (i.e., 4 participants and 8 bugs for each group). Thus, the first group (32 repair tasks) is adopted to answer RQ1 and the other groups are adopted to answer RQ2. Besides, to answer
TABLE 3: The effectiveness comparison between automated and manual program repair

| Program | Participants | PraPR | SimFix | ACS |
|---------|--------------|-------|--------|-----|
| Time-7  | (0)          | (0)   | (1)    | (0) |
| Time-15 | (2)          | (0)   | (0)    | (1) |
| Lang-10 | (0)          | (0)   | (0)    | (0) |
| Lang-7  | (4)          | (0)   | (0)    | (1) |
| Math-73 | (2)          | (0)   | (0)    | (1) |
| Math-5  | (4)          | (1)   | (1)    | (1) |
| Chart-7 | (4)          | (0)   | (1)    | (0) |
| Chart-12| (1)          | (1)   | (0)    | (0) |
| All     | (17)         | (3)   | (6)    | (4) |

RQ3, all the participants attend an online survey to submit their opinions on the assistance of APR tools.

5 RESULTS AND ANALYSIS

In this section, we will analyze our experimental results to address the three research questions.

5.1 RQ1: Comparison between automated program repair and manual program repair

To answer RQ1, we measure the repair performance of three state-of-the-art APR tools and manual program repair (participants from None group) in terms of effectiveness and efficiency. Thus, there exist 32 repair tasks performed by participants and 24 repair reports generated by APR tools for the eight bugs. It is noteworthy that although some criteria (e.g., repairability [17] and efficiency [16]) have been studied in the literature, there still exist little systematic work to compare APR tools with manual program repair.

5.1.1 Effectiveness

We first analyze the repairability (i.e., the number of correctly-fixed bugs) of automated and manual program repair overall. Then, for each bug, we distinguish how the types of bugs impact automated and manual program repair. We also investigate whether there exists a difference in patch correctness and code quality between automated and manual program repair.

Table 3 presents the repair information, where per row represents all four repair scenarios for each bug (i.e., PraPR, SimFix, ACS and participants). Specifically, each cell is represented as $x/y$, where $x$ is the number of correct patches and $y$ is the number of produced plausible patches by participants (APR tools). It is noteworthy that the submissions will not be counted if no changes are made by participants.

Repairability Evaluation. Overall, participants can fix six of all eight bugs, twice as that of each APR tool. However, a combination of the three tools can fix more bugs (except for Math-73) than that of participants. We further analyze how the bug types impact the repair performance of automated and manual program repair.

As repair action regarding condition is able to fix half of the bugs in existing approaches [12], we firstly focus on the bugs (i.e., Time-15, Lang-10, Lang-7 and Math-73) involving conditional statement. Specifically, for the bugs involving conditional statement (i.e., Time-15, Lang-10, Lang-7 and Math-73), participants can fix three of them, except Lang-10. In fact, participants need to remove the corresponding conditional statements without adding any statements when attempt to fix bug Lang-10. It is rare to perform only removal actions in repair tasks, as most bugs are fixed by modification or addition actions. We confirm that there are fewer than 1% (2/395) bugs requiring only removal actions in Defects4J [21]. Thus, participants may ignore such low-frequency repair actions due to their unconscious mind.

Similar to manual program repair, ACS cannot correctly fix Lang-10, as it aims at generating precise conditions. Meanwhile, SimFix fails to generate any correct patches for all these bugs, as no similar code snippets containing correct repair actions are found. For example, though there is one code snippet similar to the inserted statement for bug Time-15, their context varies greatly. Thus SimFix cannot extract such code snippets, which may be the reason why this tool is unsatisfactory. Besides, PraPR can only fix Lang-10 by RemoveConditionalMutator and cannot generate any additional conditional statement with correct predicates for the other three bugs.

It is also observed that participants’ performance varies in the bugs involving method call (i.e., Time-7, Chart-7). Intuitively, the edit distance for Chart-7 is 24, and the one for Time-7 is 132. An explanation may be that fixing Time-7 needs to modify the called method and corresponding parameters, where it is difficult to search the right parameters. While the repair action on Chart-7 is simple, all participants are able to fix the bug. It is also rough for mutation-based tool PraPR to fix Time-7, which needs two chunks (i.e., a sequence of continuous changes). Participants and PraPR can fix all the remaining bugs involving assignments and return statements (i.e., Math-5, Chart-12). However, ACS and SimFix cannot fix Chart-12 because of the primary design mechanism.

