JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2019; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Discovering Common Bug-fix Patterns: A Large-Scale Observational Study

Eduardo C. Campos, Marcelo de A. Maia

Faculty of Computing, Federal University of Uberlândia, Uberlândia, MG, 38400-902, Brazil

SUMMARY

Background: Automatic program repair aims to reduce costs associated with defect repair. The detection and characterization of common bug-fix patterns in software repositories play an important role in advancing this field. *Aim:* In this paper, we characterize the occurrence of known bug-fix patterns in Java repositories at an unprecedented large scale. Furthermore, we propose a novel automatic technique for unveiling frequent and isolated repair actions corresponding to realistic bug fixes in Java. *Method:* The study was conducted for Java GitHub projects organized in two distinct data sets. The first data set (Boa) contains more than 4 million bug-fix commits from 101,471 projects. The second data set (Defects4J) contains 369 real bug fixes from five open-source projects. *Results:* We characterized the prevalence of the five most common bug-fix patterns (identified in the work of Pan *et al.*) in those bug fixes. The combined results showed direct evidence that developers often forget to add `IF` preconditions in the code. *Conclusion:* We discover a total of 155 repair actions from Defects4J patches and discuss 10 pervasive repair actions that occur across all analyzed Java projects. Moreover, the overall *Precision* and *Recall* values for the clustering approach were 0.62 and 0.64, respectively. Copyright © 2019 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Locality-Sensitive Hashing, Bug-Fix Patterns, Automated Program Repair

1. INTRODUCTION

There are more bugs[†] in real-world programs than human programmers can realistically address [1]. The battle against software bugs exists since software existed. Substantial effort is spent to fix bugs. For instance, Kim and Whitehead [2] report that the median time for fixing a single bug is about 200 days. Program evolution and repair are major components of software maintenance, which consumes a daunting fraction of the total cost of software production. Research in *automatic program repair* has focused on reducing defect repair costs. A family of techniques has been developed around the idea of "test-suite based repair" [1]. The goal of test-suite based repair is to generate a patch that makes failing test cases pass and keeps the other test cases satisfied [3]. Recent test-suite based repair approaches include the work by Le Goues *et al.* [1], Nguyen *et al.* [4], Kim *et al.* [5].

The cost of debugging and maintaining software has been continuously increasing [6]. A 2013 study estimated the global cost of debugging at $312 billion, with software developers spending half their time debugging [7]. Several recent studies have established the potential of automatic program repair to reduce costs and improve software quality. The systematic study of GenProg is a notable example, which measured cost reduction in actual dollars [1]. Currently, this recent research

*Correspondence to: eduardocunha11@gmail.com, marcelo.maia@ufu.br
[†]In this paper, we use the terms "defect", "error", "fault", and "bug" as synonyms.

direction attracts much academic and industrial attention. Nonetheless, many people question the positive results. For instance, Qi *et al.* [8] have shown that the evaluation of GenProg suffers from a number of important issues, and call for more research on systematic evaluations of test-suite based repair. Moreover, existing approaches (e.g., GenProg [1], PAR [5]) seem to be able to fix only simple bugs, due to several limitations [9].

Evidence shows that not all statements are modified equally, and there are benefits from using history-based data [5][10]. The fault localization process [11, 12] can benefit from assigning a higher priority to the statements most commonly modified to fix bugs [13].

A key point of automated program repair research consists of decreasing the time to navigate the repair search space [14]. Kim *et al.* [5] introduced PAR, an algorithm that generates program patches using a set of 10 manually written fix templates. We share with PAR the idea of extracting repair knowledge from human-written patches. However, the foundations of their approach contains more manual work than ours because, in this paper, we devise an approach that can automatically extract the fix templates that will serve as input to an automatic program repair technique instead of manually extracting them.

Automatic program repair has shown promise for reducing the significant manual effort debugging requires. However, there is a deficit of earlier evaluations of automatic program repair techniques caused by repairing programs and evaluating generated patches' correctness using the same set of tests (i.e., the patches overfit to the training test suite) [15].

While automatic program repair shows great promise, it is far from being widely adopted, and still many potential improvements remain to be made [6]. In a recent work, Martinez *et al.* [16] assessed the effectiveness of different automatic repair approaches on the real-world Java bugs of *Defects4J* [17]. They conducted a study with three automatic repair systems on 224 bugs present in this data set: jGenProg, an implementation of GenProg [1] for Java; jKali, an implementation of Kali [8] for Java; and NOPOL [3]. Their results showed that these repair systems together can synthesize a patch for only 21% of these bugs.

