ReFixar: Multi-version Reasoning for Automated Repair of Regression Errors

Xuan-Bach D. Le
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
bach.le@unimelb.edu.au

Quang Loc Le
University College London
United Kingdom
quang.le@ucl.ac.uk

***Abstract*—Software programs evolve naturally as part of the ever-changing customer needs and fast-paced market. Software evolution, however, often introduces regression bugs, which unduly break previously working functionalities of the software. To repair regression bugs, one needs to know when and where a bug emerged from, e.g., the bug-inducing code changes, to narrow down the search space. Unfortunately, existing state-of-the-art automated program repair (APR) techniques have not yet fully exploited this information, rendering them less efficient and effective to navigate through a potentially large search space containing many plausible but incorrect solutions.**

In this work, we revisit APR on repairing regression errors in Java programs. We empirically show that existing state-of-the-art APR techniques do not perform well on regression bugs due to their algorithm design and lack of knowledge on bug inducing changes. We subsequently present ReFixar, a novel repair technique that leverages software evolution history to generate high quality patches for Java regression bugs. The key novelty that empowers ReFixar to more efficiently and effectively traverse the search space is two-fold: (1) A systematic way for multi-version reasoning to capture how a software evolves through its history, and (2) A novel search algorithm over a set of generic repair templates, derived from the principle of incorrectness logic and informed by both past bug fixes and their bug-inducing code changes; this enables ReFixar to achieve a balance of both genericity and specificity, i.e., generic common fix patterns of bugs and their specific contexts. We compare ReFixar against the state-of-the-art APR techniques on a data set of 51 real regression bugs from 28 large real-world programs. Experiments show that ReFixar significantly outperforms the best baseline by a large margin, i.e., ReFixar can fix correctly 24 bugs while the best baseline can only correctly fix 9 bugs.

I. Introduction

Software programs naturally evolve, e.g., having new features implemented over time, to respond to user requirements. Code changes during software evolution, however, may introduce regression bugs, which unduly break previously working functionalities of the software. Regression bugs remain challenging and prevalent in the software industry [1], [2]. Unfortunately, fixing regression bugs is time-consuming and error-prone, e.g., they can take up to 8.5 years before they are detected and fixed by developers [3], and worse still, low-quality bug fixes can even introduce new regression bugs.

To manually debug and repair a regression error, a human developer often asks "when and how did the regression bug occur?". Imagine, for example, that yesterday the program still worked, and today, after some changes have been made, it breaks some tests that were previously passing, then the changes made can be a clue for debugging. Knowing when and how a regression occurred helps understand the context of the bug, localize the potential faulty locations and narrow down the search space. This is achieved by identifying the bug-inducing changes, i.e., code changes during software evolution that introduce the bug. Often, bug-inducing changes can only be approximated as isolating bug-inducing changes from multiple changes across different versions of a program is notoriously difficult. Once bug-inducing changes have been determined, developers attempt to modify those changes to repair the bug, e.g., simply reverting the changes to the version before the bug occurred. Note that, even with just an approximate set of bug-inducing changes at hand, the search space for repairs can already be effectively narrowed down.

Regression bugs naturally fit well with the practical pipeline of automated program repair (APR). Once a regression bug occurs, there is already, at least, a failing test case that witnesses the bug. APR takes the bug-witnessing test cases as input and produces a repair that passes all tests. However, most APR techniques, unfortunately, have been designed in such a way that the useful information derived from regression bugs such as bug-inducing changes are taken for granted. GenProg and the likes [4], [5] use genetic programming to evolve the buggy program through using random mutation operators such as delete, swap, and replace statements. Template-based APR techniques, such as PAR [5], HDRepair [6], and TBar [7], mine generic repair templates from past human bug fixes and then apply the templates to generate patch candidates. These techniques lack specificity as they attempt to capture generic patterns via *multiple subject* programs. Relifix [8] repairs regression errors for C programs using repair operators derived from analysing several bug-related program versions. However, Relifix randomly generates patches and does not rank patches based on their likelihood of being correct, rendering it ineffective, as later shown in our experiments.

In this paper, we revisit APR techniques for repairing Java regression errors. We empirically show that the state-of-the-art APR techniques do not perform well on regression errors. Rationales behind this include: (1) random patch generation without effective patch ranking criteria, and (2) the lack of

knowledge on bug-inducing changes, which is a useful asset to repair regression bugs. Motivated by these insights, we propose a new regression error repair system, namely REFIXAR, that leverages the software evolution history to generate high-quality repairs. The key novelty of REFIXAR is two-fold: (1) it systematically analyzes multiple versions of the buggy program and generates abstract syntax tree (AST) mappings between arbitrary versions. This allows REFIXAR to track the evolution of each specific line of code through the history; and (2) it applies a search algorithm over a set of repair operators, derived from the principle of incorrectness logic [9] and informed by past bug fixes and bug-inducing code changes. This is empowered by the AST mappings to track changes across multiple software versions. By doing so, REFIXAR efficiently and effectively leverages the bug-inducing changes to constrain the search space to find good repairs.

We evaluate REFIXAR on 51 regression bugs from 28 real-world large Java programs, and compare REFIXAR against four state-of-the-art APR techniques: Relifix [8], TBar [7], GenProg [4] and jMutRepair embedded in ASTOR implementation [10]. Each of the baseline APR technique is representative in their categories, e.g., heuristics-based and template-based APR. To curate the benchmark, we iterate through data sets of Continuous Integration failures and collect bugs that are reproducible and witnessed by failing test cases. Experiment results show that REFIXAR significantly outperforms the baselines, i.e., while REFIXAR generates patches for 30/51 bugs where 24/30 are correct, the best baseline recommends patches for 20/51 bugs where only 9/20 are correct.

In summary, our contributions include:

- We empirically show although regression bugs naturally fit well with the practical pipeline of APR, state-of-the-art APR approaches do not perform well on a data set of 51 regression bugs on 28 real-world Java subject programs.
- Guided by the empirical results, we propose a new novel regression error repair technique, namely REFIXAR, that shows superior performance compared to state-of-the-art APR. REFIXAR tracks the evolution history of the program under repair and uses the bug-inducing changes to effectively constrain the search space.
- REFIXAR is the first APR technique specifically targeting regression bugs in Java programs. It opens up unique challenges that future research in APR for Java programs should consider addressing, including how to efficiently and effectively identify bug-inducing changes and how to best utilize the bug-inducing changes for APR (See the discussion in Section V). REFIXAR serves as the baseline for future regression error repair techniques for Java programs to build upon.

The remainder of this paper is structured as follows. Section II presents a motivating example for APR on regression bugs. Section III introduces the methodology for our approach REFIXAR, followed by Section IV that shows the evaluation of state-of-the-art APR and REFIXAR. Section V discusses future directions that help improve APR and their applications. Section VI discusses related work, and Section VII discusses threats to validity. Finally, section VIII concludes.

II. MOTIVATING EXAMPLE

In this section, we provide an example for revisiting automated program repair (APR) on regression bugs and the need for our new novel technique REFIXAR that is specifically designed for repairing regression bugs in Java programs.