Different APR tools are suitable for the bugs with corresponding types because of their design mechanisms. In contrast, participants can always repair all types of the bugs with a deeper understanding of the program, although they may ignore some low-frequency repair actions (e.g., removal actions in Lang-10) due to the unconscious mind sometimes.

Correctness Evaluation. As prior work [1], [10], [23], [21], [22] confirms that APR tools may generate floating patches (i.e., patches passing the entire available test suite may not generalize to other potential test cases), we thus investigate whether there are differences in the patch correctness between automated and manual program repair.

As shown in Table 3, the correctness of manual patches is over 90% (17/18), which is far higher than that of auto-generated patches (about 25% ≈ 9/37). APR tools usually use the available test suite to verify the generated patches. However, a patch passing the available test suite may not generalize to other potential test cases. APR tools may produce many incorrect patches with insufficient test cases, even for ACS, which aims to achieve high precision. Instead, participants can identify the generated patch correctness by human ability in comprehending the source code (e.g., identify the functionality of the buggy method from Javadoc comments).

The correctness of auto-generated patches heavily depends on the quality of the available test suite, while participants generate correct patches with a deeper understanding of the source code.

Quality Evaluation. As prior work suggests that gen-
Table 4 provides statistical information about debugging time and the number of patch candidates. In this table, column "#Patch" presents the minimum/average/maximum number of all patch candidates, as well as the time cost for column "Time(s)". It is observed that manual program repair always consumes the more debugging time (1985%, 163% and 719%) than APR tools on average. On the contrary, it seems that manual program repair needs fewer average attempts (1.7) to generate a correct patch than those of all APR tools (960.3, 540.0 and 2.0). Specifically, participants usually need to understand the functionality of the buggy method and before attempting to fix it. In contrast, APR tools can use heuristic-based methods to generate a mass of program variants based on various variation rules, and then adopt a test-driven approach to find plausible patches. As a result, manual program repair may require a significant amount of time in the comprehension process, and APR requires lots of attempts in the verification process. Although such conclusions may be unsurprising and confirmed intuitively, such a confirmation warrants rigorous and substantial experiments.

Participants usually require more debugging time to fix a bug owing to the comprehension process, while APR may need more attempts to generate a plausible patch. Another interesting finding is that manual repair cost more than all 3 tools combined, while fixing less bugs than the 3 tools.

Answer to RQ1: Overall, our comparison between automated and manual program repair reveals that (1) manual program repair habitually ignores the repair actions with a low frequency; (2) the repair tool is effective for the bugs with certain types due to their design mechanisms; (3) patches generated by manual program repair has a higher correctness ratio and quality than APR; (4) APR tools usually require less debugging time, while they need more attempts to generate a plausible patch.

5.2 RQ2: The assistance of state-of-the-art APR tools
Specifically, we first analyze repairability of participants from four groups (i.e., Location, SimFix, ACS and PraPR group) to judge whether APR aids (i.e., repair reports) can improve participants’ performance. Then we analyze how the factors (e.g., the report type and patch quality) impact participants’ performance, so as to further contribute to improving the state-of-the-arts.

5.2.1 Overall performance
For simplicity, to evaluate repairability of four groups, we use completion ratio $C_r$, defined as $P_n/T_n$, where $P_n$ is the number of participants who have correctly fixed the bug within the stipulated time (i.e., the number of correct patches), and $T_n$ is the total number of all participants taking part in this group (i.e., 4 participants).

Fig. 4 demonstrates the comparison for all eight bugs, where the four bars per bug represent four groups, resulting in a total of 32 repair types (8 bugs * 4 groups). There
are a total of nine repair types achieving the highest $C_e$ in each bug, eight of them from APR groups, and one of them from Location group. Meanwhile, Time-7 and Lang-10 can be fixed only if SimFix and PraPR reports are provided. It seems that the group with repair reports has a high chance of achieving higher $C_e$ and participants may fix some previously unrepaired bugs with appropriate repair reports. However, there are also a total of 17 repair types achieving the lowest $C_e$, 10 of them from APR groups, and only 4 of them with Location group. Meanwhile, participants can fix Math-73 with the buggy location report, while no repair tasks are successful with SimFix and ACS reports. It seems that some repair reports may also decrease $C_e$ and even mislead participants resulting in failed repair tasks.

