Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair

KUI LIU, Nanjing University of Aeronautics and Astronautics, China
JINGTANG ZHANG, Nanjing University of Aeronautics and Astronautics, China
LI LI, School of Software, Beihang University, China
ANIL KOYUNCU, Sabanci University, Turkey
DONGSUN KIM, Kyungpook National University, South Korea
CHUNPENG GE, School of Software, Shandong University, China
ZHE LIU, Nanjing University of Aeronautics and Astronautics, China
JACQUES KLEIN, University of Luxembourg, Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg

Fix pattern-based patch generation is a promising direction in automated program repair (APR). Notably, it has been demonstrated to produce more acceptable and correct patches than the patches obtained with mutation operators through genetic programming. The performance of pattern-based APR systems, however, depends on the fix ingredients mined from fix changes in development histories. Unfortunately, collecting a reliable set of bug fixes in repositories can be challenging. In this paper, we propose investigating the possibility in an APR scenario of leveraging fix patterns inferred from code changes that address violations detected by static analysis tools. To that end, we build a fix pattern-based APR tool, Avatar, which exploits fix patterns of static analysis violations as ingredients for the patch generation of repairing semantic bugs. Evaluated on four benchmarks (i.e., Defects4J, Bugs.jar, BEARS, and QuixBugs), Avatar presents the potential feasibility of fixing semantic bugs with the fix patterns inferred from the patches for fixing static analysis violations, and can correctly fix 26 semantic bugs when Avatar is implemented with the normal program repair pipeline. We also find that Avatar achieves performance metrics that are comparable to that of the closely-related approaches in the literature. Compared with CoCoNut, Avatar can fix 18 new bugs in Defects4J and 3 new bugs in QuixBugs. When compared with HDRepair, JAID, and SketchFix, Avatar can newly fix 14 Defects4J bugs. In terms of the number of correctly fixed bugs, Avatar is also comparable to the program repair tools with the normal fault localization setting, and presents better performance than most program repair tools. These results imply that Avatar is complementary to current program repair approaches. We further uncover that Avatar can present different bug-fixing performances when it is configured with different fault localization tools, and the stack trace information from the failed executions of test cases can be exploited to improve the bug-fixing performance of Avatar by fixing more bugs with fewer generated patch candidates. Overall, our study highlights the relevance of static

*Li Li and Zhe Liu are the corresponding authors.

Authors' addresses: Kui Liu, kui.liu@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Jingtang Zhang, jingtangzhang@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Li Li, lilicoding@ieee.org, School of Software, Beihang University, Beijing, China; Anil Koyuncu, anil.koyuncu@sabanciuniv.edu, Sabanci University, Istanbul, Turkey; Dongsun Kim, darkrsw@knu.ac.kr, Kyungpook National University, Daegu, South Korea; Chunpeng Ge, gecp@nuaa.edu.cn, School of Software, Shandong University, Jinan, China; Zhe Liu, zhe.liu@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg.

1049-331X/2023/1-ART $15.00
https://doi.org/10.1145/3579637

bug-finding tools as indirect contributors of fix ingredients for addressing code defects identified with functional test cases (i.e., dynamic information).

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Software defect analysis*; Software testing and debugging.

Additional Key Words and Phrases: Automated program repair, static analysis, fix pattern

1 INTRODUCTION

The momentum of automated program repair (APR) keeps activating the development of various approaches in the literature [49]. In the software engineering community, the focus is mainly placed on fixing *semantic bugs* that make the program behavior deviate from developers' intentions [46, 50]. Such bugs could be detected by analyzing the execution traces of failing and passed tests. The research community has developed generate-and-validate repair pipelines [4, 16, 23, 25, 31, 62, 63, 68] where program test cases are leveraged not only for localizing the bug locations [1, 29, 52, 72] but also as the oracle for validating the generated patches [22, 37, 53, 67].

Unfortunately, given that test suites can be incomplete and one-sided, typical APR systems are prone to generate nonsensical patches that might violate the intended program behavior or simply introduce other defects which are not covered by test suites [18]. Smith *et al.* [58] have thoroughly investigated this issue and found that *overfitted* patches are common: these patches can make the patched program pass all the available test cases, but are not *correct*. Consequently, those incorrect patches might not be accepted by developers for further maintenance.

To address the problem of patch correctness in APR, our work focuses on how to generate better patches rather than adding more test cases. In the community, two research directions are being investigated. The first direction attempts to develop techniques for automatically augmenting the test suites [74]. The second one focuses on improving the patch generation process to reduce the probability of generating nonsensical patches [16, 21, 68].

Mining fix templates from common patches in the wild is a promising approach to achieve patch correctness. As first introduced by Kim *et al.* [18], patch correctness can be improved by leveraging fix templates learned from human-written patches. In their work, the template construction was performed manually, which is a limiting factor and is further error-prone [48]. Since then, several approaches have been developed towards automating the inference of fix patterns from fix changes in developer code bases [16, 28, 35, 44, 61]. For example, Liu et al. [27] and Rolim et al. [54] proposed mining fix patterns from the patches of fixing static analysis violations. They first leveraged the static analysis tools to analyze each commit version of a program, and compared the differences of detected static violations between two connected commit versions to identify which violations are fixed. The related patches for the fixed static violations are identified with the sophisticated algorithm (e.g., Avgustinov et al. [2]) and are further parsed into code change diffs[1] at the abstract syntax tree level with GumTree [7], a code change distilling tool. Such diffs are then used to infer and mine the corresponding fix patterns with deep learning techniques. Note that, a key challenging step in the inference of patterns, however, is the identification and collection of a substantial set of relevant bug fix changes to construct the learning dataset. Patterns must further be precise and diverse to guarantee repair effectiveness [31].

There have been approaches to mining fix patterns and exploring the challenges in achieving the diversity and reliability of fix ingredients but those approaches still have limitations. Long *et al.* [35] have relied on only three simple bug types, while Koyuncu *et al.* [21] have focused on bug linking between bug tracking systems and source code management systems to identify possible bug fixes. Unfortunately, the former approach cannot find patterns to address a variety of bugs, while the latter may include patterns that are irrelevant to bug fixes since

[1]A "code change diff" consists of two code snapshots. One snapshot represents the code fragment that will be affected by a code change, while the other one represents the code fragment after it has been affected by the code change.

developer changes are not atomic [12]. It is thus challenging to extract useful and reliable patterns focusing on fix changes.

Our work proposes a new direction for pattern-based APR to overcome the limitations in finding reliable and diverse fix ingredients. Concretely, we focus on fix patterns inferred from developer patches of fixing static analysis violations that are detected by static analysis tools and are referred to as warnings, alerts, or alarms (cf. Section 2.2). The advantages of this approach are: (1) the availability of toolsets for assessing whether a code change is actually a fix [2, 11], and (2) the ability to further pre-categorize the changes into groups targeting specific violations, leading to consistent fix patterns [27, 54].

Although static analysis violations (e.g., FindBugs[2] warnings) may appear irrelevant to the problem of *semantic* bug fixing, there are two findings in the literature, which can support our intuition of leveraging fix patterns from static analysis violation patches to address semantic bugs:

- *Locations of **semantic bugs** (unveiled through dynamic execution of test cases) can sometimes be detected by static analysis tools.* In a recent study, Habib *et al.* [10], have found that some bugs in the Defects4J dataset can be identified by static analysis tools: SpotBugs[3], Infer[4], and ErrorProne[5]. Other studies [8, 47, 73] have also suggested that violations reported by static analysis tools might be smells of more severe defects in software programs.
- *Violation fix patterns have been used to successfully fix bugs in the wild.* In preliminary live studies, Liu *et al.* [27] and Rolim *et al.* [54] have shown that they can systematically fix statically detected bugs by using some of their previously-learned fix patterns. They further showed that project developers are eager to integrate the systematization of such fixes based on the mined patterns.

Fix patterns of static analysis violations have been explored in the literature to automate patch generation for fixing statically-detected bugs. To the best of our knowledge, our work is the first attempt to leverage fine-grained fix patterns of static analysis violations as fix ingredients for automated program repair that addresses semantic bugs revealed by test cases.

This paper makes the following contributions:

(1) **We propose Avatar** (static **A**nalysis **V**iol**A**tion fix pa**T**tern-based **A**utomated program **R**epair), **a fix pattern-based approach to automated program repair.** Our approach differs from related work in the dataset of developer patches that is leveraged to extract fix ingredients. We build on patterns extracted from patches that have been verified (with bug detection tools) as true bug fix patches. Given the redundancy of bug types detected by static analysis tools, the associated fixes are intuitively more similar, leading to the inference of reliable common fix patterns. Avatar implements 28 fix patterns for 18 static violation types and is publicly available at: **https://github.com/mrdrivingduck/AVATAR**.

(2) **We explore the feasibility of repairing semantic bugs with the fix patterns of static analysis violations.** We first apply Avatar configured with the perfect fault localization to semantic bugs in four Java program defect benchmarks, to assess how many semantic bugs can be fixed with fix patterns extracted from common code changes fixing static analysis violations. Eventually, Avatar generates correct patches for 51 bugs, 11 bugs, 2 bugs, and 8 bugs in Defects4J, Bugs.jar, BEARS, and QuixBugs, respectively. It indicates that the fix patterns of static analysis violations can be used to fix semantic bugs. Then, we investigate how effectively Avatar can fix semantic bugs that appear to be localizable by static analysis tools. The experiments

[2]http://findbugs.sourceforge.net
[3]https://spotbugs.github.io
[4]https://fbinfer.com
[5]https://errorprone.info

show that Avatar is capable of correctly fixing 7 bugs that can be detected by static analysis tools. It presents the possibility of integrating Avatar with static analysis tools to solve program bugs.

(3) **We systematically investigate why the fix patterns of static analysis violations can be used to fix semantic bugs.** We dissect the bugs fixed by Avatar into eight categories in terms of the related violation type, and analyze the relationship between the behavior of the fix pattern and the issue of each bug. On the one hand, we observe that some semantic bugs are indeed caused by the same problem as the static violations. On the other hand, the fix patterns for static violations can be used to resolve the semantic bugs of which issues are different from the static violations, since they are fixed in the same way of changing code. Additionally, Avatar is capable of correctly fixing semantic bugs in a way that is different from but semantically similar to the code modifying method provided by developers in the ground-truth patches.

(4) **We compare the bug-fixing performance of Avatar against the program repair tools under different fault localization settings.** We compare our approach with the state-of-the-art based on different evaluation aspects, including the number of fixed bugs, the exclusivity of fixed bugs, patch correctness, etc. When configured with the perfect fault localization, Avatar can generate correct patches for 72 bugs, which outperforms the state-of-the-art deep learning-based program repair tool, CoCoNut [39]. When the buggy method can be correctly localized, Avatar can generate correct patches for 14 bugs that HDRepair, JAID, and SketchFix cannot fix. In the normal APR scenario, Avatar can fix 82 bugs in Defects4J, 26 of them are fixed with correct patches, which is comparable to the state-of-the-art program repair tools.

(5) **We assess to what extent the bug-fixing performance of Avatar could be biased by the different fault localization techniques.** We investigate the bug-fixing performance of Avatar in terms of the number of fixed bugs and the efficiency in terms of counting the number of generated patch candidates. Avatar configured with GZoltar-0.1.1 setting fixes more bugs than it configured with GZoltar-1.7.2, but the former will generate more nonsensical patches than the latter, we conclude that a trade-off between the number of fixed bugs and the efficiency of generating patch candidates should be well maintained when selecting the adequate fault localization for Avatar since different fault localization tools will report different suspicious statements to expose the bug positions.

(6) **We investigate the possibility of using the stack trace information to improve the bug-fixing performance of Avatar.** When using JUnit, stack traces are available for the ordinary failing-executed test case(s) as developers often check test verdict by using assertions (e.g., `assertTrue()`) that produce errors with stack traces if the test condition is not satisfied. Our experimental results uncover that the information in stack traces produced by bug-triggering test cases can be used to improve the bug-fixing performance of Avatar on fixing more bugs with less nonsensical patches.

2 BACKGROUND

In this section, we provide the background information on the general pattern-based APR, as well as on the pattern inference from the static analysis violation data.

2.1 Automated Program Repair with Fix Patterns

Pattern-based APR has been widely explored in the literature [18, 24, 29, 31, 35, 38, 56, 60]. The basic idea is to represent common code changes into a *pattern* (interchangeably referred to as a *template*) that can be applied to faulty code (i.e., patching). The fixing process consists of leveraging context information of faulty code (e.g., abstract syntax tree (AST) nodes) to match context constraints defined in a given fix pattern. For example, the fix template "Method Replacer" provided in PAR [18] is presented as:

`obj.`**`method1`**`(param)` → `obj.`**`method2`**`(param)` where the faulty method call `method1` is replaced by another method call `method2` with compatible parameters and return type. A method call is the context information for this template to match the buggy code fragment. Thus, this template can be applied to any faulty statement that includes at least one method call expression. The template further guides the patch candidate generation where changes are proposed to replace the potentially faulty method call with another method call.

Mining fix patterns has some intrinsic issues. The first issue relates to the variety of patterns that must be identified to support the fixing of different bug types. There are three strategies in fix pattern mining: (1) manual design, (2) automatic mining, and (3) code change statistics. The first strategy can effectively create precise fix patterns. Unfortunately, it requires human effort, which can be prohibitive [18]. The second one infers common modification rules [35] or searches for the most redundant sub-patch instance [16, 21]. The last one selects the top-n most frequent code change instructions (at the abstract syntax tree level) as fix patterns [16, 24, 63], but it relies on the quality of bug-fixing commits collected from the maintaining history of programs. While the latter two strategies can substantially increase the number of fix patterns, it is subject to noisy input data due to *tangled changes* [12], which make the inferred patterns less relevant. The second issue relates to the granularity (i.e., the degree of abstraction). Coarse-grained and monolithic patterns [51] can cover many types of bugs but they may not be actionable in APR. A fine-grained or micro pattern [35] can be readily actionable, but cannot cover many defects.

2.2 Static Analysis Violations

Static analysis tools help developers check for common programming errors in software systems. The targeted errors include syntactic defects, security vulnerabilities, performance issues, and bad programming practices. These tools are classified as "static" because they do not require dynamic execution traces to find bugs. Instead, they are directly applied to source code or bytecode. In contrast to dynamic analysis tools, which must run test cases, static tools can cover more paths, although it makes over-approximations that make them prone to false positives.