Prior work already demonstrated how to extract recurring bug-fix patterns from commits (e.g., Pan *et al.* [18]). In fact, the work of Pan *et al.* [18] is closely related to ours. Below, we present the main similarities and differences between the two works:

- We both identify automatically extractable repair actions of software;
- We both analyze software history data available in software configuration management systems to find patterns in bug fix changes;
- We both compute the frequency of occurrence of each bug fix pattern across all Java projects;
- The main difference is that our repair actions are discovered fully automatically based on AST differencing (there is no prior manual analysis to find them) and clustering. Pan and colleagues manually analyzed part of the bug fix change history of five open source projects to define a set of bug fix patterns. This analysis involved inspecting the bug hunks and the corresponding fix hunks in the bug fix revisions, and classifying bug fix changes into different patterns (bug types) based on the syntax component kinds in the hunk pairs and their containing program context;
- Pan *et al.* [18] use the GNU diff (GNU 2003) tool to compute the changes to each file involved in a bug fix revision. These changes represent the text difference of a file between the bug version and the fix version. The main issue with text differencing algorithms is that they cannot compute fine-grained differences (our repair actions are smaller and more atomic than Pan *et al.*'s work [18]). Indeed in many languages (such as Java) a text line can contain many programming constructs. Therefore, the bug fix patterns identified by Pan and colleagues are coarse grained and do not reveal the root cause of the bugs;
- Although both works investigate bug fix patterns, they have different goals. While our work aims to provide repair actions that can be used by some automatic program repair technique, the work of Pan and colleagues aims to automatically classify bugs into specific bug types, avoiding the traditional problems of human bug categorization.

In this paper, we study a much larger data set [19] than those two previous works with 101,471 Java projects and more than 4 million bug-fix commits. In order to make our results more

DOI: 10.1002/smr

dependable, we also conducted a qualitative analysis in the real-world Java bugs present in the *Defects4J* data set [17].

This paper aims to confront the results of automatic detection of bug-fix patterns in the *Boa* data set with the results of manual inspection of these same patterns in the *Defects4J* data set. This confrontation is important for two reasons:

1. Find out if there is any bug-fix pattern that has high prevalence in both analyzed data sets. In affirmative case, such a pattern would be a strong candidate to be investigated in future automatic program repair techniques;
2. Assess the bias (i.e., noise) that can be introduced in a fully automated analysis in the *Boa* data set, since the automatic detection of these bug-fix patterns depends directly on the correctness of the *Boa* programs.

We analyzed the bug-fix commits of Java programs, taken from several million human-made bug fixes from GitHub. This software repository contains an enormous collection of software and information about software [19]. We used a domain-specific programming language called *Boa* [19] to analyze ultra-large-scale data efficiently.

In a previous conference paper [20], we have shown that developers often forget to add `IF` preconditions in the code. One evidence is that the bug-fix pattern that most appeared in the analyzed bug-fix commits of both data sets (i.e., *Boa* and *Defects4J*) was IF-APC (Addition of `IF` Precondition Check). Furthermore, we make observations to directly guide future research in automatic repair of Java programs. For instance, our findings suggest that test-suite based program repair may need to consider multi-language programming and bugs in non-source files. We have extended that paper in the following main points:

1. We propose a novel automatic technique for unveiling the most prevalent and pervasive repair actions in Java. Our approach includes a preprocessing step based on Locality-Sensitive Hashing (LSH) to remove outliers from search space before clustering the data;
2. A novel technique for automatically learning bug fixing repair actions based on AST differencing using the state-of-the-art AST diff tool GumTree [21];
3. An extensive analysis of the content of software bug-fix commits: our analysis is novel both with respect of size (395 *real* bug fixes from six open-source Java projects present in *Defects4J* data set [17]) and granularity (155 repair actions at the level of the AST).

In this paper, we use the terms “repair template” and “repair action” to refer to two different concepts: a “repair template” may be caused by the combination of multiple factors and its fix may require a certain sequence of “repair actions”. A software repair action is a kind of modification on source code that is made to fix bugs. We can cite as examples: inserting a method invocation, changing the condition of an `IF` statement, inserting a `Catch` clause for a `Try` statement, etc.