In other words, manual program repair overall can benefit from state-of-the-art APR tools in terms of the number of correctly-fixed bugs, while in some cases, such tools may also have a significant adverse effect.

5.2.2 Performance impacted by the tool types

Existing APR tools adopt various patch generation techniques based on different mechanisms. For example, ACS focuses on condition synthesis and SimFix relies on similar code snippets. Also, both of them return only one most plausible patch in source-code level, while PraPR returns all plausible patches in bytecode level. Based on these insights, we further investigate the manual program repair performance impacted by APR tools.

Fig. 4(a) shows the percentage of correct patches for the groups. Specifically, patches submitted by the Location group are 70% (16/23) correct, the same as ACS group. The percentage of correct patches decrease to 51% (15/29) for SimFix group. PraPR reports do not improve correctness too, with only 65% (15/23) patches being correct.

Fig. 4(b) shows the debugging time for the groups. Location group has an average debugging time of 21.8 minutes, which is slightly slower than 20.8, 17.8, and 18.8 minutes for PraPR, SimFix and ACS groups, respectively.

It seems that repair reports may reduce participants’ debugging time slightly, while patch correctness drops to an even lower point than that of the control group. Meanwhile, the three repair tools have little influence on the patch correctness and debugging time.

5.2.3 Performance impacted by the report types

The recent APR studies [7], [8] demonstrate that, even state-of-the-art APR tools can only fix a small ratio of real bugs (i.e., <20% for Defects4J) fully automatically and are simply aborted for the vast majority of unfixed bugs. Based on the repair outcomes of APR tools, the repair reports may contain different types of information (i.e., log and patch output), which could impact participants’ debugging strategies and performance. For example, the repair report only contains log output for the vast majority of unfixed bugs. Thus, we want to investigate whether APR tools can also provide useful debugging information to help manual program repair even for the bugs that are hard to fix automatically.

Specifically, we divide repair reports into three types based on repair outcomes: (1) log group: the corresponding repair task is only provided with a log output, including buggy statements and patches already tried. (2) incorrect patch group: the corresponding repair task is also provided with a patch output, containing some plausible patches, which are not semantically equivalent to developer patch. (3) correct patch group: the corresponding repair task is also provided with a patch output containing a correct patch, which is semantically equivalent to the developer patch.

As shown in Fig. 5(c), the log group has 73.3% (11/15) patches being correct, which is similar to the control group. The percentage of correct patches dramatically decreases to 23% for the incorrect patch group. On the contrary, the correct patch group performs much better than the above two groups, with about 88% (28/31) patches being correct. Meanwhile, the average debugging time for log group, incorrect patch group and correct patch group is 21.3, 17.0 and 19.8 minutes.

It is observed that the report types have a significant influence on the patch correctness, while debugging time is not affected.

5.2.4 Performance impacted by the patch quality

As discussed above, the repair performance increases when participants are aided by correct patches and decreases when participants are aided by incorrect patches. Based on these insights, we want to investigate whether participants can accurately identify the patch correctness and how long it takes to complete the identification step. Hence we require participants to answer questions about the given patch correctness (if provided in the repair task) and the identification time when they attempt to fix the bug.

Table 5 summarizes the identification results. Intuitively, each cell in the table is represented as $x/y$, where $x$ is the identification correctness and $y$ is the identification time. As shown in Table 5, all participants can identify 59% (26/44)
patches correctly using an average of 8.9 minutes per patch. The precision drastically increases to 91% (21/23) when they are provided with correct patches and decreases to 24% (5/21) when they are provided with incorrect patches. The identification time (7.3 minutes) for correct patches is 41% faster than incorrect patches (10.3 minutes).