Listing 1 shows a human bug fix in Traccar, a program for GPS tracking. The fix adds line 126 (denoted by "+" starting symbol), which initializes the global variable `connectionManager` as an instance of `ConnectionManager`. What could be the possible ways for this bug to be automatically fixed? Let us discuss a few scenarios below.

```
123   public static void init(DataManager dataManager) {
124       properties = new Properties();
125       Context.dataManager = dataManager;
126   +   connectionManager=new ConnectionManager();
127   }
```

Listing 1. A bug fix for Traccar, a Java system for tracking GPS.

Should there be a repair operator that adds a statement that initializes an object, a repair technique may have been able to fix this bug. However, the question that naturally follows is then "what variable to initialize and what instance should the variable be initialized to?". Although the repair operator may sound uncomplicated, it becomes not straightforward to implement, and also it is unclear whether having the repair operator would adversely impact the search space.

Another possibility for the bug to be fixed automatically is a copy-paste operator, which copies a statement from elsewhere in the same program, such as the operator proposed in GenProg [4]. Indeed, we observe that the statement `connectionManager = new ConnectionManager();` appears elsewhere in the same buggy file. We experimented with a popular implementation of GenProg in Java [10] with this bug but, sadly, GenProg was not able to fix it. Why? A closer look reveals that GenProg's operators randomly delete and add statements that render the search space intractable.

Aside from GenProg, what about other repair techniques? We experimented with Relifix [8], which we reimplemented its analogy for Java programs, TBar [7], and jMutRepair [10] with the bug, but they all failed to repair the bug. The repair techniques do not have the necessary repair operators to render the search space tractable to find the solution.

Let us now refresh that this bug is the result of a regression, meaning that all test cases had passed until some changes arrived and broke down some tests. What are those changes, and how do they impact the code and the synthesized repair? Is there any chance we can learn from the possibly wrong changes and rectify them to make the test cases pass again? To answer these questions, we traced back the evolution history of the subject program to find the bug-inducing commit, i.e., the first version in which the tests failed. Listing 2 shows

an excerpt of the changes made in the bug-inducing commit. It reveals that the bug-inducing commit changes the same buggy file and that within that same buggy file, it replaces the statement at line 92 by line 93. A closer look at the line 93 further reveals that the statement `connectionManager = new ConnectionManager();` was added, but the developer forgot to add it to line 126 as well, resulting in the regression bug and thus the bug fix in Listing 1.

It becomes clear that using the bug-inducing changes can effectively narrow down the search space for the repair of the regression bug at hand. Motivated by this example, we implemented a new repair technique, namely REFIXAR, that takes into account the bug-inducing changes to effectively and efficiently constrain the search space and derive good repairs.

```
89  @@ -90,7 +90,7 @@ public static void init(String[]
        arguments) throws Exception {
90        }
91        dataManager = new DataManager(properties);
92  -     dataCache = new DataCache();
93  +     connectionManager=new ConnectionManager();
94        if (!Boolean.valueOf(properties.getProperty("web.
            old"))) {
95            permissionsManager = new PermissionsManager();
96        }
```

Listing 2. The bug-inducing commit of the bug in Listing 1.

III. METHODOLOGY

In this section, we describe and justify the design of our approach. At a high level, REFIXAR allows for multi-version reasoning by tracking the evolution of each line of code among arbitrary program versions. In doing so, REFIXAR keeps track of which line of code was changed and how it changed. This is achieved by mappings between abstract syntax trees (ASTs) of program versions. REFIXAR then searches for good repair solutions over predefined repair operators taking into account the changes made in the bug-inducing commit to constrain the search space. Figure 1 depicts the overall framework of REFIXAR. We describe in more detail below.

A. Multi-version reasoning via AST mapping

REFIXAR allows to track the evolution of any specific line of code in the development history via AST mappings. Let us now justify the need for this reasoning.

Let $\mathbb{P}^B$ be the buggy program under repair, $\mathbb{T}^F$ be the set of tests that fail in $\mathbb{P}^B$, $\mathbb{P}^I$ be the version of the program where it is the first version that the tests $\mathbb{T}^F$ fail, $\mathbb{P}^O$ be the version right before $\mathbb{P}^I$ where the tests $\mathbb{T}^F$ still pass. The bug-inducing changes $\mathbb{C}^{O\to I}$ lie in the changes from $\mathbb{P}^O$ to $\mathbb{P}^I$. Our goal is two-fold: (1) given a potentially buggy line $\mathbb{L}^B$ in $\mathbb{P}^B$, we need to apply repair operators to $\mathbb{L}^B$ to fix the bug, and (2) to do so, we need to track the evolution of $\mathbb{L}^B$ from $\mathbb{P}^B$ back to $\mathbb{P}^I$ and $\mathbb{P}^O$ to know how $\mathbb{L}^B$ evolved from being "correct" (in $\mathbb{P}^O$) to "incorrect" (in $\mathbb{P}^I$). The bug-inducing changes $\mathbb{C}^{O\to I}$ serve as an explanation for how and why $\mathbb{L}^B$ needs to be repaired, and thus, REFIXAR uses $\mathbb{C}^{O\to I}$ to guide the repair process. Particularly, REFIXAR uses $\mathbb{C}^{O\to I}$ by breaking the changes down to smaller granularity at the AST level and using it as repair ingredients. How the repair ingredients are used depending on the repair operators used by REFIXAR and we will explain this in detail in Section III-B where the repair operators are introduced.

Let us now explain how we build the AST mapping between arbitrary versions in the evolution history of a subject program. We use GumTree [11] that allows to compare ASTs. Note that the mapping between two ASTs can only be approximate and relied on heuristics [11]. We use a combination of two approaches, including text and AST diffing, to best match a node in an AST to the associated node on another AST. Given a node $\mathbb{N}$ to build the mapping, REFIXAR first confirms if the file containing $\mathbb{N}$ has been changed. If not, obviously $\mathbb{N}$ can be mapped to exactly the same node in another AST, i.e., the same text and location in the file. Otherwise, REFIXAR uses GumTree to heuristically build an AST mapping and extract the associated node on another AST. Note that GumTree may not be able to return any associated node for a given query due to the complexity of changes between two ASTs.

```
1   -   /**
2   -    Long code comments here
3   -   */
4   -    private static OWLOntology createOntology(...) {
5   -        try {
6   -            return man.createOntology(axs);
7   -        } catch (@SuppressWarnings("unused")
        OWLOntologyCreationException e) {
8   -            return ont;
9   -        }
10  -    }
11
12      public SyntacticLocalityModuleExtractor(...) {
13          ...
14  -       ontology = checkNotNull(createOntology(man, ont
      , collect.stream()));
15  +       try {
16  +           ontology = checkNotNull(man.createOntology(
        axs));
17  +       } catch (OWLOntologyCreationException e) {
18  +           throw new OWLRuntimeException(e);
19  +       }
20      }
```

Listing 3. An example that GumTree fails to build an AST mapping.