Many software projects rigorously integrate static analysis tools into their development cycles. The Linux kernel development project is such an example project where developers systematically run static analyzers against their code before pushing it to maintainers repositories [20]. More generally, FindBugs, PMD[6], and Google Error-Prone are often used in Java projects, while C/C++ projects tend to adopt Splint[7], cppcheck[8], and Clang Static Analyzer[9].

Static analysis tools raise *warnings*, which are also referred to as *alerts*, *alarms*, or *violations*. Given that these warnings are due to the detection of code fragments that do not comply with some analysis rules, in the remainder of this paper we refer to the issues reported by static analysis tools as *violations*. Figure 1 excerpted from [27] shows an example patch for a violation detected by FindBugs. This violation (the code starting with "−") is reported because the `equals` method should work for all object types (i.e., `Object`): in this case, the method code violates the rule since it assumes a specific type (i.e., `ModuleWrapper`).

Note that, not all violations are accepted by developers as actual defects. Since static analysis tools use limited information, detected violations could be correct code (i.e., false positive) or the warning may be irrelevant (e.g., cannot occur at runtime, or not a serious issue). In the literature, many studies assume that a violation can be classified as actionable if it is discarded after a developer changed the location where the violation is detected. The violation in Figure 1 is fixed by adding an `instanceof` check (c.f., the code starting with "+" in the patch diff); this violation can thus be regarded as *actionable* since this violation is gone after fixing its source code.

[6]https://pmd.github.io
[7]https://www.splint.org
[8]http://cppcheck.sourceforge.net
[9]https://clang-analyzer.llvm.org

```
// Violation Type (SpotBugs):
// BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS
// Equals method should not assume anything about the type of its argument.

 public boolean equals(Object obj) {
- return getModule().equals(((ModuleWrapper) obj).getModule());
+ return obj instanceof ModuleWrapper && getModule().equals(((ModuleWrapper)obj).getModule());
 }
```

Fig. 1. Example patch excerpted from [27] for fixing a violation detected by FindBugs.

Intuition. Mining patterns from developer patches that fix static analysis violations may help overcome the issues of fix pattern mining described in Section 2.1. First, since static analysis tools specify the type of each violation (e.g., bug descriptions[10] of FindBugs), each bug instance is already classified as long as it is fixed by code changes. Thus, we can reduce the manual effort to collect and classify bugs and their corresponding patches for fix pattern mining. Second, it is able to mitigate the issue of tangled changes [12] because violation-fixing changes can be localized and isolated by static analysis tools [2]. In addition, the granularity of fix patterns can be appropriately adjusted for each violation type since static analysis tools often provide information on the scope of each violation instance.

3 MINING FIX PATTERNS FOR STATIC VIOLATIONS

Mining fix patterns for static analysis violations has been explored in the literature [27, 54]. The general objective so far, however, is to learn quick fixes for speeding maintenance tasks and towards understanding which violations are prioritized by developers for fixing. To the best of our knowledge, our work is the first reported attempting to investigate fix patterns of static analysis violations in the context of automated program repair (where patches are generated and validated systematically with developer test cases).

There have been two recent studies of mining fix patterns addressing static analysis violations. Our previous study [27] focuses on identifying fix patterns for FindBugs violations [13], while Rolim *et al.* [54] consider PMD violations[11]. Both approaches, which were developed concurrently, leverage a similar methodology in the inference process. We summarise below the process of fix pattern mining of static analysis violations into three basic steps (as shown in Figure 2): data collection, data preprocessing, and fix pattern mining. Implementation details are strictly based on the approach of our previous study [27].

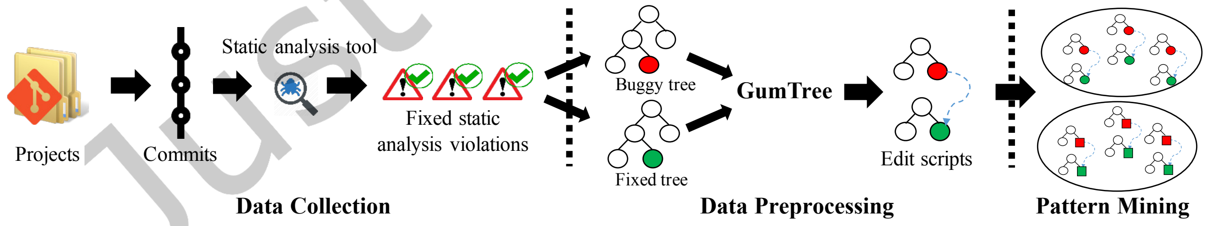


Fig. 2. Summarized steps of static analysis violation fix pattern mining.

3.1 Data Collection

The objective of this step is to collect patches that are relevant to static analysis violations that are detected by the static analysis tool, FindBugs [13]. This step is done in the wild based on the commit history of open-source

[10]http://findbugs.sourceforge.net/bugDescriptions.html
[11]https://pmdapplied.thomasleecopeland.com

projects by implementing a strict strategy to limit the dataset of changes to those that are relevant in the context of static analysis violations. To that end, it is necessary to systematically run static bug detection tools for each and every revision of the programs. This process can be resource-intensive: for example, FindBugs takes as input compiled versions of Java classes, requiring to build thousands of project revisions.

This step collects code changes (i.e., patches) only if they are identified as violation-fixing changes. For a given violation instance, we can assume that a change commit is a (candidate) fix for the instance when it disappears after the commit: i.e., the violation instance is identified in a revision of a program, but is no longer identified in the consecutive revision. Then, it is necessary to figure out whether the change actually fixed the violation instance or it just disappears by coincidence. If the affected code lines are located within the code change diff of the commit, it is regarded as an actual fix for the given violation instance. Otherwise, the violation instance might be removed just by deleting a method, class, or even a file. Eventually, all code change diffs associated with the identified fixed violation instances are collected to form the input data for fix pattern mining. We refer the reader to more details in [27].

3.2 Data Preprocessing

Once violation patches are collected, they are processed to extract concrete change actions. Patches submitted to program repositories are presented in the form of line-based GNU diffs where changes are reported in a text-based format of edit script. Given that, in modern programming languages, such as Java, source code lines do not represent a semantic entity of a code entity (e.g., a statement may span across several lines), it is challenging to directly mine fix patterns from GNU diffs.

Pattern-mining studies leverage edit scripts of program Abstract Syntax Trees (ASTs). Concretely, the buggy version (i.e., program revision file where the violation can be found) and the fixed version (i.e., consecutive program revision file where the violation does not appear) are given as inputs to the GumTree [7], an AST-based code differencing tool, to produce the relevant AST edit script. This edit script describes the repair actions that are implemented in the patch in a fine-grained manner. Figure 3 provides an example GNU Diff for a bug fix patch, and Figure 4 illustrates the associated AST edit scripts.

```
--- a/src/com/google/javascript/jscomp/Compiler.java
+++ b/src/com/google/javascript/jscomp/Compiler.java
@@ -1283,4 +1283,3 @@
   // Check if the sources need to be re-ordered.
   if (options.dependencyOptions.needsManagement() &&
-              !options.skipAllPasses &&
              options.closurePass) {

// Defects4J Dissection:
// Repair Action: Conditional expression reduction.
```

Fig. 3. Patch of the bug *Closure-31*[12] in Defects4J.

3.3 Fix Pattern Mining

Given a set of edit scripts, the objective of the pattern mining step is to group "similar" scripts to infer a common subset of edit actions, i.e., a consistent pattern across the group. To that end, Rolim *et al.* [54] rely on the greedy algorithm to compute the distance among edit scripts. Edit scripts with low distances among them are grouped together. Our previous study [27], on the other hand, leverages a deep representation learning framework (namely, CNNs [45]) to learn features of edit scripts, which are then used to find clusters of similar edit scripts. Clustering

[12]http://program-repair.org/defects4j-dissection/#!/bug/Closure/31

```
UPD IfStatement@@"if statement code"
---UPD InfixExpression@@"infix-expression code"
------DEL PrefixExpression@@"!options.skipAllPasses"
------DEL Operator@@"&&"
```

Fig. 4. AST edit scripts produced by GumTree for the patch in Fig. 3.

is performed based on the X-means algorithm. Finally, the largest common subset of edit actions among all edit scripts in a cluster is considered as a pattern.

Mined fix patterns with this approach have already been proven useful by the authors. For example, our previous study [27] and Rolim's work [54] conducted live studies by making pull requests to projects in the wild: the pull requests contained change details of a patch that is generated based on the inferred fix patterns to fix static analysis violations in developer code. Developers accepted to merge 67 out of 116 patches generated for FindBugs violations in our previous study [27]. Similarly, 6 out of 16 pull requests by Rolim *et al.* [54] have been merged by developers in the wild. Such promising results demonstrated the possibility to automatically fix bugs that are addressed by static bug detection tools.

4 OUR APPROACH

As shown in Figure 5, AVATAR consists of four major steps for automated program repair: fault localization, fix pattern matching, patch generation, and patch validation. In this section, we detail the objective and design of each step, and provide concrete information on implementation.

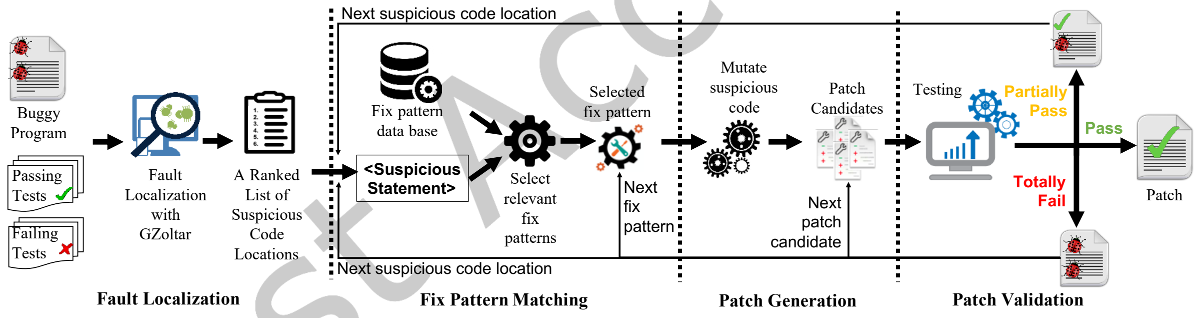


Fig. 5. Overview bug fixing process with AVATAR.

4.1 Fault Localization

We rely on the GZoltar[13] [3] framework to automate the execution of test cases for each program. In the framework, we leverage the Ochiai [1] ranking metric to actually compute the suspiciousness scores of statements that are likely to be the faulty code locations. This ranking metric has been demonstrated in several empirical studies [52, 59, 65, 71] to be effective for localizing faults in object-oriented programs. The GZoltar framework for fault localization is also widely used in the literature of APR [16, 21, 31, 32, 42, 63, 66, 68, 70], allowing a fair assessment for AVATAR's performance against the state-of-the-art APR tools.

[13]http://www.gzoltar.com

4.2 Fix Pattern Matching

In the execution of the repair pipeline, once fault localization produces a list of suspicious code locations, Avatar iteratively attempts to match each of these locations with a given pattern from the database of fix patterns. As illustrated in Algorithm 1, each suspicious statement is parsed in terms of AST to extract its context that is used to select the adequate fix pattern. With the selected fix pattern, if a valid patch that can make the patched program pass all tests [34] is generated by Avatar for the suspicious statement, Avatar will be terminated (cf. lines 7 and 9). Otherwise, Avatar keeps matching adequate fix patterns for other suspicious statements (cf. line 15) until a valid patch is generated, or all suspicious statements are trialed. To cease the process of patch generation when a valid patch cannot be generated, in case of the endless process of fixing bugs, Avatar will be stopped when the quantity of generated patches exceeds the maximum patch generation number of Avatar for one program (i.e., 5000 patch candidates) by considering the bias of bug-fixing performance and efficiency from the fault localization (e.g., top-k suspicious statements reported by fault localization tools).

Additionally, a bug could have multiple buggy positions [57] located in single or several code files, such as the Defects4J bug *Chart-19* shown in Figure 6. To fix such kind of bugs, Avatar considers each buggy position as a single bug to match a fix pattern for it. If a patch can make the buggy program pass some previously failed test without generating new failed test cases, it is considered as a partially valid patch for the buggy program and Avatar keeps matching adequate fix patterns for other suspicious statements (cf. lines 10, 12, and 13) based on this partially valid patch.

```
--- a/source/org/jfree/chart/plot/CategoryPlot.java
+++ b/source/org/jfree/chart/plot/CategoryPlot.java
@@ -695,7 +695,10 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
      * @since 1.0.3
      */
     public int getDomainAxisIndex(CategoryAxis axis) {
+        if (axis == null) {
+            throw new IllegalArgumentException("Null 'axis' argument.");
+        }
         return this.domainAxes.indexOf(axis);
     }

     /**
@@ -970,7 +970,10 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
      * @since 1.0.7
      */
     public int getRangeAxisIndex(ValueAxis axis) {
+        if (axis == null) {
+             throw new IllegalArgumentException("Null 'axis' argument.");
+        }
    int result = this.rangeAxes.indexOf(axis);
         if (result < 0) { // try the parent plot
             Plot parent = getParent();
             if (parent instanceof CategoryPlot) {
```

Failing Test:
```
org.jfree.chart.plot.junit.CategoryPlotTests:testGetDomainAxisIndex
org.jfree.chart.plot.junit.CategoryPlotTests:testGetRangeAxisIndex
```

Fig. 6. Patch of bug *Chart-19* by fixing two buggy positions.

Fix patterns in our database are collected from the artifacts released by Liu *et al.* [27] and Rolim *et al.* [54]. Table 1 shows statistics about the pattern collection in these previous works. As most of the fix patterns released by Liu *et al.* [27] will not change the program behavior, we only select 28 of them (released by Liu *et al.* [27]) for 18 violation types after manually checking that they can change the program behavior (details shown in the aforementioned website).