Besides, among the three tools, patches generated by PraPR (9.8 minutes) consume more identification time than the ones generated by SimFix (8.4 minutes) and ACS (7.6 minutes). One possible reason is that PraPR dumps each plausible fix to a class file and the users need to take advantage of a Java bytecode decompiler to decompile the resulting class file, which might be time-consuming. Meanwhile, the precision for all tools with correct patches is 100%, except PraPR with 60%. Unlike the other tools, PraPR will provide all plausible patches to users (an average of 5 patches in our experiment), and it will be a little bit confusing to identify the patches as the amount increases.

Participants usually spend more debugging time on incorrect patches, while still easily misguided. Thus the accuracy of auto-generated patches is particularly important in practice. This observation urges a strict quality control for auto-generated patches if APR tools are used as debugging aids.

Answer to RQ2: Overall, our analysis on the assistance of APR tools reveals that (1) manual program repair can benefit from the assistance of APR tools in terms of the number of correctly-fixed bugs; (2) the tool types have little impact on the correctness and debugging time; (3) the report types have a significant influence on correctness and debugging time is not affected; (4) plausible but incorrect patches have adverse effects on the identification precision and identification time.

5.3 RQ3: What are participants’ opinions on APR tools

To qualitatively investigate participants’ opinions on employing APR tools, we conduct an online survey, where all participants are required to submit their free-form opinions on APR aids. Given that there is a lack of such human analysis in the existing literature, using the survey feedback in a practical debugging scenario is also meaningful. In total, we receive 102 textual answers for several well-designed questions from 85% of the participants.

Are APR tools useful for manual program repair? In general, 94.12% of participants think manual program repair can benefit from APR tools, while only 5.88% do not. However, less than half (about 47.06%) of participants declare that they are willing to use such tools in practice. This gap reveals an urgent problem that although these tools have been well designed regarding some criteria, their usability needs to be further improved for development.

Which patch generation strategy should APR tools adopt? As it is observed that the amount of provided patches may influence the identification performance, we also conduct a corresponding questionnaire. Overall, 58.82% of participants think it is suitable for tools to provide all plausible patches and the others prefer only one patch with the highest accuracy. This reveals APR tools need to achieve a trade-off between the accuracy and quantity of patches.

What information is preferred when APR tools cannot provide any plausible patch? So far, despite the success of recent APR tools, even the most advanced tool can only generate plausible patches for a small ratio of real bugs. In other words, manual program repair cannot be provided with any patches for the vast majority of bugs. Thus, we conduct a corresponding questionnaire to analyze what information is preferred when no plausible patch is provided. In the end, 70% of responses indicate a need for buggy location and patch execution information. Although such information is provided in log output, it may be contained in thousands of lines of documents. Participants may consume lots of time to extract key information, which is not worth compared to manual program repair completely unaided. This reveals using the feedback (i.e., the log output) of APR to refine an easy-to-read report for all possible bugs might be promising, where participants can know what attempts APR tools have made.

Participants’ positive and negative opinions on the assistance of APR tools. Participants freely elaborated their views on the advantages and disadvantages of APR tools. We divide them into six reasons why participants are positive and negative about APR tools as debugging aids after an open coding phase [7]. The summarization of these reasons along with participants’ original answers are presented at Table 6.

Specifically, participants acknowledge that APR tools are able to accelerate debugging process at the beginning, as the tools can identify suspicious code elements instantly without comprehending all source code. However, the assistance of APR tools may be costly because it is difficult to use the tools and extract the key contents from complex reports. Another shared concern is that such tools simply use trial and error to generate a patch without comprehending the
TABLE 6: Participants’ positive and negative opinions on APR tools