Listing 3 gives an example code, from https://github.com/owlcs/owlapi/[1], where GumTree fails to build the AST mapping. The changes in Listing 3 involve deletion of a method spanning over several lines (from line 1 to line 10) and the changes from line 14 to lines 15-19. Given a query for finding the associated node at line 14 of the version before the changes: `ontology = checkNotNull(createOntology(man, ont, collect.stream()))`, ideally the mapping should return the node at line 16: `ontology = checkNotNull(man.createOntology(axs))` of the version after the changes. However, the AST mapping

[1] commit: 3cce7fe19517bdc0e67e5905e65bba34d5592b2c

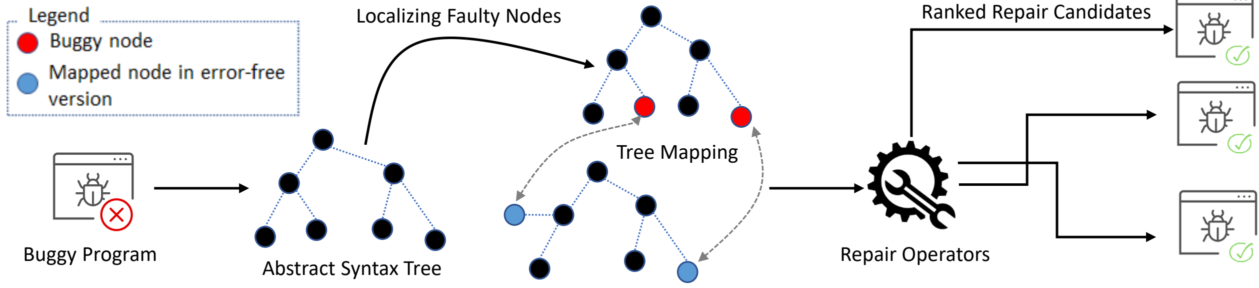

Fig. 1. Overall framework of our approach REFIXAR

built via GumTree fails to return any mapping for the node at line 14 because it looses track of node mappings due to the complex changes, e.g., the deletions of several code lines (lines 1 to 10). To overcome this issue, REFIXAR resorts to text differencing to compute the node mapping. REFIXAR uses JGit [12] to compute differences between two files at the text level. JGit allows to keep track of change actions to convert one file to another, e.g., replace line 14 by lines 15-19. Although this does not allow to precisely map line 14 to line 16, it enables REFIXAR to approximately determine the mapping, complementing GumTree where GumTree fails.

To build the AST mappings to track the evolution of code through the history of a program, REFIXAR breaks down the bug-inducing changes to AST level so that it can use them as repair ingredients for the repair operators later on. Recall that $\mathbb{C}^{O\rightarrow I}$ captures changes that induce the bug, where $\mathbb{P}^I$ is the first version that the tests fail and $\mathbb{P}^O$ is the immediate version before $\mathbb{P}^I$ where all the tests pass. REFIXAR uses GumTree to break down $\mathbb{C}^{O\rightarrow I}$ to smaller change actions and AST nodes involved. Particularly, REFIXAR leverages four change actions on AST node level to represent the transformations needed to transform one AST to another. The transformations from $\mathbb{P}^O$ to $\mathbb{P}^I$ can be represented by the following operations:

- `Update` $\mathbb{U}^{\mathbb{L}^O\rightarrow\mathbb{L}^I}$: update node $\mathbb{L}^O$ in $\mathbb{P}^O$ to node $\mathbb{L}^I$ in $\mathbb{P}^I$.
- `Insert` $\mathbb{I}^{\mathbb{L}^I}$: insert node $\mathbb{L}^I$ into $\mathbb{P}^I$.
- `Delete` $\mathbb{D}^{\mathbb{L}^O}$: delete node $\mathbb{L}^O$ from $\mathbb{P}^O$.
- `Move` $\mathbb{M}^{\mathbb{L}^O\rightarrow\mathbb{L}^I}$: move node $\mathbb{L}^O$ in $\mathbb{P}^O$ to node $\mathbb{L}^I$ in $\mathbb{P}^I$.

By doing so, the bug-inducing changes $\mathbb{C}^{O\rightarrow I}$ can be represented as a sequence of change actions to transform the AST of $\mathbb{P}^O$ to that of $\mathbb{P}^I$. For now, REFIXAR keeps $\mathbb{C}^{O\rightarrow I}$ in a bucket and uses it later on as repair ingredients for repair operators that we will discuss next in Section III-B.

B. Repair Operators

Given a potentially buggy statement $\mathbb{L}^B$ in the buggy program $\mathbb{P}^B$, the repair operators try to manipulate $\mathbb{L}^B$ to fix the bug, taking into account the evolution of $\mathbb{L}^B$ through the development history such as the program version $\mathbb{P}^I$ which is the first version where the regression bug occurs, and the program version $\mathbb{P}^O$ which is the immediate version before $\mathbb{P}^I$ wherein all tests still pass. Some operators use the bug-inducing changes $\mathbb{C}^{O\rightarrow I}$, which capture the sequence of changes from $\mathbb{P}^O$ to $\mathbb{P}^I$, to effectively constrain the search space, e.g., using $\mathbb{C}^{O\rightarrow I}$ as repair ingredients. Below, we describe the principle that justifies the design of our repair operators and the implementation of the repair operators.

The overall design of our repair operators follows a principle inspired by incorrectness logic (IL) [9]. In particular, the reachability of any program point is governed by two constraints: path presumption $\mathcal{P}$ and program state $\mathcal{S}$ (values of program variables). The path presumption $\mathcal{P}$ is a conjunction of those conditions leading to the location (e.g., branching conditions of `if` and `loop`), and program state $\mathcal{S}$ constitutes updates at assignment statements. Given a reachable location that potentially induces an error, if $\mathcal{S}$ logically implies $\mathcal{P}$ (i.e., $\mathcal{S} \Rightarrow \mathcal{P}$), then the error is feasible. To eliminate the error, we need to make some changes to the program such that the fixed program includes those $\mathcal{S}'$ and $\mathcal{P}'$ where $\mathcal{S}' \not\Rightarrow \mathcal{P}'$ and $\mathcal{P}'$ is satisfiable at the faulty location.

For illustration, let us consider the code in Listing 4.

```
1   void foo(int x) {
2     ObjectY y; // y = null by default.
3     x = 1;
4     if (x > 0) {
5 +     y = new ObjectY(); // bug fix
6       if(y.someBoolVal)
7       ...
8     }
9   }
```

Listing 4. A Null Pointer Error Example.

At line 6, we ask: “Can an NPE happen when dereferencing the variable y?”. To answer this question, we make a presumption: $\mathcal{P} \equiv x{>}0 \land y{=}null$, which presumes line 6 is reachable and an NPE happens. The program state right before line 6 comes from the updates at lines 2 and 3: $\mathcal{S} \equiv y{=}null \land x{=}1$. Since $\mathcal{S} \Rightarrow \mathcal{P}$, the method `foo` definitely contains an NPE [13]. One possible fix for this bug is to allocate a heap for y as shown at line 5. The bug fix actually mutates a value in $\mathcal{S}$, rendering a new program state $\mathcal{S}'$ such that $\mathcal{S}' \not\Rightarrow \mathcal{P}$ as $\mathcal{S}'$ can assert that y points to an object.

Based on this principle, REFIXAR implements repair operators that are classified into the following three categories.

- **W operator** to weaken $\mathcal{S}$ by, for example, removing a relevant statement.
- **S operator** to negate some conjuncts in $\mathcal{P}$ or to strengthen $\mathcal{P}$ by, for instance, conjoining additional con-

dition on a relevant condition statement or adding a new condition statement wrapping around some code.
- **M operator** to mutate values of variables in either $\mathcal{S}$ or $\mathcal{P}$ by, for example, reverting a statement to a previous one, replacing a statement by another, or modifying either the left-hand side or the right-hand side of an assignment.