Algorithm 1: Fix pattern matching and patch generation.

```
Input  : $prog$, a buggy program.
Input  : $S$, a set of suspicious statements.
Input  : $FP$, a set of pre-defined fix patterns.
Output : $patch$, a valid patch or null.
Function main ($s$,$FP$)
    $patch$ := null ;                                   /* Initialise the patch as null */
    foreach $s \in S$ do
        /* Generate a patch by fixing statement s.                                  */
        $patch \leftarrow$ fixStatement ($s$, $FP$) ;
        if $patch$ != $null$ then
            if isValidPatch ($patch$) then
                /* A valid patch is generated by fixing statement s, and Avatar stops.  */
                return $patch$;
            else
                /* A partially valid patch is generated by fixing statement s. Patching the buggy
                   program prog with patch.                                              */
                $prog \leftarrow patch$;
                next $s$;
        else
            next $s$;
/* Match fix pattern and generate patches for the suspicious statement s.           */
Function fixStatement ($s$, $FP$)
    /* Parse the suspicious statement s.                                             */
    $s$.contextAST $\leftarrow$ parseStatementIntoAST ($s$) ;
    foreach $fp \in FP$ do
        if matchFixPattern ($s$.contextAST, $fp$.contextAST) then
            $patches \leftarrow$ generatePatches ($s$, $fp$);
            foreach $p \in patches$ do
                if isValidPatch ($p$) or isPartiallyValidPatch ($p$) then
                    return $patch \leftarrow p$;
                else
                    next $p$;
```

Table 1. Statistics on fix patterns of static analysis violations.

	# Projects	# violation fix patches	# violation types	# fix patterns
Liu *et al.* [27]	730	88,927	111	174
Rolim *et al.* [54]	9	288,899	9	9

```
// Fix Pattern: Remove a useless sub-predicate expression.
UPD IfStatement
---UPD InfixExpression@expA Operator expB
------DEL Expression@expB
------DEL Operator
```

Fig. 7. A fix pattern for UC_USELESS_CONDITION[14] violation [27].

Recall that each pattern is an edit script of repair actions on specific AST node types. AST nodes associated with the faulty code locations are then regarded as the *context* of matching the fixing patterns: i.e., these nodes are checked against the nodes involved in the edit scripts of fix patterns. For example, the fix pattern shown

[14]The condition has no effect and always produces the same result as the value of the involved variable was narrowed before. Probably something else was meant or condition can be removed.

in Figure 7 contains three levels of contexts: (1) `IfStatement`[15] means that the pattern is matched only if the suspicious faulty statement is an `IfStatement`; (2) `InfixExpression`[16] indicates that the pattern is relevant when the predicate expression of the suspicious `IfStatement` is an `InfixExpression`; (3) the matched `InfixExpression` predicate in the suspicious statement must contain at least two sub-predicate expressions.

A pattern is found to be relevant to a faulty code location only if all AST node contexts at this location matches with the AST node of the pattern. For example, the bug shown in Figure 3 is located within an `IfStatement` with an `InfixExpression` which is formed by three sub-predicate expressions. This buggy fragment thus matches the fix pattern shown in Figure 7.

4.3 Patch Generation

Given a suspicious statement and an associated matching fix pattern, Avatar applies the repair actions in the edit scripts of the pattern to generate patch candidates (cf. line 22 in Algorithm 1). For example, the code change action of the fix pattern in Figure 7 is interpreted as removing a sub-condition expression (or sub-predicate expression) in a faulty `IfStatement`. Thus, three patch candidates, as shown in Figure 8, can be generated by Avatar for the buggy code in Figure 3 since the statement has three candidate sub-predicates expressions.

```
  // Patch Candidate I.
- if (options.dependencyOptions.needsManagement() &&
-               !options.skipAllPasses &&
+ if (!options.skipAllPasses &&
               options.closurePass) {

  // Patch Candidate II.
  if (options.dependencyOptions.needsManagement() &&
-               !options.skipAllPasses &&
               options.closurePass) {

  // Patch Candidate III.
  if (options.dependencyOptions.needsManagement() &&
-               !options.skipAllPasses &&
-                options.closurePass) {
+               !options.skipAllPasses) {
```

Fig. 8. Patch Candidates generated by Avatar with a fix pattern that is mined from patches for `UC_USELESS _CONDITION` violations (cf. Fig. 7), and which matches the buggy statement of bug *Closure-13* (cf. Fig. 3).

4.4 Patch Validation

Patch candidates generated by Avatar must be systematically assessed. Eventually, using test cases, our approach verifies whether a patch candidate is a *valid* patch or not. We target two types of valid patches:

- ***Fully-fixing*** patches, which are patches that make the program pass all available test cases. Once such a patch is validated, the execution iterations of Avatar are halted.
- ***Partially-fixing*** patches, which are patches that make the program pass not only all previously-passing test cases, but also part of the previously-failing test cases.

The first generated *fully-fixing* patch is prioritized over any other generated patch, and is considered as the valid patch for the given bug. After iterating over all suspicious statements with all matching fix patterns, if Avatar fails to generate a *fully-fixing* patch for a bug, but generates some *partially-fixing* patches, these patches are considered as valid patches. Finally, the correctness of all valid patches is further manually and systematically

[15]https://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/IfStatement.html
[16]https://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/InfixExpression.html

assessed by following the protocol presented by Liu *et al.* [34]: (1) *identical patches*, the two patches are exactly identical, excluding variations in whitespace, layout, and comments; and (2) *semantically-similar patches*, the patches are not identical, but developers regard that they have the same effect on the program behavior.

4.5 Fixing Bugs with Multiple Buggy Positions

A bug could involve multiple buggy positions, and such bugs can be summarized into two categories with their bug-triggering test cases: (1) the multiple buggy positions are triggered by the same test cases, e.g., the Defects4J bug *Math-67* shown in Figure 9 with two buggy positions that are triggered by the same test case `testQuinticmin`, and (2) the multiple buggy positions are triggered by the independent test cases respectively, e.g., the Defects4J bug *Chart-19* shown in Figure 6 with two buggy positions that are triggered with the test cases `testGetDomainAxisIndes` and `testGetRangeAxisIndex`, respectively.

```
--- a/src/main/java/org/apache/commons/math/optimization/MultiStartUnivariateRealOptimizer.java
+++ b/src/main/java/org/apache/commons/math/optimization/MultiStartUnivariateRealOptimizer.java
@@ -90,9 +90,9 @@ public MultiStartUnivariateRealOptimizer(final UnivariateRealOptimizer optimizer
                  /** @inheritDoc */
                  public double getFunctionValue() {
-                     return optimizer.getFunctionValue();
+                     return optimaValues[0];
                  }

                  /** @inheritDoc */
                  public double getResult() {
-                     return optimizer.getResult();
+                     return optima[0];
                  }
```

Failing Test
`org.apache.commons.math.optimization.MultiStartUnivariateRealOptimizerTest:testQuinticMin`

Fig. 9. Defects4J bug *Math-67* fixed with two buggy positions.

When Avatar fixes the bug with multiple buggy positions, it relies on the independent bug-triggering test cases. For example, a bug $b = b_1, b_2$, where b_1 and b_2 are two buggy positions of the bug b, which can be triggered by two independent test cases t_1 and t_2, respectively. When Avatar fixes the bug b, it first generates the patch p_1 for b_1 if p_1 can make the patched bug b pass the execution of test case t_1 and previously successfully executed test cases. Then Avatar will generate the patch p_2 for b_2 in the same way. So, with this mechanism, Avatar is capable of fixing bugs with multiple buggy positions triggered by the independent test cases respectively, but cannot solve bugs with mutliple buggy poositions failing to execute the same test cases, e.g., the bug Math-67 shown in Figure 9.

5 SETUP FOR ASSESSMENT

5.1 Research Questions

Our investigation into the repair performance of Avatar seeks to answer the following research questions (RQs):

- **RQ1:** ***How effective are fix patterns of static analysis violations for repairing programs with semantic bugs?*** Recall that we broadly consider as *semantic bugs* all bugs that are uncovered by executing developer *test cases*. Our first research question assesses how many benchmark bugs can be fixed with fix patterns extracted from common code changes fixing static analysis violations. To that end, (1) we first investigate whether Avatar can generate a correct patch (i.e., a valid patch fixes the related bug as expected by its developers) to fix a semantic bug, when Avatar is configured with the perfect fault localization (i.e., the developer-provided fix location) to avoid the negative impact of the incorrect fault localization. Then, (2) we investigate how effectively Avatar can fix such semantic bugs that appear to be localizable by static analysis tools, to explore the possibility of integrating Avatar with static analysis tools to solve program bugs.

- **RQ2:** ***Which bugs and which patterns are effective targets of Avatar in an automated program repair scenario?*** This research question dissects the data yielded during the investigation of RQ1, with the objective of assessing the diversity of bugs that can be fixed as well as the types of violation fix patterns that have been successfully leveraged.
- **RQ3:** ***How does Avatar compare to the state-of-the-art with respect to repair performance?*** With this research question, we aim at showing whether the proposed approach is effective in the landscape of APR systems. Does Avatar offer comparable performance? To what extent can Avatar complement existing APR systems?
- **RQ4:** ***To what extent the bug-fixing performance and efficiency of Avatar could be impacted by different fault localization techniques?*** As reported by Liu et al. [29], the fault localization configuration in APR tools could impact the bug-fixing performance of ARP tools. This research question is to investigate the impact of the different fault localization techniques in the normal program repair pipeline.
- **RQ5:** ***Could the stack trace information be used to accelerate the bug-fixing performance of Avatar?*** In Java programs using JUnit, normal failing test cases always result in crashes since JUnit assertions produce exceptions if its condition is not satisfied. The failing execution of some test cases can lead to crashes with exceptions being thrown, where crashed statements will be enumerated in the corresponding stack trace. When developers fix such bugs manually, they will firstly attempt those crashed statements before others. Therefore, this research question aims to explore the possibility of improving the bug-fixing performance of Avatar with the stack trace information from failed executions of test cases.

5.2 Subjects

We evaluate Avatar on Defects4J [17], Bugs.jar [55], QuixBugs [75] and BEARS [40], which have been used by state-of-the-art APR systems targeting Java programs. Table 2 summarizes the statistics on the number of bugs available in version 2.0.0[17] of Defects4J, Bugs.jar, BEARS, and QuixBugs. "**Projects**" and "**Bugs**" denote respectively the Java projects the bugs belong to, and the number of bugs in each project.

Table 2. Subjects used in our experiments.

	Projects	Bugs		Projects	Bugs
Defects4J-v2.0.0	JFreeChart (Chart)	26		Jsoup	93
	Apache commons-cli (Cli)	39		Mockito	38
	Closure compiler (Closure)	174		Joda-Time (Time)	26
	Apache commons-codec (Codec)	18	Bugs.jar	Apache Accumulo	98
	Apache commons-collections (Collections)	4		Apache Camel	147
	Apache commons-compress (Compress)	47		Apache Commons-math	147
	Apache commons-csv (Csv)	16		Apache Flink	70
	Google Gson (Gson)	18		Apache Jackrabbit Oak	270
	FastXML jackson-core (JacksonCore)	26		Apache Log4J2	81
	FastXML jackson-databind (JacksonDatabind)	112		Apache Maven	48
	FastXML jackson-dataformat-xml (JacksonXml)	6		Apache Wicket	289
	Apache commons-jxpath (JxPath)	22		BEARS	251
	Apache commons-lang (Lang)	64		QuixBugs	40
	Apache commons-math (Math)	106		**Total**	2,284

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

The version 2.0.0 of Defects4J [17] includes 835 bugs from 17 open-source Java projects. These bugs are collected by identifying the bug-fixing commits from the maintaining history of programs, according to leveraging the bug tracking system (e.g., JIRA[18]) and executing test suites on the bug-fixed versions of programs and their related buggy versions. The benchmark has been widely used in the community of automated program repair [33, 34]. Bugs.jar [55] collects 1,158 bugs from 8 Apache projects, which is created using the same approach as Defects4J. Its main contribution is that it has more bugs than Defects4J. BEARS [40] contains 251 bugs from 72 different GitHub Java open-source projects, which is built by mining program repositories with the commit building state from Travis Continuous Integration. Compared with the previous two benchmarks, BEARS has a larger diversity of projects. QuixBugs [75] contains 40 single-line bugs from 40 programs. Each program implements one well-known algorithm, such as Quicksort. It is a benchmark in-the-lab, different from the previous three benchmarks collecting bugs from real-world programs.

5.3 Experimental Setup

Computing Environment: all our experiments are performed on a PC running Ubuntu 18.04 LTS with Intel Core i5-9400 2.90GHz CPU (six cores) and 32GB RAM, and a laptop running Ubuntu 20.04 LTS with Intel Core i7-7700HQ 2.80GHz CPU (eight cores) and 16GB RAM.

Fault Localization: for evaluation purposes, we apply different fault localization settings to the experiment of each research question, while the default setting of AVATAR is to use the GZoltar framework with the Ochiai ranking metric. The usage of GZoltar and Ochiai reduces the comparison biases since both are widely used by APR systems in the literature.

- First, we apply AVATAR configured with the perfect fault localization setting to the semantic bugs in four benchmarks (for RQ1 in Section 6.1). This configuration is to reduce the bias given by fault localization [29] and to assess the effectiveness of fixing semantic bugs with the fix patterns in AVATAR.
- Second, we apply AVATAR to bugs in the benchmarks with the location information of static analysis violations detected by three state-of-the-art static analysis tools (namely, SpotBugs, Facebook Infer, and Google ErrorProne) to answer RQ1 in Section 6.1. To that end, we consider the bugs detected by the four static analysis tools that are reported by Habib and Pradel [10]. This configuration focuses on the effectiveness of AVATAR on such semantic bugs that can also be detected statically.
- Third, for RQ3 and RQ4, we compare AVATAR with the state-of-the-art APR tools that are evaluated on the Defects4J benchmark (see Section 6.3). To that end, we attempt to replicate two scenarios of fault localization used in APR assessments: the first scenario assumes that the faulty method name is known [24] and thus focuses on ranking the inner-statements based on Ochiai suspiciousness scores; the second scenario makes no assumption on fault location and thus uses the default setting of AVATAR.
- Finally, in RQ5, we propose two principles with the stack trace information to refine the fault localization results of GZoltar for AVATAR, then to investigate the possibility of improving bug-fixing performance for AVATAR with the stack trace information.