| Positive | Negative |
|----------|----------|
| **It is effective to accelerate debugging process initially.** | **It is difficult to get started at the beginning.** |
| P1: It helps to identify suspicious code elements quickly when developers are not familiar with the source code. | N1: It is difficult to adopt the tool at the beginning when developers are not familiar with it. |
| P2: Participants could immediately recognize the buggy code elements and acquire the plausible solution to the problem without understanding all of the source code. | N2: It is costly for developers to extract the content of the repair report. |
| P3: It can quickly identify buggy elements and further provide candidate fixes for developers to choose. | N3: An important point of the tool is to make the repair report easy for developers to understand. |
| P4: It is able to help me repair bugs faster. | N4: The provided patch is generated based on frequent trial and error, without understanding the functionality of the code. |
| **It can provide multiple suspicious buggy code elements.** | **It provide reports with a low accuracy.** |
| P5: It can guide me to identify potentially risky code elements. | N5: The accuracy of the repair report is not high, and it also suffer from poor readability and usability. |
| P6: I could be more likely to identify the buggy code elements when the patches are provided, despite most of them being incorrect. | N6: When developers are provided with the repair report, the accuracy should be improved. |
| P7: It provide multiple buggy code elements, and they are beneficial for me to repair the bug. | N7: The accuracy of buggy locations and the understandability of reports are low. |
| P8: It can help me to identify buggy code snippets. | N8: The accuracy of the tool is too low, and it even attempts to generate patches on code elements, which are obviously correct. |
| P9: It provides accurate buggy locations, which is convenient for me to understand the bug. | N9: For the tools that generate bytecode level patches, I hope the patches can be presented to developers in source-level to aid readability. |
| P10: It can identify the location where a bug may appear. | N10: When generating repair reports, tools should eliminate irrelevant information as much as possible to facilitate quick understanding. |
| **It can provide useful patches.** | **Its report is less understandable.** |
| P11: The key is that the tool can generates a usable patch. | N11: Tools should provide decompiled source-level patches if they are able to fix the bug, otherwise the suspiciousness value for each code element should be presented. |
| P12: It can always provide patches and suggest useful guidelines for repairing. | N12: Displaying abundant process data is of little significance to developers. |
| P13: It can provide useful suggestions about how to repair the bug, and sometimes it can even provided the correct patch. | N13: Some repair reports have complex content, in fact, providing the most critical information is enough. |
| P14: It can provide patches and buggy statements. | N14: Repair reports in bytecode format need to be decompiled in advance, as they are inconvenient for me to understand. |
| P15: It can indentify buggy statements and sometimes even provide plausible patches directly. | **Aperry report is less understandable.** |
| P16: It can provide me with plausible patches and suspicious buggy statements. | **Its report is less understandable.** |

source code, so as to generate some abnormal patches.

There is a popularly accepted opinion that such tools can provide some useful guidelines (i.e., suspicious code elements and candidate patches) on how to fix the bugs. However, provided guidelines do not guarantee that the debugging is going down the right path, since the reports can be confusing and misleading.

Interestingly, we also observe that almost half of the participants think the repair reports are not user-friendly. For example, it might be a little bit confusing for participants to read a given patch in bytecode-level and time-consuming to extract useful information from documents with too much redundant information. Hence some participants hope that irrelevant information can be eliminated to facilitate the extraction process.

Answer to RQ3: Overall, our questionnaire survey reveals that (1) there exists a huge gap between the repair performance in the benchmark and the usability in practice; (2) achieving a trade-off between the patch accuracy and patch quantity is important in repair reports; (3) the feedback in log output should be valued more in the future; (4) participants are positive about APR tools to provide useful guidelines about buggy locations and even fix solutions, while also less confident about the accuracy and format of repair reports.

6 Threats to validity

The selection of participants might be biased. Due to a monetary limitation, following existing work [54], [57], [74], [75], we recruit students instead of professional developers from industry, which may introduce a bias in our conclusions. To mitigate this threat, we select participants with varied experience (i.e., formal work experience, internship work experience and limited work experience). Meanwhile, Salmon et al. [76] report that both students and professional developers have similar performance for a new software engineering task. As such, we believe the selection strategy may not be a key point to our user study.

With respect to the representativeness of subjects, all of them are selected from Defects4J. It has been shown that APR tools may t the dataset in terms of repairability [17]. Although we focus on the comparison between automated and manual program repair, a bias may still be introduced. Thus, instead of adopting the entire dataset, we carefully select several representative buggy programs to mitigate the tting problem. Meanwhile, to mitigate the threat that the used bugs and APR tools may not be representative of all bugs and repair techniques, we randomly select eight real bugs with varied symptoms and bug types and three state-of-the-arts from all possible APR categories. As we have seen, this is the largest relevant study in APR field ever (e.g., there exist six bugs and two APR tools in [25]).
The final threat to validity is that participants might blindly reuse provided patches instead of really fixing bugs on their own, if plausible patches are provided in repair reports (i.e., the patch output). To prevent such behaviours, we emphasize in advance that provided patches may not be correct, and participants should make their own judgement. We also require all participants to judge the patch correctness (if provided) before submissions, and discover that there exists a similar debugging time cost between participants with or without provided patches (described in Section 5.2.4). It indicates that participants are less likely to reuse provided patches unconditionally, which may otherwise take only seconds to complete.