The remainder of this paper is organized as follows. Section 2 presents the data sets we use to perform the three studies. Section 3 presents the research questions and the bug-fix patterns considered in the studies. Section 4 details the studies and provides the results that are discussed in Section 5. In Section 6, we distilled the threats to the validity of our paper. Related work is surveyed and shown in Section 7. Section 8 concludes this paper and proposes future work.

2. DATA SETS AND CHARACTERISTICS

2.1. *Boa data set*

In this paper, we use the *September 2015/GitHub* data set offered by *Boa* [19], including 554,864 non-forked Java projects with 23,226,512 commits (i.e., during the construction of the *Boa* data set, the authors of this data set excluded all Java projects that were forks). *Boa* identifies 4,590,405 as bug-fix commits distributed among 101,471 Java projects (18.2875%). In other words, 81.7125% of Java projects present in this data set do not have any bug-fix commit. In this paper, we focus our

DOI: 10.1002/smr

```
1  counts: output sum of int;
2  p: Project = input;
3
4  exists (i: int; match(`^java$`, lowercase(p.programming_languages[i])))
5  foreach (j: int; p.code_repositories[j].kind == RepositoryKind.GIT)
6    foreach (k: int; isfixingrevision(p.code_repositories[j].revisions[k].log))
7      counts << 1;
```

Figure 1. Querying number of bug-fix commits in Java GitHub projects using *Boa* language.

analysis on these 101,471 Java projects because our goal is to study bug fixes and patterns. Figure 1 shows a query written in *Boa* language that returns the number of bug-fix commits in Java GitHub projects. The built-in function `isfixingrevision` (line 6) uses a list of regular expressions to match against the revision's log (i.e., commit's log message). If there is a match, then the function returns *true* indicating the log was for a commit fixing a bug.

Figure 2 shows the distribution of bug-fix commits among 101,471 Java projects: 81% of these projects have 1 to 15 bug-fix commits, while only 9% of them have 51 or more bug-fix commits. This bar chart shows that is not common to see open-source Java projects hosted on GitHub with a large number of bug-fix commits (e.g., more than 50 bug-fix commits).

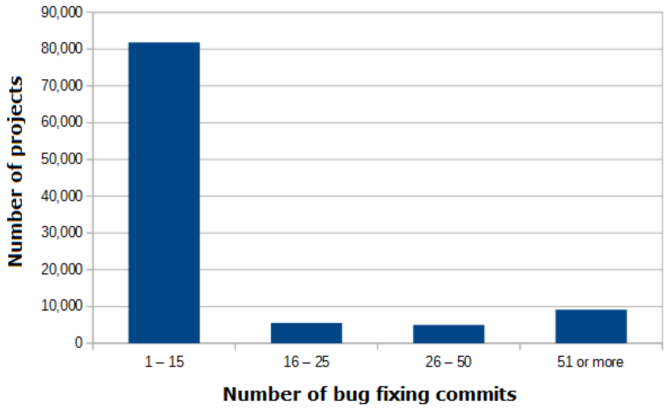

Figure 2. Distribution of bug-fix commits among Java GitHub projects.

Programming Language: Programming languages evolve over time to meet the needs of developers. This evolution is necessary to simplify common tasks and make the language easier to use [22]. The Java Language Specification (JLS) [23, 24, 25, 26] is the official specification for Java. New editions of the specification (JLS2, JLS3, and JLS4) are released as the language evolves to add new features (e.g., annotation use, enhanced-for loops, variables with generic types, etc.). Note that new language features are purely additive (each edition is fully backwards compatible with previous editions) [22].

The Java Language Specification, edition 2 (JLS2) [24] was a relatively minor update in terms of new language features. This edition added one new language feature: *assert statements*.

The Java Language Specification, edition 3 (JLS3) [25] added several significant language features, including: *annotation types*, *enhanced-for loops*, *type-safe enumerations* (enums), *generic types*, and *variable-argument methods* (varargs).

The Java Language Specification, Java SE 7 edition (JLS4) [26] made several changes, including: *binary literals*, a *diamond operator* for generic type inference, *allowing catching multiple exception types*, *suppression of varargs warnings*, *automatic resource management*, and *underscores in literals*.