Timeout: note that a generated patch may introduce new defects, e.g., endless loop, etc., making the patch validation process time-consuming. Here we configure AVATAR to validate each patch at most 5 minutes. If patch validation times out, AVATAR will stop validating this patch and turn to the next. If a valid patch, which can make the patched buggy program pass all test cases, is generated, AVATAR will be terminated. Otherwise, AVATAR will be halted when more than 5,000 patch candidates are generated by referencing a recent empirical study for APR efficiency [34].

[18]https://www.atlassian.com/software/jira

6 ASSESSMENT

6.1 Applying Avatar to Semantic Bugs

In this work, Avatar is implemented with the fix patterns inferred from the patches of static analysis violations that are identified by static analysis tools without executing any test cases, which are different from the semantic bugs that failed to pass test cases. Before assessing the bug fixing performance of Avatar in the normal program repair pipeline (i.e., the bug positions are localized with a specific fault localization technique, cf. Section 6.3.2), we propose to investigate whether the fix patterns inferred from the patches for fixing static analysis violations can be used to fix the semantic bugs that are failing to pass the concrete functional test(s) without the impact from the fault localization [29, 34]. Concretely, we first apply Avatar, configured with the perfect fault localization assumption, to the bugs in four benchmarks (i.e., Defects4J, Bugs.jar, BEARS, and QuixBugs) released in the community. Then, we further apply Avatar to the bugs that can be localizable by static analysis tools, to explore the possibility of integrating Avatar with static analysis tools to address program bugs.

Applying Avatar to Semantic Bugs in Four Benchmarks. We run Avatar on all bugs in four datasets (i.e., Defects4J, Bugs.jar, BEARS, and QuixBugs). As the objective is to assess whether a correct patch can be generated for the semantic bug with Avatar, it is configured with the perfect fault localization setting in this experiment scenario. Table 3 details the number of bugs in four benchmarks that are fixed by Avatar. Fully and partially fixed bugs are fixed with *fully-fixing* and *partially-fixing* patches (c.f. Section 4.4) generated by Avatar, respectively. Overall, Avatar can fix 107 bugs with valid patches, and 72 of them are further manually confirmed as *correct* patches by following the protocol presented by Liu *et al.* [34]. We also note that, for 29 other bugs, Avatar generates partially-fixing patches. Eight among these partially-fixing patches are manually found to be correct. Specifically, Avatar generates correct patches for 5 bugs (i.e., *Chart-14, Chart-19, Math-4, Math-77, Math-98*) with multiple buggy positions, where an example with 4 buggy positions (*Chart-14*) is shown in Figure 10.

We closely investigate the bugs with multiple positions/lines fixed by Avatar, and observe that these bugs can be summarized into two categories: (1) the integration of several simple bugs that can be triggered by independent test cases, and (2) the bugs with several buggy lines that can be fixed by directly removing those buggy code lines. It indicates that, although Avatar is capable of fixing the semantic bugs with multiple buggy positions/lines, its bug-fixing ability is still limited by the inherent shortage of fix patterns which are inferred from the patches for static analysis violations with simple code changes (e.g., fixing the null pointer exception, removing the useless code, and replacing the wrongly-used identifier). Fixing complicated bugs is still an open research question for the program repair community.

Although Defects4J has been widely used in the community of automated program repair, Avatar is the first APR tool evaluated with the latest version of Defects4J (i.e., Defects4J – version 2.0.0) that contains 835 bugs from 17 open-source Java projects. As presented in Table 3, Avatar can generate patches for 83 bugs from 14 out of 17 projects in the Defects4J benchmark to make the corresponding patched program pass all tests. More specifically, the patches for 51 out of those 83 bugs in 13 out of 14 projects are correct, while the patches for the remaining 32 bugs are plausible but incorrect. Plausbile patches are valid patches that can make the patched programs pass all tests, but do not really fix the bugs as expected by their developers.

The Bugs.jar dataset contains 1,158 bugs, but this dataset has not been maintained anymore since February 2018 and the majority of those bugs cannot be compiled successfully since the related runtime environments are not provided. It leads that Avatar cannot have trials for all of them since Avatar relies on compiled test cases to locate bug positions and validate patches. Consequently, according to the experiments of generating patches for 60 bugs in the Bugs.jar dataset, Avatar can generate patches for 14 bugs that make those 14 patched programs pass all tests, and 11 of them are correct patches.

Table 3. Number of bugs fixed by Avatar and CoCoNut with the perfect fault localization setting.

Project	CoCoNut [39]	Avatar	
		Fully Fixed bugs	**Partially fixed bugs**
Chart (C)	2/4	8/9	1/3
Cli	5/5	2/4	0/0
Closure (Cl)	8/10	11/14	2/5
Codec (Co)	1/1	2/3	0/0
Compress (Cp)	2/2	4/7	0/1
Csv	1/3	4/5	0/0
Gson (G)	1/1	1/1	1/2
JacksonCore (JC)	4/4	1/2	0/0
JacksonDatabind	4/5	0/0	0/0
Jsoup (J)	1/1	1/6	1/1
JxPath (JP)	1/1	0/1	1/1
Lang (L)	7/8	4/10	1/2
Math (M)	10/13	9/16	0/3
Mockito (Mc)	0/0	2/2	0/1
Time (T)	1/2	2/3	0/0
Defects4J (D4J) Subtotal	48/60	51/83	7/19
Bugs.jar (Bj)	-	11/14	0/1
BEARS	-	2/2	1/2
QuixBugs (Quix)	13/20	8/8	0/7
Total	61/80	72/107	8/29

†In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a valid patch is generated by the APR tool ("?" for y values in the CoCoNut column stands for "**unknown**" as it is not reported in the corresponding paper [39]). The same applies to the following similar tables.

For the BEARS dataset, Avatar only correctly fixes 2 out of 251 bugs. After looking at each bug closely, we observe that most bugs in the BEARS dataset involve multiple buggy lines or multiple buggy locations, which require new code fragments for generating patches with more complicated code change actions than the fix patterns in Avatar. Avatar's ability to address bugs with multiple buggy locations is limited to the bugs whose locations are exposed by individual test cases, which is not the case in BEARS. What's more, Avatar is a repairing tool with simple fix patterns for simple code changes, and is not good at generating new code fragments with complicated code changes for difficult bugs, so Avatar is short of the knowledge to deal with these complicated bugs. To sum up, the fix patterns inferred from the patches of static analysis violations present the potential of fixing semantic bugs, but still are not the feasible prescription for the complicated bugs that are the prime challenge faced by the automated program repair community.

Avatar correctly fixes 8 of bugs in the QuixBugs benchmark without generating any incorrect patches. QuixBugs is a dataset of 40 in-the-lab bugs in the Java implementations of 40 classic algorithms (e.g., quick-sort, depth-first-search). All of the 8 bugs are single-line bugs that can be easily matched with the corresponding fix patterns to mutate the related buggy operators or variable references.

Overall, comparing the number of bugs fixed by Avatar and the number of bugs in each project and each benchmark dataset, Avatar presents overfitting to the evaluation dataset as other APR tools invesitigated by Durieux [5]. For example, Avatar correctly fix 8 out of 26 Chart bugs in Defects4J, but it can only fix one out of 93 Jsoup bugs. On the other hand, Avatar can generate correct patches for 51 out of 835 Defects4J bugs, but can

```
--- a/source/org/jfree/chart/plot/CategoryPlot.java
+++ b/source/org/jfree/chart/plot/CategoryPlot.java
@@ -2163,7 +2163,7 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
             markers = (ArrayList) this.backgroundDomainMarkers.get(new Integer(
                     index));
         }
-        boolean removed = markers.remove(marker);
+        if (markers == null) { return false; } boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
         }
@@ -2445,7 +2445,7 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
             markers = (ArrayList) this.backgroundRangeMarkers.get(new Integer(
                     index));
         }
-        boolean removed = markers.remove(marker);
+        if (markers == null) { return false; } boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
         }

--- a/source/org/jfree/chart/plot/XYPlot.java
+++ b/source/org/jfree/chart/plot/XYPlot.java
@@ -2290,7 +2290,7 @@ public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
             markers = (ArrayList) this.backgroundDomainMarkers.get(new Integer(
                     index));
         }
-        boolean removed = markers.remove(marker);
+        if (markers == null) { return false; } boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
         }
@@ -2526,7 +2526,7 @@ public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
             markers = (ArrayList) this.backgroundRangeMarkers.get(new Integer(
                     index));
         }
-        boolean removed = markers.remove(marker);
+        if (markers == null) { return false; } boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
         }
```

Fig. 10. A single patch fixing 4 buggy positions of *Chart-14*.

only fix 11 out of 1,158 bugs in the Bugs.jar benchmark. Such results indicates that AVATAR also overfits to the benchmark dataset.

RQ1►*AVATAR is capable of fixing semantic bugs collected from real-world open-source projects as well as in-the-lab bugs. AVATAR is also able to fix some bugs with multiple buggy positions/lines, but it is limited to simple bugs that are fixed with simple code change actions in a straightforward way. It is a shortage for AVATAR on addressing complicated bugs since the fix patterns in AVATAR are extracted from the patches of fixing relatively simple static bugs. Fixing complicated bugs and overfitting to benchmark dataset are still open for AVATAR as well as the program repair community.*

Applying AVATAR to Statically-Detected Bugs in Defects4J. In this study, we refer to statically-detected bugs as those whose buggy positions overlap with violation positions reported by static analysis tools. We further apply AVATAR to statically-detected bugs in Defects4J to explore the possibility of fixing these bugs with AVATAR. Table 4 provides details of the bugs (in Defects4J) that can be detected by static analysis tools (see Section 5.3) and are successfully repaired by AVATAR in Defects4J.

Table 4. Statically-detected bugs fixed by Avatar.

Bug ID	SpotBugs	Infer	ErrorProne	Avatar's Fixing Pattern
Chart-1	`NP_ALWAYS_NULL`	`NULL_DEREFERENCE`		`NP_ALWAYS_NULL`
Chart-4	`NP_NULL_ON_SOME_PATH`	`NULL_DEREFERENCE`		`NP_NULL_ON_SOME_PATH`
Chart-8			`ChainingConstructor IgnoresParameter`	`DLS_DEAD_LOCAL_STORE`
Chart-24	`DLS_DEAD_LOCAL_STORE`			`DLS_DEAD_LOCAL_STORE`
Csv-15			`OperatorPrecedence`	`UCF_USELESS_CONTROL_FLOW`
Math-50	`FE_FLOATING_POINT_EQUALITY`			`UCF_USELESS_CONTROL_FLOW`
Math-77			`MissingOverride`	`UPM_UNCALLED_PRIVATE_METHOD`
Total	4/15	2/8	3/19	

x/y reads as: x is the number of bugs fixed by Avatar among the y bugs in Defects4J 2.0.0 that can be localized by each static analysis tool. The information of localizable bugs is partially excerpted from Habib and Pradel's study [10] as well as our execution of the three tools.

We claim that out of the 15, 8, and 19 bugs that can be detected respectively by SpotBugs (v4.0.1), Facebook Infer (v0.17.0), and Google ErrorProne (Ant 2.3.1) on Defects4J 2.0.0, Avatar can successfully generate correct patches for 4, 2 and 3 bugs, respectively. Remark that *Math-77* is a bug with two buggy positions, and Avatar cannot completely fix this bug. Avatar successfully fixes the single buggy position revealed by ErrorProne, and thus we classify it as a positive repair here.

Overall, seven distinct localizable bugs have been correctly fixed with patches generated by fix patterns of FindBugs mined from violation-fixing patches in [27]. All seven bugs are related to distinct violation types. Among 15 violations reported by SpotBugs, *Chart-1* (`NP_ALWAYS_NULL`), *Chart-4* (`NP_NULL_ON_SOME_PATH`), and *Chart-24* (`DLS_DEAD_LOCAL_STORE`) are fixed by the corresponding patterns in Avatar, where *Chart-1* and *Chart-4* are also warned by Infer's `NULL_DEREFERENCE` violation rule. In addition, *Chart-8* warned as `ChainingConstructorI gnoresParameter`, *Math-50* warned as `FL_FLOATING_POINT_EQUALITY`, *Csv-15* warned as `OperatorPrecedencce` and *Math-77* warned as `MissingOverride` by ErrorProne, are fixed by Avatar's `DLS_DEAD_LOCAL_STORE`, `UCF_USELESS_CONTROL_FLOW`, and `UPM_UN CALLED_PRIVATE_METHOD` pattern, respectively. The fix patterns used by Avatar and the corresponding static violations are available in Table 4.

Note that Liu *et al.* [27]'s previous work claimed their mined patterns (for fixing violations detected by FindBugs) could be applied to violations reported by other static analysis tools. Our experiment indeed proves that the patterns fixing two bugs reported by SpotBugs (i.e., the successor of FindBugs) also work with the help of reports by Facebook Infer.