7 Practical Guidelines

Based on the observations in our experiment, we have learned essential aspects to consider when using APR tools in a real debugging scenario. Now we summarize our suggestions on improving the practicality of APR tools.

Repair report should be presented in a more user-friendly way. As discussed in Section 5.1, it is necessary to recheck generated patches for deployment due to the quality issue. Moreover, it would be misleading for participants to identify patch correctness if they are provided with all plausible patches. Thus, it is better to normalize automatically generated patches and achieve a trade-off between the accuracy and quantity of patch according to the practical situation.

Misleading repair information should be reduced. Misleading information (e.g., incorrect patches in Section 5.2.4) may delay the repair process as participants will spend a lot of time verifying such information. Meanwhile, half of the participants are less confident about the accuracy ratio of the provided repair reports. This can be done by designing better algorithms to provide accurate information. However, the remarkable progress of APR tools requires a long and continuous effort. In a similar area of fault localization, Le et al. [7] suggested predicting the accuracy of fault localization information before utilizing them, which can be adopted for APR. Also, we can combine multiple reports from different APR tools to provide developers with more accurate repair information.

The feedback in the log output should be observed. As discussed in Section 5.3, many participants consider it hard to extract significant information (e.g., buggy statements and fixes already tried) in the log output. A possible explanation is that these tools focus on generating as many plausible patches as possible in the patch output, and do not take other useful debugging hints in the log output into account. Thus, there exists lots of redundant information in the log output. In fact, it is observed that the feedback in the log output can provide useful guidelines for debugging [7]. It seems a promising direction to unify plausible patches in the patch output and execution information in the log output. Such a flexible debugging approach can always provide developers with refined debugging information for all possible bugs.

8 Conclusion

We conduct a large-scale human study to compare automated with manual program repair, and further investigate whether the assistance of APR tools can benefit manual program repair. Our experiment involving eight real bugs and 160 repair tasks indicates that manual program repair may be influenced by the frequency of repair actions. Besides, APR tools are able to improve manual program repair in terms of the number of correctly-fixed bugs, while the patch correctness may be adversely affected. Furthermore, it is confirmed that the assistance of APR tools is promising, while the accuracy and format of repair reports need improving. Based on these observations, some guidelines on improving the usability of existing APR tools (e.g., the misleading information in reports and the importance of feedback) are provided.

In the future, we will conduct a series of research from the suggested directions, to explore how to make APR tools consumable from the developer’s perspective. And further, it would be interesting to investigate how APR tools can be integrated into the regular developer workflow (e.g., IDE plugins).

Acknowledgments

This work is supported partially by National Natural Science Foundation of China (61932012).

References

[1] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” IEEE Transactions on Software Engineering, vol. 45, no. 1, pp. 34–67, Jan 2019.

[2] S. Planning, “The economic impacts of inadequate infrastructure for software testing,” National Institute of Standards and Technology, 2002.

[3] J. Anvik, L. Hiow, and G. C. Murphy, “Coping with an open bug repository,” in Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, 2005, pp. 35–39.

[4] J. Aranda and G. Venolia, “The secret life of bugs: Going past the errors and omissions in software repositories,” in 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009, pp. 298–308.

[5] U. Software, “Increasing software development productivity with reversible debugging,” https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf 2016.

[6] C. Boulder, “University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually,” https://www.roguewave.com/company/news/2013/university-of-cambridgereverse-debugging-study. 2013.

[7] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, “Can automated program repair refine fault localization? a unified debugging approach,” in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 75–87. [Online]. Available: https://doi.org/10.1145/3393563.3397351.

[8] 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, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 19–30. [Online]. Available: https://doi.org/10.1145/3293882.3330859.

[9] 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, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 789–799. [Online]. Available: https://doi.org/10.1145/3180135.3180182.
[10] M. Monperrus, “Automatic software repair: A bibliography,” ACM Comput. Surv., vol. 51, no. 1, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3105906.