As returned by *Boa*, the major language of a project is the one with the highest percentage of source code, considering the files in the project. Figure 3 shows the distribution of the analyzed

DOI: 10.1002/smr

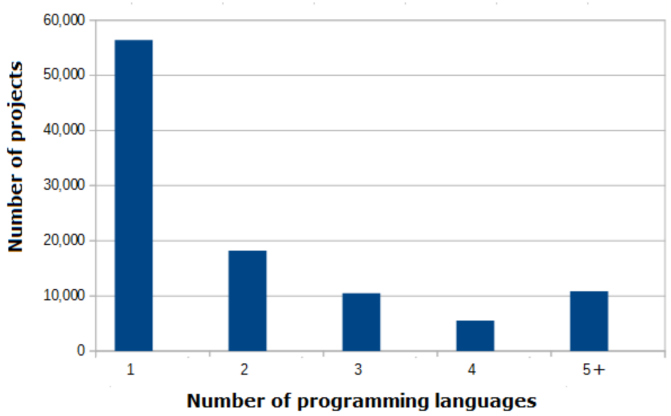

Figure 3. Number of programming languages in each Java project using GIT.

Table I. Types of changed files present in the *Boa* data set (JLS: Java Language Specification).

File Type	Total	Description
SOURCE_JAVA_JLS4	83,798	The file represents a Java source file that parsed without error as JLS4
TEXT	541,023	The file represents a text file
BINARY	752,945	The file represents a binary file
SOURCE_JAVA_ERROR	2,073,558	The file represents a Java source file that had a parse error
SOURCE_JAVA_JLS2	2,607,413	The file represents a Java source file that parsed without error as JLS2
XML	6,818,299	The file represents an XML file
SOURCE_JAVA_JLS3	15,748,967	The file represents a Java source file that parsed without error as JLS3
UNKNOWN	23,426,568	The file's type was unknown

projects per number of programming languages. As we can see, 56,414 out of 101,471 Java projects (i.e., 55.5961%) use only one programming language (i.e., Java). However, 10,837 out of 101,471 Java projects (i.e., 10.6798%) use five or more programming languages.

Types of changed files: Table I shows the types and descriptions of changed files present in the *Boa* data set and the number of changed files per file type. We consider a file *changed* if it is new, modified, or deleted in a commit. In total, 52,052,571 files were changed. As shown in Table I, the types of changed files SOURCE_JAVA_JLS2, SOURCE_JAVA_JLS3, and SOURCE_JAVA_JLS4 refer to the editions of the Java Language Specification (JLS). In our study, we decide to study these java files separately in order to understand better which edition of the specification was often changed to fix bugs.

2.2. *Defects4J data set*

In this paper, we also use *Defects4J* [17], a data set and extensible framework providing real bugs to enable reproducible studies in software testing research. Currently, *Defects4J* contains 395 real bugs from six open-source projects: JFreeChart [‡], Closure Compiler [§], Commons Lang [¶], Commons Math [‖], Mockito [**] and Joda-Time [††]. We do not use the JFreeChart project because the version control system for this project is Apache Subversion (SVN) and we decided to study only projects

[‡]JFreeChart, http://jfree.org/jfreechart/
[§]Closure Compiler, http://code.google.com/closure/compiler/
[¶]Apache Commons Lang, http://commons.apache.org/lang
[‖]Apache Commons Math, http://commons.apache.org/math
[**]Mockito, http://site.mockito.org/
[††]Joda-Time, http://joda.org/joda-time/

Prepared using smrauth.cls
DOI: 10.1002/smr

hosted on GitHub repository. For these projects, the version control system is Git and they account for a total of 369 bug fixes. Table II presents the main descriptive statistics of projects in *Defects4J*.

Table II. The main descriptive statistics of the considered bugs in Defects4J. The number of lines of code and the number of test cases are extracted from the most recent version of each project.

Program	#Bug Fixes	Source KLOC	Test KLOC	#Test Cases
Closure Compiler	133	306	179	16,998
Commons Lang	65	22	6	2,245
Commons Math	106	85	19	3,602
Joda-Time	27	28	53	4,130
Mockito	38	43	22	1,611
Total	369	484	279	28,586

Defects4J is a large, peer-reviewed and structured data set of real-world Java bugs. Each bug in *Defects4J* comes with a test suite and at least one failing test case that triggers the bug. To our knowledge, *Defects4J* is the largest open database of well-organized real-world Java bugs.