RQ1► *According to fixing bugs detected by static analysis tools, Avatar demonstrates the effectiveness of violation-fixing patterns in a scenario of automatically repairing developers' code, by involving violation positions reported by various static analysis tools.*

Our experiment reveals that Avatar's fix patterns of static analysis violations are not effective on other statically-detected bugs. We investigate the cases of such bugs, and conclude as follows:

(1) Since Avatar implements a limited number of fix patterns, several bugs cannot be matched by any fix pattern. Here, we use *Chart-17*, as an example, detected by SpotBugs as a `CN_IDIOM_S UPER_CALL`[19] violation, shown in Figure 11. The patch generation provided by developers needs five different repair actions, which cannot be matched with any fix patterns in Avatar.
(2) There are also cases where the fixes are truly *domain-specific*, so Avatar's patterns are too general to scale out. We refer to domain-specific fixes as those that are beyond the repairing ability of Avatar. For example,

[19]http://findbugs.sourceforge.net/bugDescriptions.html#CN_IDIOM_NO_SUPER_CALL

```
// Defects4J Dissection - Repair Actions:
// Assignment addition,
// Assignment expression modification,
// Method call addition,
// Method call removal,
// Variable type change

// Violation Type (SpotBugs):
// CN_IDIOM_NO_SUPER_CALL
// clone method does not call super.clone().

--- a/source/org/jfree/data/time/TimeSeries.java
+++ b/source/org/jfree/data/time/TimeSeries.java
@@ -854,7 +854,8 @@ public class TimeSeries extends Series implements Cloneable, Serializable {
      *         subclasses may differ.
      */
  public Object clone() throws CloneNotSupportedException {
-   Object clone = createCopy(0, getItemCount() - 1);
+   TimeSeries clone = (TimeSeries) super.clone();
+   clone.data = (List)ObjectUtilities.deepClone(this.data);
    return clone;
  }
```

Fig. 11. The bug (*Chart-17*) detected by SpotBugs is not fixed by Avatar since no fix pattern can be matched.

```
// Violation Type (SpotBugs):
// CO_COMPARETO_INCORRECT_FLOATING
// compareTo()/compare() incorrectly handles float or double value.

--- a/src/java/org/apache/commons/math/fraction/Fraction.java
+++ b/src/java/org/apache/commons/math/fraction/Fraction.java
@@ -256,8 +256,8 @@ public class Fraction extends Number implements Comparable<Fraction> {
      *         than <tt>object</tt>, 0 if they are equal.
      */
     public int compareTo(Fraction object) {
-        double nOd = doubleValue();
-        double dOn = object.doubleValue();
+        long nOd = ((long) numerator) * object.denominator;
+        long dOn = ((long) denominator) * object.numerator;
         return (nOd < dOn) ? -1 : ((nOd > dOn) ? +1 : 0);
     }
```

Fig. 12. The patch for bug *Math-91* with over-written feature that cannot be fixed by Avatar.

some fixes are actually new features or refactoring, which involve adding or moving multiple lines, such as *Math-91* in Figure 12. The real patch over-writes the calculation of "`nOd`" and "`dOn`" with the specific values except updating the data type, where Avatar does not have enough knowledge to fix them.

(3) In other cases, the violation warned by static analysis tools is actually a coincidental false positive claimed in [10], since the reported violation does not really match the semantically faulty code entity to be modified. Figure 13 provides a descriptive example (*Csv-14*) of such false positives. The violation `OperatorPrecedence`[20] should be fixed with grouping parenthesis to disambiguate expressions that contain both "||" and "&&". However, the real fixing patch is to replace the compiled values for the valuable "`c`". After the bug *Csv-14* is fixed, the violation `OperatorPrecedence` still can be detected by Google ErroProne at the same position. There is no relationship between the fixing behavior and the violation type. Therefore, assessing the ability of detecting semantic bugs for static analysis tools, it should consider the detected position as well as the relation between the violation and the behavior of the buggy code.

[20]https://errorprone.info/bugpattern/OperatorPrecedence

```
// Violation Type (ErrorProne):
// OperatorPrecedence

--- a/src/main/java/org/apache/commons/csv/CSVFormat.java
+++ b/src/main/java/org/apache/commons/csv/CSVFormat.java
@@ -1036,7 +1036,7 @@ public final class CSVFormat implements Serializable {
     char c = value.charAt(pos);

     // RFC4180 (https://tools.ietf.org/html/rfc4180) TEXTDATA =  %x20-21 / %x23-2B / %x2D-7E
-    if (newRecord && (c < '0' || c > '9' && c < 'A' || c > 'Z' && c < 'a' || c > 'z')) {
+    if (newRecord && (c < 0x20 || c > 0x21 && c < 0x23 || c > 0x2B && c < 0x2D || c > 0x7E)) {
         quote = true;
     } else if (c <= COMMENT) {
     // Some other chars at the start of a value caused the parser to fail, so for now
```

Fig. 13. The bug *Csv-14* is false-positively identified as a statically-detected bug.

6.2 Dissecting the Fix Ingredients

Bugs in the benchmark Defects4J are collected from the real-world projects, that failed to pass the concrete functional test cases, while fix patterns implemented in AVATAR are extracted from common patches for fixing static analysis violations. We thus investigate how fix patterns in AVATAR can be leveraged to address semantic bugs from the benchmarks listed in Table 2. To that end, we resort to dissect the ingredients that were successfully leveraged in the generated correct patches, and analyze the similarity between the code change actions of fixing these bugs by developers and the change actions of fix patterns in AVATAR. Table 5 provides the summary of this dissection.

First, we note that all correctly fixed bugs were addressed with patches generated from patterns mined in the study of Liu *et al.* [27] (i.e., based on FindBugs violations). Fix patterns from the study by Rolim *et al.* [54] (which are based on PMD violations) are indeed associated with exceedingly simple violations, which are unlikely to be revealed as semantic bugs (i.e., detected via developer test cases). An example of such simple pattern is their EP7 fix pattern: "replace `List<String> a = new ArrayList()` with `List<String> a = new ArrayList<>()`". In any case, 6 among the 9 fix patterns released by Rolim *et al.* are related to performance, code practice or code style. Our manual investigation of Defects4J bugs reveals that none of the bugs are associated with these types of issues.

Second, we note that the fix patterns of only eight (out of 18) violation types[21] have been successfully used to generate correct patches for Defects4J bugs (c.f. Table 5). Among the 80 (72 fully and 8 partially) correctly fixed bugs, 75 (93.75%) are fixed by the patterns of 4 violation types: `NP_NULL_ON_SOME_PATH`, `DLS_DEAD_LOCAL_STORE`, `UC_USELESS_CONDITION`, and `U CF_USELESS_CONTROL_FLOW`. With the description of violations provided in FindBugs and the repair actions of their fix patterns, we then explain why the fix patterns of static analysis violations can address semantic bugs.

The violation `NP_NULL_ON_SOME_PATH` is explained as dereferencing a null value when the code is executed. In AVATAR, its fix pattern is to add a null check at the dereferencing location straightforwardly, making the program exit the context method or return a default value of the method declaration type when the value of the suspicious variable is `null`. A real example is *Cli-5*: the buggy method tries to strip the leading hyphens from the input `String` parameter "`str`" by calling *String.startsWith()*, without considering the input to be a null string reference that will lead to a *NullPointerException*.

Such an issue can be resolved by adding a null check for the parameter "`str`" before asserting the leading hyphens. Indeed, AVATAR's patch adds a null check for the input parameter and fixes this bug. Figure 14 shows

[21]Note that comparing with the previously published version of AVATAR [30], the newly added fix pattern of only one out of eight violation types has been successfully used to fix one more bug with the correct patch, and other bugs are newly fixed because of the multi-position setting of AVATAR and the new version of Defects4J that contains more bugs than the previously published version.

Table 5. Fix ingredients in the static analysis violation fix patterns used by AVATAR to correctly fix bugs.

Violation Types associated with the Fix Patterns	Fix Ingredients from the Violation Fix Patterns	Fixed Bug IDs
NP_NULL_ON_SOME_PATH	1. Wrap buggy code with if-non-null-check block: "`if (var != null) {buggy code}`"; 2. Insert if-null-check block before buggy code: "`if (var == null) {return false;} buggy code;`" or "`if (var == null) {return null;} buggy code;`" or "`if (var == null) {throw IllegalArgumentException.} buggy code;`".	C-4,14,19,26, Cli-5, Cl-2, Co-17, Csv-4,5,11, G-6, M-4, Mc-29,38, Quix-5, BEARS-56.
DLS_DEAD_LOCAL_STORE	Replace a variable with other one: e.g., "var1 = ~~var2~~var3;".	C-8,11,24, Cp-14, L-6,57,59, M-33,59,98, T-7, Bj-L-16,Bj-O-194, Bj-M-45,63,79,114, Quix-8,11,31.
UC_USELESS_CONDITION	1. Mutate the operator of an expression in an if statement: e.g., "if (expA ~~>~~>= expB) {...}"; 2. Remove a sub-predicate expression in an if statement: "if (~~expA ||~~ expB) {...}" or "if (expA ~~|| expB~~) {...}"; 3. Remove the conditional expression: "~~expA ? expB :~~ expC" or "~~expA ?~~ expB ~~: expC~~".	Cl-18,31,38, Cl-62,73,168, Cp-19, JC-25, L-15, M-82,85, T-19, BEARS-133, 135 Quix-6,14,24,28, Bj-A-21, Bj-O-183, Bj-M-38,54.
UCF_USELESS_CONTROL_FLOW	1. Remove an if statement but keep the code inside its block: "~~if (exp) {~~ code ~~}~~"; 2. Remove an if statement with its block code: "~~if (exp) { code }~~".	C-18, Cli-32, Cl-11,106,115,126,138, Co-8, Cp-27,31, Csv-15, G-4, J-63,68, JP-17, L-10, M-50.
ICAST_IDIV_CAST_TO_DOUBLE	Cast the operands of division from integer to double: "FastMath.pow(2 * FastMath.PI, ~~-dim~~ ((double) -dim) / ~~2~~ 2d);".	M-11, Bj-M-137.
NP_ALWAYS_NULL	Mutate the operator of null-check expression: "var ~~!=~~ == null", or "var ~~==~~ != null".	C-1.
UPM_UNCALLED_PRIVATE_METHOD	Remove a method declaration: "~~Modifier ReutrnType methodName(Parameters) { code }~~".	Cl-46.
BC_UNCONFIRMED_CAST	Wrap buggy code with if-instanceof-check block: "`if (var instanceof Type) {buggy code}` `else {throw IllegalArgumentException;}`".	M-89.

† Only correctly fixed (including partially correctly-fixed bugs highlighted with *italic*) bugs are listed in this table.
∗ "Bj-A", "Bj-L", "Bj-M" and "Bj-O" represent "Accumulo", "Log4J2", "Commons-Math" and "OAK" programs in the Bugs.jar dataset, respectively.

the detail of the patch in GNU Diff format. The bugs *Chart-14, 19*, *Codec-17*, *Csv-4*, *Csv-11*, *Math-4*, and *Mockito-38* have the same problem and are addressed in the same way by AVATAR. The bugs *Chart-4, 26*, *Closure-2*, *Csv-5*, *Gson-6*, and *Mockito-29* also face the same problem and are fixed by AVATAR, warping the impacted code fragments with the conditional `non-null` check block. The aforementioned 16 bugs are caused by ignoring the `null` cases of the referenced variables, which are just right for the fix pattern `NP_NULL_ON_SOME_PATH` in AVATAR.

FindBugs describes the violation `DLS_DEAD_LOCAL_STORE` as assigning a value to a local variable whose value is not read or used in any subsequent instruction. It often indicates an error because the value computed is never used. The repair action of its fix pattern is to substitute a variable at the buggy location to another variable, which addresses the same code change behavior of replacing the buggy referenced variable with the correct one. Take *Lang-6* as an example. At the buggy location, a for-loop is used to accumulate the number of characters, but the local variable "`pt`" defined in the for-loop is never used, which causes the bug.

AVATAR fixes this bug by replacing the variable "`pos`" with the local variable "`pt`". Figure 15 shows the detail of the patch. By replacing the buggy referenced variable with the adequate variable, AVATAR correctly fixes other 19 bugs (i.e., *Chart-8, 11, 24, Compress-14, Lang-57, 59, Math-33, 59, 98, Time-7, Bugs.jar-Log4J2-16, Bugs.jar-OAK-194, Bugs.jar-Math-45, 63, 79, 114,* and *QuixBugs-8, 11, 31*) as well. Although the initial object of the fix pattern

```
// Violation Type: NP_NULL_ON_SOME_PATH

// Defects4J Dissection:
// Bug Pattern: Missing null check addition.

// Repair Action:
// Adding a conditional null check branch.

--- a/src/java/org/apache/commons/cli/Util.java
+++ b/src/java/org/apache/commons/cli/Util.java
@@ -33,5 +33,5 @@ class Util {
     */
     static String stripLeadingHyphens(String str) {
-        if (str.startsWith("--")) {
+        if (str == null) { return null; } if (str.startsWith("--")) {
            return str.substring(2, str.length());
        }
```

Fig. 14. *Cli-5* in Defects4J fixed by AVATAR with the fix pattern for the NP_NULL_ON_SOME_PATH violation.

```
// Violation Type: NP_NULL_ON_SOME_PATH

// Defects4J Dissection:
// Bug Pattern: Wrong variable reference.

// Repair Action:
// Replacing the buggy variable reference with another one.

--- a/src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java
+++ b/src/main/java/org/apache/commons/lang3/text/translate/CharSequenceTranslator.java
@@ -92,5 +92,5 @@ 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. 15. *Lang-6* in Defects4J fixed by AVATAR with the fix pattern for the DLS_DEAD_LOCAL_STORE violation.

DLS_DEAD_LOCAL_STORE is to address the uncalled variable, its repair action can be transplanted to fixing bugs with the wrong-referenced variable. Additionally, the patches generated by AVATAR for bugs *Compress-14* and *Lang-57* are different from the ground-truth patches provided by developers, but they are semantically similar to each other since they reach the same target in different ways. It indicates that AVATAR can fix bugs in a way that is different from the developers' bug-fixing results.

For the violation UC_USELESS_CONDITION, FindBugs explains it as a case that a condition always produces the same result as the value of the involved variable was narrowed before, and will not affect the related code. Its repair action is either removing the condition or trying to modify the condition (e.g. by flipping the operator). One example is the patch for bug *Closure-31* shown in Figure 3, where AVATAR's patch removes one of the three conditions in the buggy if-statement and fixes this bug. Figure 16 shows another example, the patch for the bug *Time-19*, that is generated AVATAR by changing the operator in the buggy conditional infix-expression. AVATAR also generates correct patches for the other 20 bugs (i.e., *Closure-18, 38, 62, 73, 168, Compress-19, JacksonCore-25, Lang-15, Math-82, 85, BEARS-133, 135, QuixBugs-6, 14, 24, 28, Bugs.jar-Accumulo-21, Bugs.jar-OAK-138*, and Bugs.jar-Math-38, 54). We observe that these bugs are caused by the redundant conditional expressions or the incorrect operators in the conditional infix-expressions, which can be fixed by removing the unneeded conditional

```
// Violation Type: UC_USELESS_CONDITION

// Defects4J Dissection:
// Bug Pattern: Logic expression modification.

// Repair Action:
// Flipping the buggy operation in the infixexpression.