[11] ——, “Resolving the living review on automated program repair,” IEEE/ACM Trans. Softw. Eng., vol. 44, no. 5, pp. 119–146, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2771409.

[12] Y. Xiong, J. Wang, Y. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017, pp. 416–426.

[13] J. Jiang, Y. Xiong, H. Zhe, 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, 2018, pp. 298–309.

[14] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” ACM Trans. Softw. Eng. Methodol., vol. 28, no. 4, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3340554.

[15] R. Just, D. Jalalifard, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 437–440.

[16] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, “On the efficiency of test-based program repair: A systematic assessment of 16 automated repair systems for java programs,” in 42nd ACM/IEEE International Conference on Software Engineering (ICSE), 2020.

[17] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, “Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 302–313.

[18] J. Bader, A. Scott, M. Pradel, and S. Chandra, “Getafix: Learning to fix bugs automatically,” Proc. ACM Program. Lang., vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360586.

[19] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, “Sapfix: Automated end-to-end repair at scale,” in Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ser. ICSE-SEIP ’19. IEEE Press, 2019, p. 269–278. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00039.

[20] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Atzori, “Amp: Deepbelief: learning to repair compilation errors,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 925–936.

[21] J. Yi, S. H. Tan, S. Mechtaev, M. Bohme, and A. Roychoudhury, “A correlation study between automated program repair and test-suite metrics,” Empirical Software Engineering, vol. 23, no. 5, pp. 2948–2979, 2018.

[22] Y. Tao, J. Kim, S. Kim, and C. Xu, “Automatically generated patches as debugging aids: A human study,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 64–74. [Online]. Available: https://doi.org/10.1145/2635866.2635873.

[23] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, “On the accuracy of spectrum-based fault localization,” in Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), Sep. 2007, pp. 89–98.

[24] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Mogenthaler, and J. Penix, “Using static analysis to find bugs,” IEEE Software, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[25] Y. W. Wong, R. Gao, Y. J. L. Abreu, and F. Wotawa, “A survey on software fault localization,” IEEE Transactions on Software Engineering, vol. 42, no. 4, pp. 707–740, Aug 2016.

[26] T. Ji, L. Chen, X. Mao, and X. Yi, “Automated program repair by using similar code containing fix ingredients,” in 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), IEEE, 2016, pp. 197–202.

[27] F. Logozzo and T. Ball, “Modular and verified automatic program repair,” SIGPLAN Not., vol. 47, no. 10, p. 133–146, Oct. 2012. [Online]. Available: https://doi.org/10.1145/2398857.2384626.

[28] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811.

[29] ——, “Automatic patch generation by learning correct code,” in Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 298–312. [Online]. Available: https://doi.org/10.1145/2838574.2838627.

[30] S. H. Tan and A. Roychoudhury, “Relifix: Automated repair of software regressions,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, May 2015, pp. 471–482.

[31] M. Kren, J. Chen, R. Wu, D. Hao, and S. Chen, “Context-aware patch generation for better automated program repair,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), May 2018, pp. 1–11.

[32] J. Xuan, M. Martinez, F. DeMarco, M. Clément, 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, Jan 2017.

[33] B. Samuel, L. Xia, L. Yiling, and Z. Lingming, “On the effectiveness of unified debugging: An extensive study on 16 program repair systems,” in 35th IEEE/ACM International Conference on Automated Software Engineering (ASE’20), 2020.

[34] Y. Tao, M. Wen, B. Lin, Y. Qin, D. Zou, X. Mao, and H. Jin, “Automated patch correctness assessment: How far are we?” in Proceedings of the 35th International Conference on Automated Software Engineering (ASE). ACM, 2020, pp. 1166–1178.

[35] 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.

[36] 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, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2910307.2910305.

[37] M. Martinez and M. Monperrus, “Ultra-large repair search space with automatically mined templates: The cardumen mode of astor,” in International Symposium on Search Based Software Engineering. Springer, 2018, pp. 65–86.

[38] T. Durieux and M. Monperrus, “Dynamoth: dynamic code synthesis for automatic program repair,” in Proceedings of the 11th International Workshop on Automation of Software Test, 2016, pp. 85–91.

[39] J. Xuan, M. Martinez, F. DeMarco, M. Clément, 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. 46, no. 10, pp. 1040–1067, 2018.