There are several advantages of using *Defects4J* for a study. Among them, we can highlight:

- **Realism**: It contains real bugs (as opposed to seeded bugs as in Nguyen *et al.* [4]; Kong *et al.* [27]);
- **Scale**: It contains bugs that reside in large software projects (as opposed to bugs in student programs as in Smith *et al.* [15]);
- **Isolated Bugs**: A fundamental challenge when collecting bugs is deciding what constitutes a bug, and what does not [17]. When interacting with version control systems, developers frequently *group separate changes into a single commit*. Herzig *et al.* [28] studied this problem and named it as *tangled code changes* [29]. Fortunately, all bug fixes present in the *Defects4J* data set do not include unrelated changes such as features or refactorings. The authors of this data set manually reviewed the source code diffs of reproducible bugs to verify that they did not include irrelevant changes (e.g., if necessary, they isolated the bug fix from the source code diff).

3. METHODOLOGY

In this section, we present the research questions that we aim to answer in this work and the bug-fix patterns investigated in our studies (i.e., Study I and II).

3.1. Bug-fix patterns

Pan *et al.* [18] identified 27 bug-fix patterns through manual inspection of the bug fix change history of seven open-source Java projects. They found that the most common categories of bug-fix patterns are Method Call and If-related. Moreover, Pan *et al.* conducted an analysis involving bugs injected by the Eclipse developers and found that the most common bug injection patterns are: Change of `IF` Condition Expression (IF-CC), Method Call with different actual parameter values (MC-DAP), Method Call with different number of parameters or different types of parameters (MC-DNP), Change of Assignment Expression (AS-CE), and Addition of `IF` Precondition Check (IF-APC). Below we detail each one of these five bug-fix patterns:

1. **Change of `IF` Condition Expression (IF-CC)**: The bug fix changes the condition expression of an `IF` condition [18]. Example:

```
- if (listBox.getSelectedIndex() == 0)
+ if (listBox.getSelectedIndex() > 0)
```


DOI: 10.1002/smr

2. **Method Call with different actual parameter values (MC-DAP)**: The bug fix changes the expression passed into one or more parameters of a method call [18]. Example:

```
- String.getBytes("UTF-8");
+ String.getBytes("ISO-8859-1");
```

3. **Method Call with different number of parameters or different types of parameters (MC-DNP)**: The bug fix changes a method call by using different number of parameters, or different parameter types. This may be caused by a change of method interface, or use of an overloaded method [18]. Example:

```
- getSolrQuery(f.getFilter());
+ getSolrQuery(f.getFilter(), analyzer);
```

4. **Change of Assignment Expression (AS-CE)**: The bug fix changes the expression on the right-hand side of an assignment statement. The expression on the left-hand side is the same in both the buggy and fix versions [18]. Example:

```
- names[0] = person.getName();
+ names[0] = employees[0].getName();
```

5. **Addition of `IF` Precondition Check (IF-APC)**: This bug fix adds an `IF` predicate to ensure a precondition is met before an object is accessed or an operation is performed. Without the precondition check, there may be a `NullPointerException` error caused by the buggy code [18]. Example:

```
- repo.getFileContent(path);
+ if (repo != null && path != null)
+    repo.getFileContent(path);
```

3.2. Research Questions

This subsection presents the four research questions considered in the study about bug-fix patterns and general features of bug-fix commits.

Motivation for $\mathbf{RQ_1}$: Current automatic program repair only modifies source files, but some bugs do not reside in source files [30]. Moreover, some modified source files are in programming languages other than Java because a project may be implemented in multiple programming languages [30]. The results of this research question highlight the importance of fixing bugs in multiple programming languages and in non-source files (e.g., configuration files).

$\mathbf{RQ_1}$: *Which file types are usually changed to fix a bug?*

The **Study I** will be conducted to answer this research question. The results will provide insights on how to improve existing automatic program repair approaches to achieve their best performance.

Motivation for $\mathbf{RQ_2}$: Current automatic program repair uses quite limited mutation operators [30]. Although it is widely known that existing approaches use incomplete operators, it is challenging to make improvements [30][31]. In this research question, we analyze the statement types used to fix real bugs. The results of this research question provide insights on designing more comprehensive operators.