--- a/src/main/java/org/joda/time/DateTimeZone.java
+++ b/src/main/java/org/joda/time/DateTimeZone.java
@@ -897,7 +897,7 @@ public abstract class DateTimeZone implements Serializable {
                    return offsetLocal;
                }
            }
-       } else if (offsetLocal > 0) {
+       } else if (offsetLocal >= 0) {
            long prev = previousTransition(instantAdjusted);
            if (prev < instantAdjusted) {
                int offsetPrev = getOffset(prev);
```

Fig. 16. *Time-19* in Defects4J fixed by Avatar with the fix pattern for the UC_USELESS_CONDITION violation.

expressions or flipping the operators in buggy infix-expressions. The UC_USELESS_CONDITION violation describes different issues from those bugs, but their fixing behavior reaches the same destination.

Being different from the UC_USELESS_CONDITION, the violation UCF_USELESS_CONTROL _FLOW means that the control flow of a statement continues onto the same place regardless of whether the branch is taken or not. Thus, its fix pattern is to remove the control flow but keep the code in the block, or remove all code of the entire control flow, as presented in Table 5. The bug *Lang-10* shown in Figure 17 is caused by the "IfStatement" which will change the value of the variable "regex" and break its parent loop "ForStatement", thus, it is fixed by removing the entire control flow of the "IfStatement" with Avatar. Avatar also generates correct patches for other 12 bugs (i.e., *Cli-32, Closure-11, 115, 126, Codec-18, Compress-27, 31, Csv-15, Gson-4, Jsoup-63, 68* and *Math-50*) that have the same problem with the same repair action. In addition, Avatar correctly fixes *Chart-18, Closure-106, 138*, and *JxPath-17* by removing the control flow statement but keeping the code in the block. The fix pattern for the UC_USELESS_CONDITION violation is to remove the useless control flow in code, which is actionable to remove the buggy control flow in programs to fix the related bug.

The other 5 bugs are fixed with the remaining four fix patterns. ICAST_IDIV_CAST_TO_DOUBLE warns about the division result of implicitly casting integers to float-point numbers may lose precision that may deviate from the developer's expected behavior for programs. Its fix pattern in Avatar will cast both of the integer operands to float-point numbers in the dividing expression before performing the division, which can resolve the losing-precision problem. It indeed fixes the related bugs *Math-11* and *Bugs.jar-Math-137*. The violation NP_ALWAYS_NULL describes a *NullPointerException* will be thrown because of a null pointer dereference. Avatar addresses this issue by flipping the operator in the conditional null-checking expression to avoid dereferencing, which correctly fixes the bug *Chart-1*. UPM_UNCALLED_PRIVATE_METHOD is used to present that a private method is never called in the program, which is fixed by removing the entire uncalled method directly. Our experimental results show that its fix pattern can be used to fix the bug caused by the inadequately override method (i.e., *Closure-46*). BC_UNCONFIRMED_CAST represents the unchecked/unconfirmed cast in code that could cause the faulty execution with ClassCastException. Thus, its fix pattern is to add the instanceof checking before the cast, which indeed fixes the semantic bug caused by the unchecked cast (i.e., *Math-89*). Note that, the bug fixing performance of these four fix patterns is not associated with the static analysis tools and their granularity, which could be related to the two-aspect reasons discussed below.

The experimental results suggest that the fix patterns for static analysis violations can be used to address semantic bugs that make the program fail on functional tests. However, some of the fix patterns mined from the

```
// Violation Type: UCF_USELESS_CONTROL_FLOW

// Defects4J Dissection:
// Bug Pattern: Conditional block removal.

// Repair Action:
// Removing the entire buggy if statement.