Guided by this principle, REFIXAR employs 12 repair operators, **R1** to **R12**, each of which belongs to either **W**, or **S**, or **M** operator, to generate variants of the original buggy program. Among the 12 repair operators, four operators are newly proposed by us and the remaining eight operators, proposed by Relifix for C programs [8], are transferred to Java by REFIXAR. This allows us to directly study whether the operators proposed for C programs can be applied to Java programs, and compare the effectiveness of Relifix and REFIXAR. Note that it is common among the APR literature that repair operators are often borrowed from one technique to another, e.g., HDRepair [6] and PraPR [14] borrowed operators from mutation testing.

The 12 repair operators are divided into three groups: **W operator** includes **R3** and **R8**; **S operator** consists of **R4**, **R5**, **R6**, **R8**, **R11** and **R12**; and **M operator** is **R1**, **R2**, **R7**, **R9** or **R10**. We next describe the detail of each repair operator.

1) Transferring operators for C programs to Java programs: We describehow REFIXAR implements 8 repair operators that were proposed for fixing C programs in Relifix [8].

R1: Revert to previous statement. Given a potentially buggy statement $\mathbb{L}^B$ in program $\mathbb{P}^B$, REFIXAR tries to revert $\mathbb{L}^B$ to the corresponding statement $\mathbb{L}^O$ in the program version $\mathbb{P}^O$. The rationale behind this *revert* operator is that developers may have done some wrong changes that they should have left it intact, resulting in the regression bug at hand.

```
907  -   classes.add(cls);
908  +   classes.add(0, cls);
```

Listing 5. A bug fix for a regression in Apache Common Lang.

```
907  -    classes.add(0, cls);
908  +    classes.add(cls);
909       ...
910  -    if(!ignoreAccess && !MemberUtils.isAccessible(
         method)) {
911  +    if (!ignoreAccess && !MemberUtils.isAccessible(
         method)) {
912       ...
913  -    if(annotation == null && searchSupers) {
914  +    if (annotation == null && searchSupers) {
915       ...
```

Listing 6. The bug-inducing changes for the bug depicted in Listing 5.

REFIXAR implements the *revert statement* operator via AST mappings of the buggy statement $\mathbb{L}^B$, taking into account the bug inducing changes $\mathbb{C}^{O \to I}$. Particularly, $\mathbb{L}^B$ is first mapped back to $\mathbb{L}^I$ in the program version $\mathbb{P}^I$, and then $\mathbb{L}^I$ is mapped back to the corresponding statement $\mathbb{L}^O$ in program version $\mathbb{P}^O$. By this two-stage AST mapping, REFIXAR tracks down the source statement $\mathbb{L}^O$ that evolved to $\mathbb{L}^I$ and potentially created the regression bug. Once $\mathbb{L}^O$ is identified, the revert operator translates to the replacement of $\mathbb{L}^B$ by $\mathbb{L}^O$. There is a special case where $\mathbb{L}^O$ does not exist, meaning there is no associated node of $\mathbb{L}^I$ in $\mathbb{P}^O$. This can be because in the bug-inducing changes $\mathbb{C}^{O \to I}$, $\mathbb{L}^O$ was deleted from $\mathbb{P}^O$ or that $\mathbb{L}^I$ was inserted only in $\mathbb{P}^I$. In the former, REFIXAR performs the revert by adding $\mathbb{L}^I$ back to the program version $\mathbb{P}^B$, effectively reverting the deletion of $\mathbb{L}^I$. In the latter, REFIXAR performs the revert by deleting the node $\mathbb{L}^{B'}$ in $\mathbb{P}^B$ which is associated with $\mathbb{L}^I$, effectively reverting the insertion of $\mathbb{L}^I$.

Listing 5 and Listing 6 show the fix and the bug-inducing changes for a bug in Apache Common Lang. Looking at Listing 6, the developer made the wrong change at line 907 and the bug fix in Listing 5 reverts the line to the version before the regression occurs.

R2: Swap changed statement with its neighbouring statement. The goal of this operator is to identify some statement that changed in the bug-inducing changes $\mathbb{C}^{O \to I}$, find its associated statement in the buggy program version $\mathbb{P}^B$ and swap the associated statement with its neighboring statement in $\mathbb{P}^B$. The challenge here involves several AST mappings back and forth between several program versions such as $\mathbb{P}^B$, $\mathbb{P}^I$, and $\mathbb{P}^O$. Given a potentially buggy statement $\mathbb{L}^B$ in the buggy program $\mathbb{P}^B$, REFIXAR finds a set of statements $\mathbb{C}_B^{O \to I}$ that have been changed in the bug-inducing changes $\mathbb{C}^{O \to I}$ and are associated with $\mathbb{L}^B$. Note that $\mathbb{C}_B^{O \to I}$ contains statements in one or both program versions $\mathbb{P}^O$ and $\mathbb{P}^I$, e.g., delete $\mathbb{L}^O$ in $\mathbb{P}^O$ and insert $\mathbb{L}^I$ in $\mathbb{P}^I$. After that, REFIXAR randomly chooses a statement $\mathbb{L}^C$ from $\mathbb{C}_B^{O \to I}$, which belongs to either program version $\mathbb{P}^O$ or $\mathbb{P}^I$. REFIXAR then uses the AST mapping to map $\mathbb{L}^C$ back to its associated statement $\mathbb{L}^{B'}$ in $\mathbb{P}^B$. Note that due to the approximation of AST mapping, $\mathbb{L}^{B'}$ may or may not be the same as $\mathbb{L}^B$. Finally, $\mathbb{L}^{B'}$ is swapped with one of its neighboring statement in $\mathbb{P}^B$.

R3: Remove incorrectly added statement. The goal of this operator is to identify some statement that was added in the bug-inducing changes $\mathbb{C}^{O \to I}$, find its associated statement in the buggy program version $\mathbb{P}^B$ and remove the associated statement from $\mathbb{P}^B$. Similar to the operator **R2**, given a potentially buggy statement $\mathbb{L}^B$ in the buggy program $\mathbb{P}^B$, REFIXAR finds a set of statements $\mathbb{C}_B^{O \to I}$ that have been added in the bug-inducing changes $\mathbb{C}^{O \to I}$ and are associated with $\mathbb{L}^B$. After that, REFIXAR randomly chooses a statement $\mathbb{L}^C$ from $\mathbb{C}_B^{O \to I}$, and maps it back to the associated statement $\mathbb{L}^{B'}$ in $\mathbb{P}^B$. Finally, REFIXAR deletes $\mathbb{L}^{B'}$ from $\mathbb{P}^B$.

R4: Negate added condition. The goal of this operator is to identify some Boolean condition that was added in the bug-inducing changes $\mathbb{C}^{O \to I}$, find its associated node in $\mathbb{P}^B$ and negate it. Similar to **R3**, given a potentially buggy statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR finds a set of Boolean conditions $\mathbb{C}_B^{O \to I}$ that have been added in $\mathbb{C}^{O \to I}$ and are associated with $\mathbb{L}^B$. It randomly chooses a candidate $\mathbb{L}^C$ from $\mathbb{C}_B^{O \to I}$, and maps it back to the associated node $\mathbb{L}^{B'}$ of Boolean type in $\mathbb{P}^B$. Finally, REFIXAR negates the condition $\mathbb{L}^{B'}$ in $\mathbb{P}^B$.