DOI: 10.1002/smr

RQ_2: *Which statement types are often added or deleted by developers to fix bugs?*

The **Study I** will be conducted to answer this research question. The results will provide general features of bug-fix commits and can be leveraged by automatic patch generation algorithms to prioritize some types of statements relative to others, since some statements are more likely to appear than others in a given patch.

Motivation for RQ_3 and RQ_4: We analyzed the same bug fix patterns studied by Pan *et al.* [18] to investigate whether the frequency of occurrence of these 5 bug fix patterns persisted for a larger sample of software projects (i.e., Pan *et al.* analyzed 6,978 bug-fix commits from seven open source Java projects, while we investigated more than 4 million bug-fix commits from 101,471 open source Java projects). Our analysis revealed that the frequency of occurrence of these bug fix patterns is not the same for a larger sample of projects. In our analysis, the IF-APC pattern had the highest prevalence among these 5 patterns studied (29.2019%), while this same bug fix pattern was neither pointed out by Pan *et al.* as one of the 3 most prevalent. Moreover, although the IF-APC pattern is a common bug injection pattern, its prevalence in hunk pairs of analyzed projects is **low** (please see Table 2 of Pan *et al.*'s paper for additional details). Therefore, we have decided to study the prevalence of these bug fix patterns in a larger data set.

RQ_3: *What is the prevalence of the 5 most common bug-fix patterns identified in the work of Pan et al. [18] in the bug fixes present in the Boa data set?*

The **Study I** will be conducted to answer this research question. The identification of these bug-fix patterns is relevant to assist the researchers in the task of automatic generation of patches [5].

RQ_4: *What is the prevalence of the 5 most common bug-fix patterns identified in the work of Pan et al. [18] in the bug fixes present in the Defects4J data set?*

The research question RQ_4 is similar to RQ_3. The basic difference between them is: the answer for RQ_4 is based on manual inspection of real bug fixes from projects present in the *Defects4J* data set, while the answer for RQ_3 is based on automatic inspection of real bug fixes from projects present in the *Boa* data set. **Study II** will be conducted to answer this last research question. Moreover, the results obtained in this second study will be compared with the results from the previous study (i.e., **Study I**).

Motivation for RQ_5: The subfield of *automatic program repair* (APR) is concerned with automatically fixing bugs, without human intervention. A family of techniques has been developed around the idea of applying different repair actions to fix different kinds of bugs. All previous automatic patch generation systems work with a set of *manually crafted repair actions* [5, 32, 1, 33, 34, 35, 36, 37, 38] to patch bugs that fall within the scope of these repair actions. Each repair action represents a common way to fix a bug. However, as emphasized by Monperrus in [9], there is no apparent principle behind the collection of repair actions. They have been collected by reading bug fixes and verifying whether they would fit in the corresponding overall approach. This *ad hoc* strategy is error-prone and suffers from excessive time and effort. Specifically, our work aims to propose an automatic and systematic approach for unveiling fine-granularity fixing ingredients (e.g., expressions) in Java. Fixing ingredients are those existing code elements reused to generate fixing patches [39]. Previous studies have shown that good fixing ingredients exist more often at a finer granularity than that of statements [39, 40]. Applying mutation operators with ingredients at a finer granularity increases the

DOI: 10.1002/smr

likelihood to include the correct patches in the search space [39, 40]. Therefore, our main motivation is to contribute to the patch generation process of search-based APR approaches.

$\mathbf{RQ_5}$: *What pervasive bug fix patterns exist in Java? How do we automatically discover them?*

The **Study III** will be conducted to answer this research question. We investigated the *Defects4J* data set to perform this study because it contains isolated bugs written in Java language. Recent algorithms have been proposed based on tree structures (such as the AST) to address the challenges of code differencing. GumTree [21] and ChangeDistiller [41] are examples of such algorithms which produce edit scripts that detail the operations to be performed on the nodes of a given AST to yield another AST corresponding to the new version of the code. In particular, in this work, we build on GumTree's core algorithms for preparing an edit script as a sequence of repair actions for transforming an AST. Given a buggy version and a fixed version of a program, GumTree is claimed to build in a fast, scalable and accurate way the sequence of AST-level repair actions (a.k.a edit script) between the two associated AST representations of the program.

It is important to note that the research question $\mathbf{RQ_5}$ is not a duplicate of research question $\mathbf{RQ_2}$ since the former works on a fine-granularity at the AST node level and the mutation operators of the latter work at the **Statement** level, which are too coarse-grained to find the correct fixing ingredients. Moreover, previous studies have shown that good fixing ingredients exist more often at a finer granularity than that of statements [39, 40].