--- a/src/main/java/org/apache/commons/lang3/time/FastDateParser.java
+++ b/src/main/java/org/apache/commons/lang3/time/FastDateParser.java
@@ -304,13 +304,13 @@ public class FastDateParser implements DateParser, Serializable {
         boolean wasWhite= false;
         for(int i= 0; i<value.length(); ++i) {
             char c= value.charAt(i);
-            if(Character.isWhitespace(c)) {
-                if(!wasWhite) {
-                    wasWhite= true;
-                    regex.append("\\s*+");
-                }
-                continue;
-            }
+
+
+
+
+
+
+
             wasWhite= false;
             switch(c) {
             case '\'':
```

Fig. 17. *Lang-10* in Defects4J fixed by Avatar with the fix pattern for the violation UC_USELESS_CONTROL_FLOW.

available data were not successfully applied to fix semantic bugs in the considered benchmarks, or were useful only on a small number of bugs. Nevertheless, this experimental fact could be biased by the available artifact, and we could not conclude that such fix patterns are definitely not effective. Indeed, we can identify two reasons why these patterns were not effective in our experiments: (1) the low diversity of bugs in the benchmark dataset which maintained manually, and (2) the representativity of the mined fix patterns may be limited: they are mined from publicly-recorded patches; other patches made during development that are associated to functional test cases may be missing in our inference patch dataset. We further note that, static analysis violations may be entirely different from the functions bugs that make programs fail to pass test cases. Exploring such cases is however out of the scope of our study. Instead, our work attempts to leverage the fix patterns that can be relevant for addressing semantic bugs.

RQ2►*Avatar exploits a variety of fix ingredients from the fix patterns of static violations to address a diverse set of bugs in both the real-world and in-the-lab benchmark datasets. Semantic bugs could be caused by problems that are different from the static violations, but they still can be solved by the fix patterns for static violations.*

6.3 Comparing against the State-of-the-Art

To reliably compare against the state-of-the-art in Automated Program Repair (APR), we must ensure that the Fault Localization (FL) step is properly tuned, as FL could bias the bug fixing performance of APR tools [29, 33, 34]. We identify three major configurations in the literature:

(1) ***Restricted_FL*-based APR**: in this case, APR systems make the assumption that some information of the code location is available. For example, in HDRepair [24], fault localization is restricted to the faulty methods,

which are assumed to be known. Such a restriction substantially increases the accuracy of the target list of fault locations for which a patch must be generated [29]. In Section 6.1, we have made a similar strong assumption that fault locations are known, as our objective was to assess the patch generation performance of AVATAR.

(2) ***Normal_FL*-based APR**: in this case, APR systems directly use off-the-shelf fault localization techniques (e.g., GZoltar [3]) to localize bug positions at the line level. Specifically, 13 state-of-the-art open-source APR tools are re-executed with the same fault localization setting (i.e., GZoltar-v1.7.2 + Ochiai ranking metric) as AVATAR.

(3) **APR tools with *Unspecified/Unconfirmed* FL configuration**: in this case, APR tools only claim that the fault localization is proceeded with spectrum-based fault localization without clearly clarifying the exact FL technique. And we failed to re-execute the related APR tools for different reasons (e.g., the problems from availability or executability).

We thus compare the bug fixing performance of AVATAR with the state-of-the-art APR tools after classifying them into these three groups. Note that, in this section, the experiments for AVATAR are proceeded with the Defects4J dataset since the state-of-the-art APR tools are mainly evaluated on this benchmark.

6.3.1 Comparison against Restricted_FL-based APR Tools. We first compare AVATAR against the state-of-the-art APR systems, which implement a *restricted* fault localization configuration, including HDRepair [24], JAID [4], and SketchFix [14]. For AVATAR, we select faulty locations using the same assumption as these three tools, i.e., focusing on attempting to repair suspicious code statements that are reported by the fault localization tool (i.e., the GZoltar-v0.1.1 with the Ochiai metric [52]) but only considering the suspicious statements within the known faulty methods. This assumption leaves out many noisy statements, reducing the probability of generating plausible patches for bugs and further increasing the chance to generate a correct patch before a plausible one or execution timeout.

In the community, APR tools have been proposed to be evaluated with the assumption of perfect fault localization. Such APR tool (CoCoNut [39]) is included in this category as well. The comparison results between AVATAR and CoCoNut are presented in Table 3. We observe that, AVATAR can correctly fix more bugs in Defects4J than CoCoNut[22], but CoCoNut can correctly fix more bugs in QuixBugs than AVATAR. AVATAR achieves higher correct ratios (**CR**) of generating correct patches for fixed bugs in QuixBugs dataset with 100% precision, but achieves lower CR on Defects4J dataset with 61.4% precision than CoCoNut (i.e., 65% and 80%, respectively). Figure 18 further illustrates the differences between the bugs correctly fixed by AVATAR and CoCoNut. CoConut can correctly fix more bugs in QuixBugs that cannot be correctly fixed by AVATAR, but AVATAR can correctly fix more Defects4J bugs that cannot be correctly fixed by CoCoNut. CoCoNut cannot generate correct patches for 39 (=36 + 3) bugs in Defects4J and QuixBugs that can be correctly fixed by AVATAR. It indicates that AVATAR can be complementary to the state-of-the-art CoCoNut APR tool.

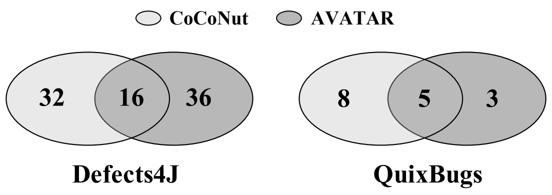


Fig. 18. Relationship of correctly fixed bugs by CoCoNut and AVATAR.

[22]We re-executed CoCoNut with the Defects4J and QuixBugs, but failed to re-execute it with Bugs.jar and BEARS.

Table 6 presents the comparing results against the APR tools under the assumption of known faulty methods. Note that, HDRepair, JAID, and SketchFix are evaluated with the early version of Defects4J, thus their fixing performance on the newly-added bugs in Defects4J-v2.0.0 is not available, which is presented with the dash symbols in Table 6. In addition, 4 bugs (*Closure-63*, *Closure-93*, *Lang-2*, *Time-21*) are deprecated in Defects4J 2.0.0, so the related results in each work are filtered out, respectively.

Table 6. Comparison of Avatar with HDRepair [24], JAID [4] and SketchFix [14].

Project	**HDRepair**	**JAID**	**SketchFix**	**Avatar**	
				Fully fixed	Partially fixed
Chart	0/2	2/4	6/8	**7**/**10**	0/2
Closure	0/7	**5**/11	3/5	**5**/**19**	0/1
Lang	2/6	1/**8**	**3**/4	2/6	0/1
Math	4/7	1/8	**7**/8	**7**/**18**	1/3
Mockito	0/0	0/0	0/0	**2**/**2**	0/1
Time	0/1	0/0	0/1	**2**/**2**	0/0
Others*	-	-	-	8/22	1/8
Total	6/23	9/31	19/26	**25**/**57** (**33**/**79**)*	1/8 (2/16)*
CR (%)	26.1	29.0	**73.1**	43.9 (41.8)	12.5

The results for HDRepair, JAID, and SketchFix are provided by their authors.
***Others** refer to the bugs newly added in Defects4J-v2.0.0 that are not evaluated by APR systems except Avatar, and the numbers presented in the parentheses include the results for those newly-added bugs fixed by Avatar. **CR** represents the correctness ratio of valid patches generated by APR tools that are correct [34], the same for Table 7.
*The number of bugs fixed by Avatar shown in this table is a little different from the data in Table 3. For fixing each bug, the input of Avatar is a ranked list of suspicious statements in the faulty methods, which is different from the input of Avatar in the experiments of Section 6.1 and Section 6.3.2.

Compared with three state-of-the-art tools under the intersectional bug collections of six projects (*Chart*, *Closure*, *Lang*, *Math*, *Mockito*, *Time*), Avatar correctly fixes more bugs than all three tools, and yields a higher probability to generate correct patches among all plausible patches than HDRepair and JAID (cf. CR(%) in Table 6). All 25 bugs fixed by Avatar are not addressed by HDRepair, and Avatar also correctly fixes 6 bugs (i.e., *Chart-1*, *Closure-62*, *Closure-73*, *Lang-57*, *Lang-59*, *Time-19*) that are only plausibly (but incorrectly) fixed by HDRepair. 21 bugs correctly fixed by Avatar are not addressed by JAID, and 17 bugs correctly fixed by Avatar are not addressed by SketchFix.

Overall, as illustrated in Figure 19, 18 bugs correctly fixed by JAID, HDRepair or SketchFix are not correctly resolved by Avatar. On the opposite, JAID, HDRepair, or SketchFix cannot generate correct patches for 14 bugs that are correctly fixed by Avatar. ***Avatar could be complementary to the state-of-the-art APR tools configured with the restricted fault localization.***

In addition, Avatar can generate valid patches for 22 bugs newly added in Defects4J, and 8 of them are fixed with correct patches. Finally, Avatar partially fixes 16 bugs that have multiple faulty code fragments, and 2 of the associated patches are manually checked to be correct. Note that, considering the bugs with multiple buggy positions, due to the limitation of bug-fixing mechanism of Avatar, Avatar cannot fix the the bugs with multiple bug positions of which bug-triggering test cases are overlapped, but can only fix the bugs with multiple bug positions that are triggered by independent test cases.

Compared with the results summarized in Table 3 in Section 6.1 where the perfect fault localization is configured, Avatar's performance falls quickly under the guide of method-level fault localization results. We analyze the reasons as follows:

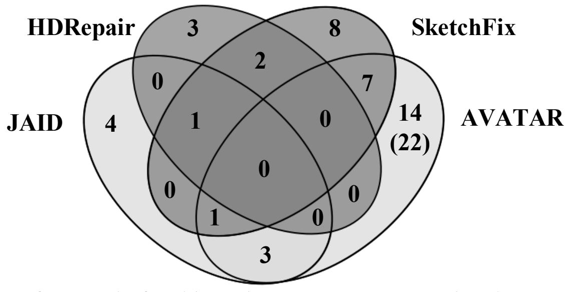


Fig. 19. Relationship of correctly fixed bugs by HDRepair, JAID, SketchFix, and Avatar, respectively.

(1) In the experiment setting of Section 6.1, we allow Avatar to generate all plausible patches for a single bug. We aim to fully evaluate the fixing ability of Avatar. However, we focus on a real-world circumstance here, and Avatar is only allowed to generate one plausible patch for one bug, where a plausible patch may prevent generating a correct patch.
(2) The inaccuracy of fault localization results will mislead Avatar on the direction of fixing, which will bias the results. Our experiment shows that Avatar sometimes repairs at locations without any fault. So the generated plausible patches are inevitably false positive ones.

RQ3►*Avatar presents the higher efficacy of generating meaningful patches with a set of pre-defined patterns than the state-of-the-art approaches on the Defects4J benchmark, under the assumption of perfect fault localization. We also show that the inaccuracy of bug localization in a known method will bias the bug-fixing performance.*

6.3.2 Comparison against Normal_FL-based APR Tools. We compare the bug fixing performance of Avatar with the *Normal_FL*-based state-of-the-art APR tools that are evaluated on the Defects4J benchmark. These APR tools take as input a ranked list of suspicious statements that are reported by an off-the-shelf fault localization technique. In this experiment, we consider a group of APR systems, namely jGenProg [41], jKali [41], jMutRepair [42], DynaMoth [6], Nopol [70], Cardumen [43], kPAR [29], FixMiner [21], TBar [31], ARJA [76], RSRepair-A [76], ACS [68] and SimFix [16] which are re-executed with the same fault localization configuration (i.e., GZoltar-v1.7.2 and the Ochiai ranking metric) by Liu et al. [34] as Avatar. Thus, we directly excerpt the related results in their experiments[34], to avoid the potential bias from the different fault localization settings [29]. Table 7 reports the comparison results in terms of the number of *plausibly-fixed* bugs and the number of *correctly-fixed* bugs. Since other tools are evaluated on the earlier version of Defects4J, the newly-added bugs in Defects4J-2.0.0 that are fixed by Avatar are summarized into the **Others** column. For Avatar, in the last two columns of Table 7, the numeric values presented in parentheses include the newly-added and fixed Defects4J bugs.

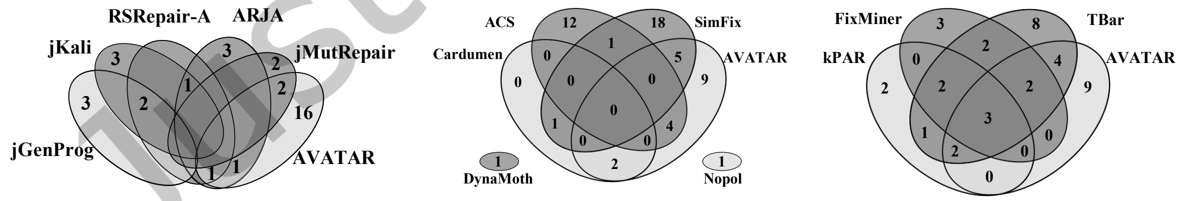
We group the existing program repair tools into three categories by their core approaches: random-based approaches (i.e., jGenProg, jKali, jMutRepair, ARJA, and RSRepair-A), sophisticated heuristic approaches (i.e., DynaMoth, Nopol, Cardumen, ACS, and SimFix), and template-based approaches (i.e., kPAR, FixMiner, andTBar), and illustrate the differences between they and Avatar on the fixed bugs in terms of venn graphs presented in Figure 20. Compared against random-based, sophisticated heuristic, and template-based program repair approaches, Avatar can correctly fix 16, 9, and 9 new bugs that were not fixed by them before, respectively. Such results indicate that Avatar could be complementary to the state-of-the-art program repair tools when they are set with the same fault localization configuration.

We note that Avatar outperforms all of the *Normal_FL*-based APR systems in terms of both the number of plausibly fixed bugs and the number of correctly fixed bugs, except T-Bar, ACS, and SimFix. Here we claim that TBar considers various fix patterns released by the program repair community in recent years. Avatar also yields

Table 7. Number of bugs reported as having been fixed by different APR tools.

Fault Localization	**APR Tool**	**Chart**	**Closure**	**Lang**	**Math**	**Mockito**	**Time**	**Others**[*]	**Total**	**CR**[*](%)
GZoltar-0.1.1	**Avatar**	6/11	4/16	2/6	7/24	2/2	0/0	6/24	21/59 (27/83)[*]	35.6 (32.5)
GZoltar-1.7.2		6/11	6/15	1/6	5/18	0/0	0/0	2/17	18/50 (20/67)[*]	36 (29.9)
Normal_FL-based APR Tools	**jGenProg [42]**	0/5	2/2	0/2	3/11	0/0	0/0	-	5/20	25.0
	jKali [42]	0/4	3/8	2/4	1/9	0/0	0/0	-	6/25	24
	jMutRepair [42]	1/4	2/5	0/2	2/11	0/0	0/0	-	5/22	22.7
	DynaMoth [6]	0/6	-	0/2	1/13	0/0	0/1	-	1/22	4.5
	Nopol [70]	0/6	-	1/6	0/18	0/0	0/1	-	1/31	3.2
	Cardumen [43]	2/4	0/2	0/0	1/6	0/0	0/0	-	3/12	25.0
	kPAR [29]	3/13	2/10	1/18	4/22	0/0	0/1	-	10/63	15.9
	FixMiner [21]	5/14	0/2	0/2	7/15	0/0	0/0	-	12/33	36.4
	TBar [31]	7/16	3/12	6/21	8/23	0/0	0/0	-	24/72	33.3
	RSRepair-A [76]	0/4	4/22	0/3	0/12	0/0	0/0	-	4/41	9.8
	ARJA [76]	1/10	2/29	0/3	3/15	0/1	0/0	-	6/58	10.3
	ACS [68]	2/2	0/0	3/3	11/16	0/0	1/1	-	17/22	77.3
	SimFix [16]	3/8	7/19	5/16	10/25	0/0	0/0	-	25/68	36.8
APR tools with *Unspecified/ Unconfirmed FL*	**ssFix [66]**	3/7	2/11	5/12	10/26	0/0	0/4	-	20/60	33.3
	CapGen [63]	4/4	-	5/5	12/16	-	0/0	-	21/25	84.0
	LSRepair [32]	3/8	0/0	8/14	7/14	1/1	0/0	-	19/37	51.4
	ELIXIR [56]	4/7	0/0	8/12	12/19	0/0	2/3	-	26/41	63.4
	Hercules [57]	6/9	6/8	10/13	21/28	-	3/5	-	46/63	73.0
	PraPR [9][*]	4/14	12/62	3/19	6/40	0/5	1/8	-	26/148	17.6
	DeepRepair [64]	0/4	-	4/5	1/5	-	-	-	5/14	35.7
	VFix [69]	6/6	-	5/5	1/2	-	0/0	-	12/13	92.3
	ConFix [19]	4/13	6/21	5/15	6/37	-	1/6	-	22/92	23.9
	GenPat [15]	3/8	5/7	4/11	3/13	1/2	0/1	-	16/42	38.1
	DLFix [26]	5/12	6/10	5/12	12/28	1/1	1/2	-	30/65	46.2

[*]**PraPR** generates plausible patches for 148 Defects4J bugs and 43 of them can be fixed with correct patches, but 23 of them are fixed with correct patches that are ranked as the first plausible patch.



(a) Comparing with random-based approaches. (b) Comparing with sophisticated heurisctical approaches. (c) Comparing with random-based approaches.

Fig. 20. Comparing Avatar with the three kinds of state-of-the-art program repair approaches.

a higher probability to generate correct patches among its plausible patches than those tools, except FixMiner, ACS, and SimFix. In all, we claim that Avatar achieves good performance for fixing the Defects4J bugs both in quantity and quality.

Compared with the results in Section 6.3.1, although both experiments use the same fault localization metric, some of the bugs are not correctly fixed in the *Normal_FL*-based configuration. We analyze the reasons as follows:

(1) As we have mentioned previously, the experiment in Section 6.3.1 uses only an effective subset of a suspiciousness ranking list to meet the assumption that method-level fault locations are known. However, here we use the whole ranked suspicious list, which could bias AVATAR from fixing the real buggy lines.
(2) Since AVATAR generates patches based on the whole suspiciousness list, the quantity of generated patches may be inevitably huge, especially for bugs with thousands of suspicious locations. We believe generating patches as many as possible is a waste of resources. So, we allow AVATAR to generate up to 5,000 patches at most. As a result, some of the correct patches ranked after this threshold because of low suspiciousness in code are ignored by AVATAR.
(3) Due to the limitation of fault localization techniques, some bugs fail to be localized by fault localization tools, and consequently do not exist in the suspiciousness list. It is impossible for AVATAR to correctly fix such bugs.

RQ3►*In terms of quantity and quality of generated plausible patches, AVATAR addresses more bugs than its immediate competitors. Nevertheless, we note that AVATAR is complementary to the other state-of-the-art APR systems, fixing bugs that others do not fix.*

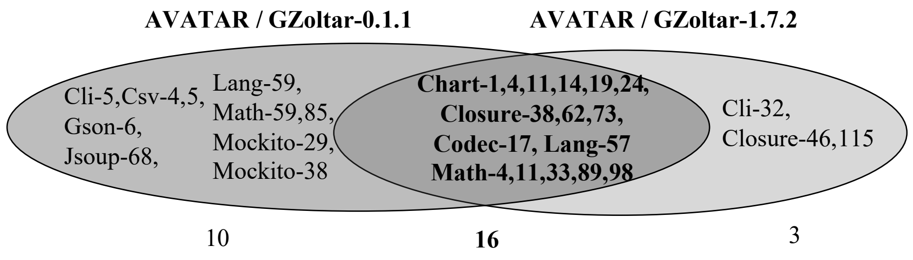
6.3.3 Comparison against with Unspecified/Unconfirmed FL-configured APR tools. We also compare AVATAR against APR systems whose authors do not explicitly describe the actual fault localization configuration, but still manage to fix bugs that we could not localize with GZoltar-1.7.2/Ochiai. For example, ELIXIR [56], PraPR [9] and GenPat [15] rely on the Ochiai technique to identify potential buggy statements, but more details about off-the-shelf fault localization techniques are not provided. Hercules stated that it uses a spectrum based fault localization technique to detect potential repair locations. CapGen [63] applies GZoltar and Ochiai to detect bug positions, but the exact version of GZoltar is not clarified. In a recent study, Liu et al. [29] reported that the different versions of GZoltar present different performance on locating Defects4J bugs. We include in this category other APR tools that we failed to re-execute them because of the problems from availability (i.e., APR tool is not publicly available) or executability (i.e., an APR tool is publicly available, but cannot be executed as-is for diverse issues, such as the specific configuration for LSRepair [32] and the online connection for ssFix [66]). Thus, the bug-fixing results for these APR tools are directly excerpted from their papers.

Overall, AVATAR shows a competitive repair performance, comparing with APR tools with Unspecified/Unconfirmed FL techniques. The compared performance results are illustrated in Table 7. Although our tool does not completely outperforms other tools in this category, it produces comparable or better results than 5 out of 11 APR tools. Due to the differences in fault localization result, AVATAR underperforms other 6 APR tools such as Hercules, ELIXIR, and DLFIX. This might be mitigated by applying the same FL setting to our tool. AVATAR also shows the best performance on fixing bugs for *Chart* and *Mockito* subjects comparing with other tools in this category.

RQ3►*AVATAR underperforms some of APR systems. Nevertheless, AVATAR is still complementary to them as it is capable of addressing some Defects4J bugs that state-of-the-art cannot fix.*

6.4 Bias from Fault Localization Settings

In APR community, there is a primary challenge faced by practitioners: the plausible patches generated by APR tools. Qi et al. [53] and Smith et al. [58] reported that the plausible patches generated by APR tools can make the patched buggy programs pass all test suites, but these patches do not correctly fix the related bugs



AVATAR / GZoltar-0.1.1
AVATAR / GZoltar-1.7.2
Cli-5,Csv-4,5, Gson-6, Jsoup-68,
Lang-59, Math-59,85, Mockito-29, Mockito-38
Chart-1,4,11,14,19,24, Closure-38,62,73, Codec-17, Lang-57 Math-4,11,33,89,98
Cli-32, Closure-46,115
10
16
3

Fig. 21. Bugs correctly fixed by Avatar under GZoltar 0.1.1 and 1.7.2, respectively.

as expected by developers, or even make the patched programs worse than the buggy ones. As analyzed by Liu et al. with the empirical study [29], such plausible patches could be generated by modifying the non-buggy statements that are incorrectly identified as the suspicious statements by the corresponding fault localization technique. And the same APR tools could present different bug-fixing performance when they are configured with different fault localization settings [34]. Therefore, this section aims at investigating to what extent Avatar can be affected on bug-fixing performance when it is configured with different fault localization techniques. As presented in Table 7, when the patch generation process of Avatar is fed with suspicious bug positions uncovered by different fault localization techniques (i.e., GZoltar-0.1.1 and GZoltar-1.7.2 in this study), Avatar presents different bug-fixing performances (cf. Table 7) in terms of the plausibly-fixed and correctly-fixed bugs as well as the related correctness ratio for generated patches. It further confirms that the APR tool could be biased by the fault localization techniques from the aspect of normal program repair pipeline, which is not investigated in the literature [29, 34].

GZoltar-0.1.1 is the early version of GZoltar, and GZoltar-1.7.2 is the advanced one. However, the number of bugs plausibly-fixed and correctly-fixed by Avatar with as well as the related correctness ratio are higher than Avatar with GZoltar-1.7.2. As detailed in Figure 21, with GZoltar-0.1.1, Avatar can correctly fix 10 bugs that cannot be correctly addressed with GZoltar-1.7.2, only three bugs belong to the opposite situation. From the aspect of the number of fixed bugs, Avatar can benefit from the early version of GZoltar, although its advanced version can be used to detect more bugs.

We further investigate the efficiency in terms of the number of patch candidates generated by Avatar to fix each bug with two different versions of GZoltar, of which results are illustrated in Figure 22. The efficiency of generating patch candidates has been highlighted in a recent work [34] as one of important aspects to assess APR tools. "*Plausible*" represents the bugs that are fixed by Avatar with GZoltar-0.1.1 and GZoltar-1.7.2, respectively. "*Correct*" denotes the correctly fixed bugs with GZoltar-0.1.1 and GZoltar-1.7.2, respectively While "*Correct'*" considers the 16 bugs that are correctly fixed by Avatar using both the versions of GZoltar.

When focusing on the efficiency of generating patches for all plausibly-fixed bugs, Avatar with GZoltar-0.1.1 presents a higher efficiency than Avatar with GZoltar-1.7.2. Nevertheless, when looking at the correctly fixed bugs and the intersection of correctly fixed bugs, Avatar with GZoltar-0.1.1 is less efficient than Avatar with GZoltar-1.7.2.

To sum up, the advanced fault localization technique, GZoltar, could not assist Avatar to fix more bugs than its early version, but it can make Avatar generate correct patches for fixing bugs in a more efficient way. Therefore, fixing more bugs and fixing bugs in a more efficient way lead to a trade-off on selecting fault localization techniques for Avatar to fix real bugs.

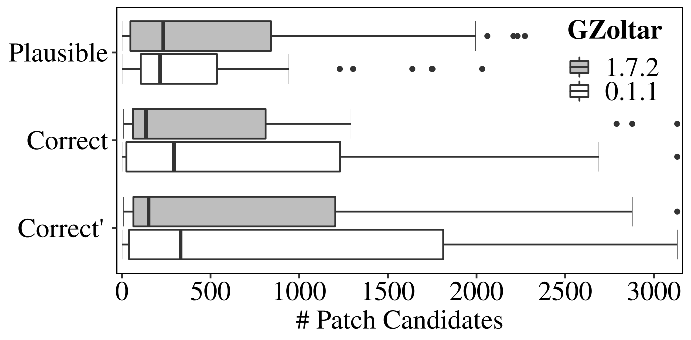


Fig. 22. Efficiency of fixing bugs with AVATAR under GZoltar-0.1.1 and GZoltar-1.7.2.

RQ4► *The bug fixing performance of AVATAR may be biased by different fault localization techniques. The trade-off between the number of fixed bugs and the efficiency of generating patch candidates should be well addressed to select the related fault localization technique when applying AVATAR to practical buggy programs.*

6.5 Improving Bug-Fixing Performance with Stack Trace Information

AVATAR revises suspicious statements in the ranked list exposed in the fault localization process one by one to generate patch candidates until one valid patch is found. It is somehow different from the manually debugging process. 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 23), developers are more prone to address these crashed statements than others. The line-05 in Figure 23 indeed is the buggy statement shown in Figure 24. So, we mimic the manual debugging and apply the stack trace information to the fault localization of AVATAR, to assess whether the bug-fixing performance of AVATAR can be improved.