[40] M. Martinez and M. Monperrus, “Ultra-large repair search space with automatically mined templates: The cardumen mode of astor,” in International Symposium on Search Based Software Engineering. Springer, 2018, pp. 65–86.

[41] A. Koyuncu, K. Liu, T. F. Bisshandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, “Fixminer: Mining relevant fix patterns for
automated program repair,” Empirical Software Engineering, vol. 25, no. 3, pp. 1980–2024, 2020.

[47] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Tbar: revisiting template-based automated program repair,” in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 31–42.

[48] ———, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 1–12.

[49] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, “Sorting and transforming program repair ingredients via learning from similarities,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 479–490.

[50] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning,” in Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[51] 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, 2020, pp. 101–114.

[52] Y. Li, S. Wang, and T. N. Nguyen, “Dflix: Context-based code transformation learning for automated program repair,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 602–614.

[53] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 1161–1173.

[54] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in Proceedings of the 2011 international symposium on software testing and analysis, 2011, pp. 199–209.

[55] M. Weiser and J. Lyle, “Experiments on slicing-based debugging aids,” in Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers. USA: Ablex Publishing Corp., 1986, p. 187–197.

[56] Q. Wang, C. Parnin, and A. Orso, “Evaluating the usefulness of ir-based fault localization techniques,” in Proceedings of the 2015 International Symposium on Software Testing and Analysis, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–11. [Online]. Available: https://doi.org/10.1145/2771783.2771797

[57] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, “Revisit of automatic debugging via human focus-tracking analysis,” in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 808–819. [Online]. Available: https://doi.org/10.1145/2884781.2884834

[58] M. A. Francel and S. Rugaber, “The value of slicing while debugging,” Sci. Comput. Program., vol. 40, no. 2–3, p. 151–169, Jul. 2001. [Online]. Available [https://doi.org/10.1016/S0167-6423(01)00013-2]

[59] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE ’13. IEEE Press, 2013, p. 802–811.

[60] “Joda-time,” in https://github.com/JodaOrg/joda-time.

[61] L. Chen and L. Zhang, “Fast and precise on-the-fly patch validation for all,” arXiv preprint arXiv:2007.11449, 2020.

[62] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multi-lingual program repair benchmark set based on the quixy challenge,” in Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2017, pp. 55–56.

[63] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs.jar: a large-scale, diverse dataset of real-world java bugs,” in Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 10–13.

[64] F. Madeiral, S. Uri, M. Maia, and M. Monperrus, “Bears: An extensible java bug benchmark for automatic program repair studies,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 468–478.

[65] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–37, 2018.

[66] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, “A learning-to-rank based fault localization approach using likely invariants,” in Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 177–188.

[67] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017, pp. 609–620.

[68] Y. Qi, X. Yao, J. P. Li, and R. Wang, “Using automated program repair for evaluating the effectiveness of fault localization techniques,” in Proceedings of the 2013 International Symposium on Software Testing and Analysis, 2013, pp. 191–201.

[69] B. Miranda, E. Cruciani, R. Verdechcia, and A. Bertolino, “Fast approaches to scalable similarity-based test case prioritization,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 222–232.

[70] E. Cruciani, B. Miranda, R. Verdechcia, and A. Bertolino, “Scalable approaches for test suite reduction,” in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 419–429.

[71] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, “Dissection of a bug dataset: Anatomy of 395 patches from defects4j,” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018, pp. 130–140.

[72] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, “Has the bug really been fixed?” in 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, May 2010, pp. 55–64.

[73] T. K. Landauer, “Research methods in human-computer interaction,” in Handbook of human-computer interaction. Elsevier, 1988, pp. 905–928.

[74] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu, “Test report prioritization to assist crowdsourced testing,” in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 225–236.

[75] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, “Multi-objective test report prioritization using image understanding,” in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2016, pp. 202–213.

[76] I. Salman, A. T. Misirli, and N. Juristo, “Are students representatives of professionals in software engineering experiments?” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, 2015, pp. 666–676.

[77] T.-D. B. Le, D. Lo, and F. Thung, “Should i follow this fault localization tool’s output?” Empirical Software Engineering, vol. 20, no. 5, pp. 1237–1274, 2015.