R5: Add condition. Given $\mathbb{L}^B$ in $\mathbb{P}^B$, this operator collects a pool of Boolean conditions appearing in the same file containing $\mathbb{L}^B$ in the buggy program version $\mathbb{P}^B$. It then randomly chooses a candidate condition among the pool and adds the candidate to the condition in $\mathbb{L}^B$ if $\mathbb{L}^B$ has a condition, e.g., `if` or `loop`. Otherwise, it adds a guard condition surrounding $\mathbb{L}^B$, e.g., `if(c){`$\mathbb{L}^B$`}`.

R6: Add condition to changed expression. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR finds a set of expressions $\mathbb{C}_B^{O\to I}$ that were changed in $\mathbb{C}^{O\to I}$ and are associated with $\mathbb{L}^B$. REFIXAR then randomly chooses a candidate expression $\mathbb{L}^C$ from $\mathbb{C}_B^{O\to I}$. Note that $\mathbb{L}^C$ belongs to either program version $\mathbb{P}^O$ or $\mathbb{P}^I$, and thus, REFIXAR maps $\mathbb{L}^C$ back to its associated node $\mathbb{L}^{B'}$ in program version $\mathbb{P}^B$. REFIXAR then applies the operator **R5** on $\mathbb{L}^{B'}$.

R7: Add statement. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR collects a pool of statements appearing in the same file containing $\mathbb{L}^B$ in $\mathbb{P}^B$, randomly selects a candidate statement from the pool and adds the candidate before or after $\mathbb{L}^B$.

R8: Convert statement to condition variable statement. Given a statement $\mathbb{L}^B$ of Boolean type in $\mathbb{P}^B$, REFIXAR converts $\mathbb{L}^B$ to an `if` condition, i.e., `if(`$\mathbb{L}^B$`)`. The `if` condition will wrap around a set of sequential statements ranging from the location of $\mathbb{L}^B$ to a random sibling of it.

2) Operators proposed by REFIXAR *for Java programs:* REFIXAR implements 4 repair operators newly proposed in this paper. We describe the operators in detail below.

R9: Revert to previous expression. This operator is similar to the operator **R1**, except that it focuses in the expression level instead of statement level like **R1**. Operator **R9** prevents reverting the whole statement, which can range over several lines, and instead enables partial revert, i.e., revert of an expression which is part of a statement only. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR finds a set of expressions $\mathbb{C}_B^{O\to I}$ that were changed in the bug-inducing changes $\mathbb{C}^{O\to I}$ and are associated with (parts of) $\mathbb{L}^B$. REFIXAR randomly selects an expression $\mathbb{L}^C$ from $\mathbb{C}_B^{O\to I}$ and finds the associated node $\mathbb{L}^O$ in program version $\mathbb{P}^O$. REFIXAR also maps $\mathbb{L}^C$ to its associated node $\mathbb{L}^{B'}$ in $\mathbb{P}^B$. The revert operator now translates to the replacement of $\mathbb{L}^{B'}$ by $\mathbb{L}^O$.

R10: Replace a changed expression by another changed expression. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR finds a pool of expressions $\mathbb{C}_B^{O\to I}$ that were changed in the bug-inducing changes $\mathbb{C}^{O\to I}$ and are associated with (parts of) $\mathbb{L}^B$. REFIXAR then randomly chooses an expression $\mathbb{L}^C$ from the pool $\mathbb{C}_B^{O\to I}$, and maps it to the corresponding expression $\mathbb{L}^{B'}$ in $\mathbb{P}^B$. REFIXAR then replaces $\mathbb{L}^{B'}$ with another randomly chosen expression from $\mathbb{C}_B^{O\to I}$.

```
64      if (streamingTrack.getTrackExtension(
            TrackIdTrackExtension.class) == null) {
65  -       long maxTrackId = 1;
66  +       long maxTrackId = 0;
```

Listing 7. A bug fix to illustrate operator R10.

Listing 7 and Listing 8 show a bug fix and the bug-inducing commit for the bug. The bug fix replaces the line 65 by line 66, which indeed changes the expression on the right-hand side of the assignment from `1` to `0`. The bug-inducing commit contains the ingredients needed for repair, i.e., the expression `0`, which is an expression that was changed in the bug-inducing commit (see the comparison `ts.size() > 0` on right-hand side of line 66). The operator **R10** takes the changed expressions in the bug-inducing commit as repair ingredients to effectively constrain the search space, resolving the bug presented in Listing 7.

```
64      if (streamingTrack.getTrackExtension(
            TrackIdTrackExtension.class) == null) {
65          ...
66  -       TrackIdTrackExtension tiExt = new
            TrackIdTrackExtension(ts.size() > 0 ? (ts.get(ts.
            size() - 1) + 1) : 1);
67  +       long maxTrackId = 1;
68  +       for (Long trackId : trackIds) {
69  +           maxTrackId = Math.max(trackId, maxTrackId);
70  +       }
```

Listing 8. An excerpt of bug-inducing commit for the bug in Listing 7.

R11: Add null-check guard for changed expression. This operator helps repair bugs where the developer changes some expression but forgot to add a null check for the changed expression. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR finds a set of expressions $\mathbb{C}_B^{O\to I}$ that were changed in the bug-inducing changes $\mathbb{C}^{O\to I}$ and are associated with (parts of) $\mathbb{L}^B$. REFIXAR then randomly chooses an expression $\mathbb{L}^C$ from $\mathbb{C}_B^{O\to I}$, and maps $\mathbb{L}^C$ to its corresponding expression $\mathbb{L}^{B'}$ in $\mathbb{P}^B$. REFIXAR collects all object names appearing in $\mathbb{L}^{B'}$ and adds a null-check guard for a randomly chosen object name, e.g., `if(c != null){`$\mathbb{L}^B$`}` where `c` is an object name appearing in $\mathbb{L}^{B'}$.

R12: Add try-catch for changed statements. This operator wraps a try-catch around a statement that was changed in the bug-inducing changes $\mathbb{C}^{O\to I}$ to prevent an exception from crashing the program under repair. Given a statement $\mathbb{L}^B$ in $\mathbb{P}^B$, REFIXAR checks if $\mathbb{L}^B$ was changed in $\mathbb{C}^{O\to I}$. If so, REFIXAR simply adds a try-catch surrounding $\mathbb{L}^B$.

C. Generating and exploring the search space

In this section, we discuss how REFIXAR uses the repair operators introduced in Section III-B to generate the search space and how it traverses the search space to find repairs.

Algorithm 1 depicts the generation and traversal of search space in REFIXAR. REFIXAR first performs fault localization using a spectrum-based technique, namely Ochiai [15], which outputs a ranked list of potentially buggy locations/lines. REFIXAR groups the lines as three groups, including: (1) lines that lie in the methods that were changed in bug-inducing changes $\mathbb{C}^{O\to I}$, (2) lines that lie in the files that were changed in $\mathbb{C}^{O\to I}$, and (3) others. REFIXAR iterates through each group in order, and within each group, it then iterates through each line in the descending order of the likelihood of a line being

faulty. Given a potentially buggy line $\mathbb{L}^B$ in the buggy program version $\mathbb{P}^B$, REFIXAR iterates through all repair operators and find operators that are applicable at $\mathbb{L}^B$, e.g., operator **R10** (*Replace a changed expression by another changed one*) would be inapplicable at $\mathbb{L}^B$ if the bug-inducing changes $\mathbb{C}^{O \to I}$ do not include those expressions that are associated with (parts of) $\mathbb{L}^B$, meaning $\mathbb{C}_B^{O \to I}$ is empty. REFIXAR applies the operators at $\mathbb{L}^B$ several times to create candidate repairs. To do so, REFIXAR is parameterized by the number of times ($\mathbb{N}$) that a location can be mutated and the maximum number of solutions ($\mathbb{M}$) that it should collect. Particularly, REFIXAR randomly chooses an operator from the applicable operators, applies the operator on $\mathbb{L}^B$, and repeats this process $\mathbb{N}$ times. REFIXAR stops whenever it finds $\mathbb{M}$ solutions.