4. STUDIES AND RESULTS

In this section, we present the two studies we conducted to answer the four research questions aforementioned. We performed one study per data set (i.e., *Boa* and *Defects4J*). Study I enabled us to answer the research questions $\mathbf{RQ_1}$, $\mathbf{RQ_2}$, and $\mathbf{RQ_3}$, while Study II enabled us to answer the research question $\mathbf{RQ_4}$. Finally, the realization of Study III allowed us to answer the research question $\mathbf{RQ_5}$.

4.1. Study I: Bug fixes present in the Boa data set

We study bug-fix commits to Java programs, taken from several million human-made bug fixes from GitHub. We analyzed the prevalence of the 5 most common bug-fix patterns identified in the work of Pan *et al.* [18] in those bug-fix commits. Moreover, we investigated the nature of bug fixes in terms of what file types are often changed to fix bugs or what types of statements are frequently added or deleted to fix bugs. For this study, we considered the *September 2015/GitHub* data set offered by *Boa* mentioned above in Subsection 2.1.

4.1.1. $\mathbf{RQ_1}$: Figure 4 shows the number of bug-fix commits per file type. As shown in Figure 4, the 2 types of changed files that appear most frequently in those bug-fix commits are: SOURCE_JAVA_JLS3 and UNKNOWN. The number of bug-fix commits related to these 2 types of changed files are respectively, 2,341,344 and 2,212,030. Text and binary files are changed least frequently. This is unsurprising, since such files are often documentation, and binaries should be changed rarely. XML files in Java projects usually represent build files or configuration files (the names of the most found configuration files end with "xml" or "properties" [30]); 17.55% of analyzed bug-fix commits are related to changes in XML files. As these bugs are not related to source files, they could not be fixed by current automatic program repair techniques [30]. Rather more surprising is how frequently UNKNOWN files are changed. We deepen our analysis in these committed UNKNOWN files and found that they are related to other programming languages like: C++, C, JavaScript, Groovy, Scala, Python, etc. Although the analyzed projects were mainly written in Java, 45,057 out of 101,471 Java projects (i.e., 44.40%) use 2 or more programming languages. This results showed that 48.18% of bug-fix commits are related to changes in non-Java source files.

DOI: 10.1002/smr

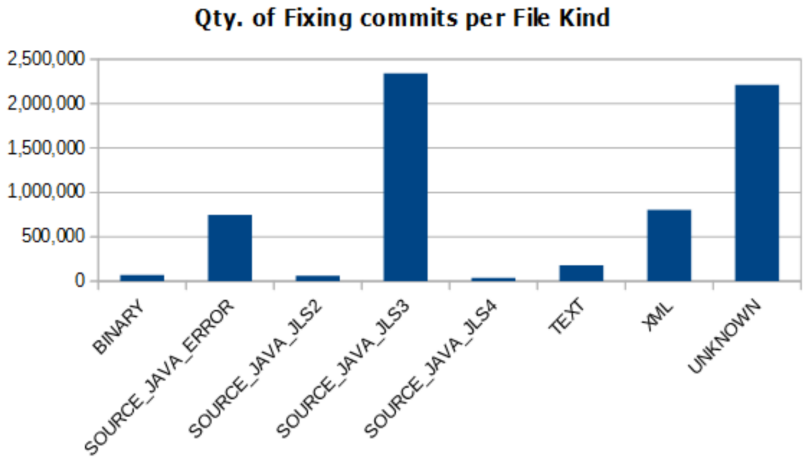

Figure 4. Number of bug-fix commits per file type.

Some modified source files are in programming languages other than Java for two reasons. First, a project may be implemented in multiple programming languages. For example, Cassandra is a database, and its programmers implement a Python driver. A bug report [‡‡] says that the driver did not parse queries correctly. To fix the bug, programmers modified one faulty line of the `cursor.py` file [22]:

```
- 39:_cfamily_re = re.compile("...", re.I | re.M)
+ 39:_cfamily_re = re.compile("...", re.IGNORECASE | re.MULTILINE |
    re.DOTALL)
```

Second, a project may implement an interface for a programming language. For example, Derby is a database that supports queries in SQL, and its programmers use SQL queries as test cases [22].