```
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(CharSequence
  Translator.java:95)
06:   at org.apache.commons.lang3.text.translate.CharSequenceTranslator.translate(CharSequence Translator.
   java:59)
07:   at org.apache.commons.lang3.StringEscapeUtils.escapeCsv(StringEscapeUtils.java:556)
   ......
```

Fig. 23. Excerpted stack trace after executing the test cases of bug *Lang-6* in Defects4J.

After closely looking at the stack trace information, we observe that three kinds of information could be used to refine the fault localization results: ① the class targeted by the failing-executed test case(s) (e.g., the tested class `org.joda.time.format.DateTimeFormatter` for bug Time-7 shown in Figure 25), ② the exact code method tested by the failing-executed test case(s) (e.g., the method `parseInto` in the class `org.joda.time.format.DateTimeFormatter` in Figure 25), and ③ the crashed statements from the source code of the buggy program (e.g., the line-05 highlighted with red in Figure 23). Other information is not related to the bug positions, thus is discarded in this study for refining the fault localization. With the observation, we propose to refine the fault localization as below: ❶ **Prioritize the crashed source code statements (s_1) in the stack trace.** It straightforwardly considers that the crashed source code statements in stack trace have a

```
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. 24. Developer's patch for fixing bug *Lang-6* in Defects4J.

Excerpted stack trace after executing the test cases of bug Time-7:

```
---org.joda.time.format.TestDateTimeFormatter::testParseInto_monthDay_feb29_newYork_startOfYear
  org.joda.time.IllegalFieldValueException: Cannot parse "2 29": Value 29 for dayOfMonth must
    ......
```

Excerpted patch for fixing bug Time-7:

```
--- 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) {
         ......
         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);
         ......
```

Fig. 25. Excerpted stack trace after executing the test cases of bug *Time-7* in Defects4J.

higher suspiciousness value than other statements, since manually debugging will first focus on such statements. ❷ **Prioritize the statements in the code methods (s_2) and classes (s_3) targeted by test cases**. It 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. Overall, we straightforwardly rerank suspicious statements exposed by GZoltar + Ochiai by prioritizing statements s_1, s_2, and s_3 over other suspicious statements for the fault localization of Avatar.

Table 8 presents the comparing results[23] of Avatar with refined fault localization (refined FL, i.e., GZoltar-0.1.1 + Ochiai + prioritization) against the normal fault localization (normal FL, i.e., GZoltar-0.1.1 + Ochiai). With the refined FL, all of the 26 bugs, that are correctly fixed by Avatar with the normal FL, still can be correctly fixed.

[23]In this experimental scenario, only the Defects4J benchmark is used as the evaluation dataset since Avatar leverages GZoltar to localize the bug position that relies on the execution traces of passing and failing test cases of each buggy program. We failed to execute GZoltar on most bugs in Bugs.jar and BEARS since we failed to compile them because of different issues (e.g., the unknown JDK version, the unclear configurations, and the unavailable dependencies listed in their pom.xml file). So, Avatar cannot fix any bugs in Bugs.jar and BEARS in the Normal-FL scenario since they cannot be localized. The QuixBugs programs are much simpler than the real-world programs, their bugs can be precisely localized by GZoltar. The bugs in QuixBugs fixed by Avatar in the Normal-FL scenario are the same as the perfect-FL scenario, thus we did not report it in the part.

AVATAR also correctly fixes 2 bugs that are plausibly fixed before. The two bugs are not accurately localized with the normal FL, which makes AVATAR modify the non-buggy code and generate the plausible but incorrect patches. In addition, AVATAR can (correctly) fix (6) 8 more bugs, as they are correctly located by our refined FL but are failed to be localized by normal FL. To sum up, the fault localization refined with the stack trace information makes AVATAR fix more bugs with a higher correctness ratio of generated patches.

Table 8. Comparison on improved bug-fixing performance of AVATAR.

FL Setting	GZoltar-0.1.1 + Ochiai		GZoltar-0.1.1 + Ochiai + prioritization	
	# fixed bugs	correctness ratio	# fixed bugs	correctness ratio
AVATAR	26/82	31.7%	(26+2+6)/(82+0+8)	37.8%

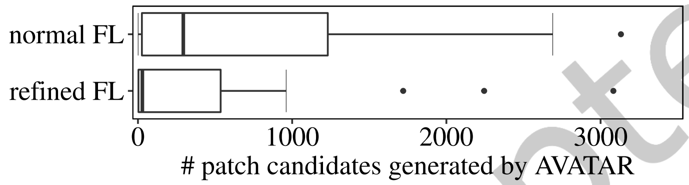


Fig. 26. Comparison on the number of patch candidates generated by AVATAR with normal and refined FL.

Looking at the efficiency of fixing bugs in terms of the number of generated patch candidates [34], shown in Figure 26, the efficiency of AVATAR is dramatically improved by generating fewer patch candidates for fixing bugs with refined fault localization 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. The results indicate that refining the fault localization with the stack trace information for AVATAR can improve its bug-fixing efficiency by generating fewer patch candidates and reducing the trials on non-sensical patch candidates.

In the manual debugging process, developers will utilize the exception thrown in the stack trace to determine the problem causing the bug. We thus explore whether the stack trace information can be used to match fix patterns in AVATAR. 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.

RQ5► *The information in stack trace after executing the bug-triggering test cases can be used to effectively improve the bug-fixing performance of AVATAR in terms of the number of the fixed bugs and the generated patch candidates. And the information in stack trace shows the potential of matching adequate fix patterns for fixing the related bugs.*

7 THREATS TO VALIDITY

A threat to external validity is related to the use of Defects4J bugs as a representative set of semantic bugs. This threat is mitigated as it is currently a widely used dataset in the APR literature related to Java. The other threat to internal validity is due to the use of Java programs as subjects. Other static tools, especially for C programs, such as Splint, cppcheck, and Clang Static Analyzer are not investigated. What's more, we only considered fix patterns mined from FindBugs and PMD violations. The other threat to internal validity is with respect to the spectrum-based fault localization setting of Avatar that requires the targeted buggy programs to have the failed and successfully executed unit tests for checking execution traces. A threat to construct validity may involve the perfect fault localization setting to assess Avatar. This threat is minimized by the other different experiments that are comparable with evaluations in the literature.

8 RELATED WORK

The software development practice is increasingly accepting generated patches [20]. Recently, various directions in literature have been explored to contribute to the advancement of automated program repair. One commonly studied direction is the pattern-based (also called example-based) APR. Kim *et al.* [18] initiated PAR as a milestone of APR based on fix templates that were manually extracted from 60,000 human-written patches. Later studies [24] have shown that the six templates used by PAR could fix only a few bugs in Defects4J. Long and Rinard also proposed a patch generation system, Prophet [38], that learns code correctness models from a set of successful human patches. They further proposed a new system, Genesis [35], which can automatically infer patch generation transformed from developer submitted patches for program repair.

Motivated by PAR [18], more effective automated program repair systems have been explored. HDRepair [24] was proposed to repair bugs by mining closed frequent bug fix patterns from graph-based representations of real bug fixes. Nevertheless, its fix patterns, except the fix templates from PAR, still limit the code change actions at abstract syntax tree (AST) node level, but are not specific for some types of bugs. ELIXIR [56] aggressively uses method call related templates from PAR with local variables, fields, or constants, to construct more expressive repair expressions that go into synthesizing patches.

Tan *et al.* [60] integrated anti-patterns into two existing search-based automated program repair tools (namely, GenProg [25] and SPR [36]) to help alleviate the problem of incorrect or incomplete fixes resulting from program repair. In their study, the anti-patterns are defined by themselves and are limited to the control flow graph. Additionally, their anti-patterns are not meant to solve the problem of deriving better patches automatically, providing more precise repair hints to developers.

More recently, CapGen [63], SimFix [16], FixMiner [21] are further proposed to fix bugs automatically based on the frequently occurred code change operations (e.g., Inserting IfStatement) (c.f., Table 3 in [16]) that are extracted from the patches in developer change histories.

So far, however, pattern-based APR approaches focus on leveraging patches that developers applied to semantic bugs. To the best of our knowledge, our approach is the first to investigate the case of leveraging patches fixing static analysis violations: they are many more, better identifiable, and more consistent.

9 CONCLUSION

The correctness of APR-generated patches and efficiency are now identified as a barrier to the adoption of automated program repair systems. Towards guaranteeing correctness and efficiency, researchers have been investigating example-based approaches where fix patterns from human patches are leveraged in patch generation. Nevertheless, such ingredients are often hard to collect reliably. In this work, we propose to rely on developer patches addressing real static analysis bugs. Such patches are concise and precise, and their efficacy (in removing the bugs) is systematically assessed (by the static detectors). We build Avatar, an APR system that utilizes fix ingredients from static analysis violations patches. We empirically show that Avatar is indeed effective in repairing programs with semantic bugs. Avatar outperforms several state-of-the-art approaches and complements others by fixing some of the Defects4J bugs which are not fixed by any APR system in the literature yet. To boost the development of APR, we further investigate the potential reason why Avatar can address semantic bugs, the bug-fixing performance of Avatar biased by different fault localization settings, and assess the possibility of utilizing the stack trace information after the execution of bug-triggering test cases to improve the bug-fixing performance of Avatar. As future work, we plan to make Avatar to be flexible in addressing new fix patterns and to integrate with the static analysis tools for resolving program defects automatically.

ACKNOWLEDGEMENTS

This work was supported by funding from the National Key R&D Program of China (2020YFB1005500), the National Natural Science Foundation of China (Grant No. 62172214), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and 2021R1I1A3048013), Additionally, the research was partially supported by Kyungpook National University Research Fund, 2020.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE, 89–98.

[2] Pavel Avgustinov, Arthur I Baars, Anders S Henriksen, Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. 2015. Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th International Conference on Software Engineering*. ACM, 437–447.

[3] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th ACM International Conference on Automated Software Engineering*. ACM, 378–381.

[4] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647.

[5] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313.

[6] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop in Automation of Software Test*. ACM, 85–91.

[7] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324.

[8] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. 2011. An experience report on using code smells detection tools. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 450–457.

[9] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 19–30.

[10] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 317–328.

[11] Sarah Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 41–50.

[12] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. ACM, 121–130.

[13] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM SIGPLAN Notices* 39, 12 (2004), 92–106.

[14] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23.

[15] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 255–266.

[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. 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, 298–309.

[17] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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, 437–440.

[18] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 802–811.

[19] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *Empirical Software Engineering* 24, 6 (2019), 4071–4106.

[20] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2017. Impact of Tool Support in Patch Construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 237–248.

[21] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[22] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535.

[23] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing automated program repair with deductive verification. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 428–432.

[24] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 213–224.

[25] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 602–614.

[27] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018).

[28] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 275–286.

[29] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. 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, 102–113.

[30] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. 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, 456–467.

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42.

[32] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*. IEEE, 658–662.

[33] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2020. A Critical Review on the Evaluation of Automated Program Repair Systems. *Journal of Systems and Software* (2020).

[34] Kui Liu, Shangwen Wang, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. 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, 625–627.

[35] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

[36] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[37] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 702–713.

[38] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 51. ACM, 298–312.

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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, 101–114.

[40] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478.

[41] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.
[42] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444.
[43] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In *Proceedings of the 10th International Symposium on Search Based Software Engineering*. Springer, 65–86.
[44] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 492–495.
[45] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. 2003. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks* 16, 5-6 (2003), 555–559.
[46] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 298–309.
[47] Naouel Moha, Yann-Gael Gueheneuc, Anne-Fran Duchien, et al. 2010. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36.
[48] Martin Monperrus. 2014. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 234–242.
[49] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.
[50] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.
[51] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.
[52] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. ACM, 609–620.
[53] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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, 24–36.
[54] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. *arXiv preprint arXiv:1803.03806* (2018).
[55] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 10–13.
[56] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 648–659.
[57] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 13–24.
[58] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.
[59] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 314–324.
[60] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 727–738.
[61] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 832–837.
[62] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.
[63] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11.
[64] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 479–490.
[65] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology* 22, 4 (2013), 31.
[66] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 660–670.

[67] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799.

[68] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426.

[69] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 512–523.

[70] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[71] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 191–200.

[72] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 52–63.

[73] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? an exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, 242–251.

[74] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841.

[75] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *Proceedings of the 1st International Workshop on Intelligent Bug Fixing*. IEEE, 1–10.

[76] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* (2018).