REFIXAR reports the solutions that it found and then ranks them by the ascending order of the number of transformations needed to transform the buggy program to the solution. This ranking mechanism follows the Occam's Razor principle that the simpler solution is more likely to be correct [16]. We note that the number of transformations is measured by using AST transformation actions provided by GumTree [11].

Algorithm 1: Algorithm to generate and traverse the search space by REFIXAR

Input:
- $\mathbb{N}$: number of mutants to create on a location
- $\mathbb{M}$: maximum number of solutions to collect
- $\mathbb{R}^O$: set of repair operators
- $\mathbb{P}^B$: buggy program under repair
- $\mathbb{L}$: potentially faulty locations

Output: $\mathbb{S}$: Solutions/Repairs

```
1  S ← {}
2  foreach L^B ∈ L do
3  |  R^A ← {}
4  |  foreach R ∈ R^O do
5  |  |  if R is applicable at L^B then
6  |  |  |  R^A ← R^A ⋃ R
7  |  |  end
8  |  end
9  |  for i ← 0 to N do
10 |  |  R ← choose_one_operator(R^A)
11 |  |  X ← apply_mutation(R, L^B)
12 |  |  if X passes all tests then
13 |  |  |  S ← S ⋃ X
14 |  |  |  if size_of(S) = M then
15 |  |  |  |  sort(S)
16 |  |  |  |  return
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

IV. EVALUATION

In this section, we describe the data set that we use for our experiments, the baseline automated program repair (APR) techniques that we compare with our technique REFIXAR, experiment settings, and experiment results.

A. Data set, baselines, evaluation metrics and settings

Data set. We collect a data set of 51 regression bugs from 28 large real-world Java programs. The largest and smallest programs in the data set have ≈200K and ≈52K lines of code, respectively. Table I shows details of the five largest subject programs in our dataset. For each program, #Bugs depicts the number of bugs, #Commits the number of commits in the development history, and #LOC the number of lines of code.

TABLE I
TOP FIVE LARGEST SUBJECT PROGRAMS IN OUR DATA SET.

Project	#Bugs	#Commits	#LOC
INRIA/spoon	5	3.1K	200K
raphw/byte-buddy	2	5.2K	170K
FasterXML/jackson-databind	7	7K	120K
traccar/traccar	11	6.3K	65K
openpnp/openpnp	3	3.1K	52K

To collect this data set, we iterate through two existing data sets for Continuous Integration (CI) failures, namely Bears [17] and BugSwarm [18]. Criteria to collect our benchmark include: (1) each bug involves fixes to Java source files, and (2) there is at least one bug-witnessing test case in the *existing* test suite that reveals the buggy behavior of the program. Bears and BugSwarm contain CI failures, each of which can be a compilation error or a test case failure. We filter out failures that involve fixes to files that are not Java source, e.g., build configuration files, and only retain bugs that satisfy our two criteria described above. We next filter out bugs that we cannot reproduce, e.g., bugs that require complex environment settings. Furthermore, to ensure the retained bugs are indeed regression bugs, we assure that each bug has a program version before the bug occurs that passes all the test cases (including the bug-witnessing tests). To do this, we traverse back along the development history before a bug happens and find the bug-inducing commit, i.e., the first commit where the bug-witnessing test cases fail. We do this by using `git bisect` testing. The `git bisect` does a binary search starting from the commit hash of the current buggy program version and finds the first commit that the bug-witnessing test cases fail. The buggy program version, the bug-witnessing test cases and the bug-inducing commits are then used as input to REFIXAR and the baselines.

Baseline techniques. We experiment with REFIXAR against state-of-the-art APR techniques, each of which is representative in their category: TBar [7], the most recent state-of-the-art template-based repair technique; GenProg [4], a well-known classic repair technique using genetic programming; jMutRepair [10], a mutation-based technique; and Relifix [8], a regression repair technique originally designed for C programs. TBar [7] uses several repair templates for Java programs. The templates are manually derived from past successful human bug fixes. GenProg [4] uses random mutation operators such as delete, replace, append or swap nodes on abstract syntax tree level and uses these operators and genetic programming

to evolve the buggy program. GenProg relies on the so-called *redundancy* assumption that it assumes that the fix ingredients appear elsewhere in the same program, e.g., a null check added at some location but forgotten to be added to another location. jMutRepair [10] uses random mutation operators to mutate the buggy program to create repair candidates. None of the above techniques (TBar, GenProg, and jMutRepair) take into account the evolution of the buggy program under repair, and thus they take for granted the bug-inducing changes. Relifix [8] was proposed to fix regression errors in C programs. We reimplemented the Relifix's repair operators (as discussed in Section III-B) and its repair algorithm such that they work on Java programs. We discuss further the differences between Relifix and our technique REFIXAR and the rationale behind their performances in Section V.

Evaluation metrics. To evaluate the effectiveness, we measure the patch generation rate and correct patch rate for each APR technique on our benchmark, following [19]. Patch generation rate measures the number of bugs that an APR technique can generate patches for out of the total number of bugs considered in experiments. Correct patch rate measures the number of correct patches out of the total number of patches generated by an APR technique. The higher the patch generation rate and correct patch rate, the better.

Experiment settings. We run all repair techniques on the same machine (8GB of RAM and Intel(R) Core(TM) i7-6820HK CPU). We set one hour as the time limit for each run. For REFIXAR, each repair run of each bug is terminated as soon as REFIXAR finds five solutions. Note that other repair techniques used in our experiments do not generate multiple repair solutions in the same run. We will discuss this limitation of the baseline techniques in detail in Section V.

B. Research questions and experiment results

Research questions. We answer three research questions:

- **RQ1:** How effective are state-of-the-art APR and REFIXAR on the data set of 51 Java regression errors? To answer this question, we evaluate REFIXAR and the four baseline techniques on our benchmark.
- **RQ2:** How is the contribution of each repair operator to the success of REFIXAR? To answer this question, we investigate which ones help REFIXAR repair more bugs.
- **RQ3:** Case studies on the effectiveness of the APR techniques. What are cases where they fail and succeed? What should be improved to increase the APR effectiveness?

Experiment results. We report the results for the above research questions below.

RQ1: repair effectiveness. We compare REFIXAR against *Relifix* [8], *TBar* [7], *GenProg* [4], and *jMutRepair* [10], which we explained in detail in Section IV-A. Table II shows the results on the effectiveness of the tools (where PGR denotes the patch generation rate and CPR denotes the correct patch rate). The results show that REFIXAR significantly outperforms the baselines in terms of patch generation rate (PGR) and correct patch rate (CPR). Particularly, REFIXAR has a PGR of ≈59% (30/51), followed by *Relifix* with ≈39% (20/51) of PGR, *TBar* with ≈ 33% (17/51) of PGR, *GenProg* and *jMutRepair* of ≈16% (8/51) and ≈6% (3/51) respectively. In terms of correct patch rate (CPR), REFIXAR achieves ≈80% (24/30), followed by *Relifix* with ≈45% (9/20), *TBar* with ≈41% (7/17), *jMutRepair* with ≈30% (1/3), and *GenProg* with ≈0% (0/8). We note that the results for *jMutRepair* and *GenProg* are in line with recent studies, e.g., [10] and [20]. All bugs that are fixed by the baseline techniques are also fixed by REFIXAR.

TABLE II
RESULTS OF APR TOOLS ON 51 JAVA REGRESSION BUGS.

Rate	REFIXAR	GenProg	jMutRepair	TBar	Relifix
PGR	30/51	8/51	3/51	17/51	20/51
CPR	24/30	0/8	1/3	7/17	9/20

RQ2: contribution of repair operators. In this research question, we attempt to understand deeper which repair operators (described in Section III-B) contribute the most to the superior performance of REFIXAR. Out of the 12 operators proposed in Section III-B, the operator *Revert to previous statement* (R1) is the most effective, which helps REFIXAR generated patches for 9 bugs, out of which 7 of them are correctly fixed by REFIXAR. The next useful operators are R5, R11, and R12. Some operators do not help REFIXAR fix any bugs, e.g., operators R2 and R8. Indeed, R2 and R8 are transferred from C to Java. This implies that operators that work for C programs may not directly transfer to Java programs.

RQ3: Case studies. We present some case studies and discuss on why some techniques succeed and fail to repair the bugs.

Listing 9 shows a human-written bug fix for LANG-1317 issue in Apache Commons Lang. The bug fix replaces the method call `classes.add(cls)` by an overloaded method call `classes.add(0, cls)`, which has an additional parameter, namely the constant `0`, to the call. REFIXAR and *Relifix* generate the exact same fix to the human-written fix. One may be surprisingly skeptical and ask "*Where does the constant parameter `0` come from so that the tools can automatically fix the bug?*". It might be challenging to find the ingredients from which an automated repair tool can use to generate repair candidates. Indeed, traditional APR tools such as *GenProg*, *jMutRepair*, and *TBar* failed to fix this bug because the repair ingredients do not appear in their search space. There is no exact statement `classes.add(0, cls)` appearing elsewhere in the same program and thus *GenProg* failed to fix the bug due to its *redundancy* assumption. *jMutRepair*, despite having an operator to mutate a method call, could not find the ingredients, e.g., the constant `0`, to put in place. Similarly, *TBar* failed for the same reason.

The true power of REFIXAR and *Relifix* comes from the ability to analyze over the development history of the program under repair, utilizing the bug-inducing changes as repair ingredients to its repair operators. Particularly, among dozens of lines of code changes in the bug-inducing changes, REFIXAR detects that the bug-inducing commit touches

the line `classes.add(cls)`, in which it was changed from `classes.add(0, cls)`. REFIXAR uses the AST mappings that it built to track down the evolution of the line `classes.add(cls)`, detecting the suspicious change which could be responsible for the bug. An operator *revert* then helps REFIXAR repair this bug, effectively reverting the buggy line to its corresponding statement in the version before the bug occurs. By utilizing the bug-inducing changes to constrain the search space, REFIXAR efficiently and effectively locate the repair ingredients needed to repair the bug at hand.

```
905   List<Class<?>> classes = (searchSupers ?
          getAllSuperclassesAndInterfaces(cls)
906           : new ArrayList<Class<?>>());
907   -    classes.add(cls);
908   +    classes.add(0, cls);
```

Listing 9. An excerpt of a bug fix for Apache Commons Lang

Listing 10 shows a human-written bug fix for which all APR tools used in our experiments failed to produce a fix. The bug involves moving an `if` statement in a `for-loop` body. Should there be a repair operator that *moves* a sub-tree in an abstract syntax tree, a repair technique might have been able to fix this bug. REFIXAR traces back to the bug-inducing commit, which appears to not even touch the buggy file, and thus REFIXAR fails to repair this bug because of the reliance on the bug-inducing commit that it found. However, whether `git bisect` testing that REFIXAR relies on is accurate in finding bug-inducing commit or not is still a question. We conjecture that the accuracy of the method to find bug-inducing commit affects the performance of REFIXAR. We will discuss this more as a future work in the Section V.

```
373   for (...) {
374   +     if (buffer.readIdProperty(propName)) {
375   +         continue;
376   +     }
377   ...
378   -     if (buffer.readIdProperty(propName)) {
379   -         continue;
380   -     }
```

Listing 10. An excerpt of a bug fix for jackson-databind

V. Discussion & Future work

We discuss the key differences between REFIXAR and *Relifix* and the rationale behind the superior performance of REFIXAR. We also discuss the challenges and opportunities.

REFIXAR shares the similar spirit as *Relifix*, in which they both rely on multi-version reasoning for automated repair. However, there are several important obstacles that prevent *Relifix* to reach the performance of REFIXAR. First, *Relifix*'s algorithm to generate and traverse the search space is conservative, blocking a repair operator from being reused during repair process once the operator creates a candidate that is not compilable [8]. Furthermore, *Relifix*'s search space is limited by the fact that it only returns one solution that it found to pass all tests. We argue that a repair operator may generate several candidates, of which many do not compile, but the repair operator may ultimately help generate a good repair at the end. By realizing this, REFIXAR's algorithm does not block repair operators like *Relifix*, allowing REFIXAR to more flexibly explore the search space and find several good quality solutions. Next, REFIXAR ranks the solutions based on the widely-accepted Occam Razor principle, which prefers simpler solutions that involve fewer transformations on abstract syntax trees to create repairs. Indeed, this helps REFIXAR rank the correct solutions among top five for all the bugs (24 bugs) that it can correctly fix. Last but not least, *Relifix* was proposed for repairing regression bugs in C programs and our experiment partly showed that some operators for C programs do not directly transfer to repairing Java programs.

REFIXAR opens up several challenges and opportunities for automated repair of regression errors and related research areas. First, REFIXAR can serve as the baseline technique for automated repair of Java regression errors that future techniques can build upon. Second, future work can address how much the accuracy of methods that find bug-inducing changes (BIC) affects REFIXAR. In other words, REFIXAR can be used as an underlying technique to automatically assess the effectiveness of methods that find BIC, e.g., SZZ [21], as opposed to manual analysis done in recent studies [22]. That is, a BIC finder is considered better than others if it helps REFIXAR repair more bugs. We believe that this is an interesting direction that REFIXAR enables us to explore.

REFIXAR also enables an automatic way to compare the accuracy of tree differencing methods such as GumTree [11] and ChangeDistiller [23]. REFIXAR's current implementation uses GumTree to build mappings between Abstract Syntax Trees (ASTs) and to represent transformations needed. Indeed, assessing correctness or performance of AST differencing methods is a challenging task as it is tedious and error prone to manually assess them on AST level. By using REFIXAR, one can automatically and objectively assess the accuracy of tree differencing algorithms in the sense that a better tree differencing algorithm would help REFIXAR repair more bugs. We plan to incorporate several AST differencing methods other than GumTree and perform this study in the near future.

Another future work would be to combine REFIXAR with the compositional analysis using incorrectness logic [9]. This analysis was designed to find bugs in continuous integration systems (CI) over large codebases. We envision that multi-version reasoning APR such as REFIXAR is a good fit to close the loop for bug catching and repair for CI. Indeed, such an integration of static reasoning into syntactic reasoning of REFIXAR not only backs a theory for those pre-defined operators or even a proof search for sound repairs, but also would help further reduce the repair search space.

VI. Related Work

The closest to our work is *Relifix*, a technique for repairing regression errors in C programs [8]. As discussed in Section V, REFIXAR and *Relifix* share a similar spirit but differ in generating and traversing the search space. Moreover, in our

experiments, we showed that some repair operators proposed for C programs do not directly transfer to Java programs. The work in [24] studies how bugs arise through the bug-inducing commits and recommends patches based on only revert operator. Ming et al. [25] study the correlations of commits between bug-inducing and bug-fixing. Based on that, they can obtain specific fix patterns. In line with these works, REFIXAR uses bug-inducing changes to constrain the search space. Different from them, REFIXAR builds AST mappings to track the evolution of each line of code. The AST mappings help REFIXAR form a compact search space and thus enable it to tractably find good solutions.

Bohme et al. [3] propose a collection of regression errors in C programs together with a comprehensive study on the complexity of regression errors. Their data set was later on used in the experiments of *Relifix* [8]. We followed steps in [3] to construct our data set of regression errors for Java programs and also to find bug-inducing commits. Sliwerski et al. [21] propose a classical technique, namely *SZZ*, to uncover bug-inducing changes based on `git blame`. REFIXAR, in contrast, uses `git bisect` testing to find bug-inducing changes. REFIXAR can help complement recent studies on assessing the accuracy of *SZZ* and its variants [25], [26].

The works in [27]–[30] generate context-sensitive patches consistent with inferred specifications. While SemFix [27], Angelix [31], and S3 [29] use symbolic execution and constraint solving to generate specifications based on a test suite, [28] relies on static deductive verification with specifications in the form of Hoare triples. Different from these works, REFIXAR purely bases on syntactic reasoning via Abstract Syntax Tree mappings. We, however, envision that we can enhance REFIXAR with a semantic analysis for better accuracy.

PAR [5] introduces templates that were manually derived from human-patches. TBar [7] revisits and improves the template-based repair approach proposed in PAR. Recent approaches explored ways to automatically mine bug fix patterns and use them for program repair, e.g., [6], [32], [33]. HDRepair uses data mining to mine repair templates and use mutation operators to generate the search space [6]. It uses the repair templates to traverse the search space in a way that repair candidates that more frequently match with bug-fix templates in the history are ranked higher. DeepFix [33] proposes to use sequence-to-sequence deep learning for program repair. It first encodes a program as a sequence of primitive commands and then searches for minimal transformations to fix the buggy programs. Similarly, SequenceR [32] was also based on the sequence-to-sequence approach. While DeepFix concentrates on C programs, SequenceR focuses on Java programs. PraPR [34] proposes to use mutation operators on the bytecode level for program repair. SimFix proposes to use existing patches and similar code for program repair [35]. While these approaches complement REFIXAR, none of them performs multi-version reasoning and uses bug-inducing changes to constrain the search space like REFIXAR.

VII. THREATS TO VALIDITY

External validity. Threats to external validity correspond to the generalizability of our findings. Our study considers 51 regression bugs from 28 large real-world Java programs and 4 popular baseline APR techniques. Still, this may not represent all APR and bugs and thus may affect our study's generalizability. To mitigate this risk, we constructed the data set of real bugs from many subject programs and experimented with APR techniques that are representative in their categories, e.g., heuristic-based and template-based APR. In the future, we plan to experiment with more subject programs and tools.

Internal validity. Threats to internal validity refer to possible errors in our implementation and experiments. To mitigate this risk, we have carefully examined our implementation and experiments and investigated the rationale for the obtained results via the three research questions. We re-implemented *Relifix*'s algorithm following its original paper [8]. We confirmed with the first author of *Relifix* via email correspondence on implementation details that need clarifications to ensure that our implementation of *Relifix* is as close as possible to its algorithm presented in its paper. We have also manually analyzed our constructed data set to ensure that each bug used in our experiment is indeed a reproducible regression.

Construct validity. Threats to construct validity correspond to the suitability of our evaluation. We manually examined the machine-generated patches to assess their correctness, which may be subject to potential biases. To mitigate this risk, we have double-checked the results carefully. Employing an independent test suite can help further improve the confidence in patch validation, although obtaining an independent test suite can be difficult for large real-world programs. We plan to explore this further in future work.

VIII. CONCLUSION

We presented REFIXAR, the first automated repair tool specifically designed to target regression bugs in Java programs. REFIXAR renders a tractable search space by tracking the evolution history of each line of code, utilizing the bug-inducing changes and a compact set of repair operators. We compared REFIXAR against four state-of-the-art automated repair techniques, each of which is the representative in their categories, on a data set of 51 regression bugs in 28 large real-world Java programs. Experiment results showed that REFIXAR is significantly superior compared to the baselines in terms of patch generation rate and correct patch rate. We answered three research questions to analyze the rationale behind the superior performance of REFIXAR.

In the future, we would study how the effectiveness of methods that find bug-inducing changes, e.g., SZZ [21], affects REFIXAR. Next, we also plan to use REFIXAR to evaluate the accuracy of tree differencing algorithms as we discussed in Section V. We also plan to study the use of automated test case generation tools for program repair as in [36]. Overall, we believe that REFIXAR can serve as a useful common ground that underpins these future research challenges.

References

[1] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[2] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *International Conference on Product Focused Software Process Improvement*. Springer, 2010, pp. 3–16.

[3] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 105–115.

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

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

[6] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER) 2016*. IEEE, 2016.

[7] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[8] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 471–482.

[9] P. W. O'Hearn, "Incorrectness logic," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371078

[10] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[11] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[12] *JGit*. [Online]. Available: https://www.eclipse.org/jgit/

[13] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O'Hearn, and J. Villard, "Local reasoning about the presence of bugs: Incorrectness separation logic," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 225–252.

[14] A. Ghanbari and L. Zhang, "Prapr: practical program repair via bytecode mutation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1118–1121.

[15] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[16] J. Skilling, *Maximum Entropy and Bayesian Methods: Cambridge, England, 1988*. Springer Science & Business Media, 2013, vol. 36.

[17] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019. [Online]. Available: https://arxiv.org/abs/1901.06024

[18] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.

[19] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

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

[21] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[22] Y. Fan, D. A. da Costa, D. Lo, A. Hassan, and L. Shanping, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2020.

[23] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[24] M. Wen, Y. Liu, and S.-C. Cheung, "Boosting automated program repair with bug-inducing commits," in *In the International Conference on Software Engineering, New Ideas and Emerging Results Track*, ser. ICSE-NIER:20. New York, NY, USA: Association for Computing Machinery, 2020.

[25] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 326–337. [Online]. Available: https://doi.org/10.1145/3338906.3338962

[26] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 772–781.

[28] X. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 428–432.

[29] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[30] ——, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 376–379.

[31] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[32] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transaction on Software Engineering*, 2019.

[33] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

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

[35] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

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