Current automatic repair approaches have been evaluated on only a limited number of programming languages, such as C and Java. However, a project may be implemented in multiple programming languages and automatic program repair may require significant improvement to fix bugs in other programming languages.

Summary of $\mathbf{RQ_1}$. We notice that many bugs reside in non-Java source files (e.g., source files of different programming languages like Scala, Groovy, PHP, etc.) or non-source files (e.g., XML files). Our results confirm the findings obtained by Zhong and Su [30] (please see the Findings 1, 2, 13, and 14 for more details). Many implementations of research techniques that automatically repair software bugs target programs written in C language (e.g., Prophet [33], GenProg [1]) or Java language (e.g., NOPOL [3], PAR [5]). Thus existing approach may be insufficient in fixing certain bugs. However, it is desirable to understand where such bugs reside, so we could investigate their nature and explore corresponding repair approaches.

4.1.2. $\mathbf{RQ_2}$: In order to find out which statement types appear more frequently in bug-fix commits, we use *Boa* to compute the number of bug-fix commits that added/deleted a particular statement type in order to solve the corresponding bug. We investigate the following 14 statement types present in the *Boa* Programming Guide: `ASSERT`, `BLOCK`, `BREAK`, `CATCH`, `CONTINUE`, `EXPRESSION`, `FOR`, `IF`, `RETURN`, `SYNCHRONIZED`, `THROW`, `TRY`, `SWITCH`, and `WHILE`. The statement type `BLOCK` is somewhat different because it was designed by *Boa* inventors to characterize a statement

[‡‡] https://issues.apache.org/jira/browse/CASSANDRA-2993

DOI: 10.1002/smr

that contains a list with two or more statements within it (e.g., the statements in the method body). Concerning the statement type `EXPRESSION`, it encompasses arithmetic or logical expressions, expressions with binary operators, etc. For a complete list, please see the Section `Expression Kind` present in the *Boa* Programming Guide.

We often need statement-level changes to understand bug fixes. For example, it requires different knowledge to fix `IF Statements` and `Return Statements`, although modified internal code elements are the same (e.g. variables) [30]. The repair actions on a code element typically increase with its complexity. For example, in the *Boa* programming guide, there are 50 expression types. In other words, the `Expression` node has a rich set of subclasses. `METHODCALL` is one of its subclasses, and it may invoke complicated API methods. By definition, each `Expression Statement` has at least an `Expression` node, and each `Return Statement` can have an `Expression` node. Thus, `Expression Statement` is complicated, and we find many API repair actions on this code element. In contrast, `Break Statement` and `Continue Statement` are relatively simple compared to the `Expression Statement` since they do not contain API elements. Thus, repair actions on these code elements are not related to APIs. All the modifications of these last two statements are to move from one line to another, since they do not have internal structures. Regarding the `Block Statement`, it refers to the statements in the method body. Note that most methods (in C-like languages, such as Java) contain a single statement of type `BLOCK`, which contains the list of statements within it. For instance, in the *Boa* programming guide, there are 20 possible statement types that could appear in the `Block Statement`.

Concerning the `IF` statement, we investigate how many bug-fix commits added or deleted null checks. The `IFNULLCHECK` statement is an `IF` statement where the boolean condition is of the form: `null == expr` OR `expr == null` OR `null != expr` OR `expr != null`. Basically, we build a query written in *Boa* language that counts how many null checks were previously in the file (previous version of the file, if exists) and how many null checks are currently in the file (actual version of the file). If there are more null checks than previously, the bug-fix commit corresponds to an addition. However, if there are less null checks than previously, the bug-fix commit corresponds to a removal. We performed this algorithm for all changed files and bug-fix commits of our *Boa* data set (i.e., 52,052,571 and 4,590,405, respectively). Concerning the 14 statement types aforementioned, we performed a similar algorithm, but considering the number of times each statement type appears in each version of a file (i.e., buggy and fix versions).

Summary of $\mathbf{RQ_2}$. This research question is important to investigate the nature of bug fixes in terms of statement types that are added or deleted to fix a particular bug. For instance, we can identify the prevalence of some statement types with respect to others. Figure 5 shows the results we obtained. As shown in this figure, there is a prevalence of `EXPRESSION`, `BLOCK`, `IF`, and `RETURN` statements with respect to the others. The median number of statements within the `BLOCK` statement type is 6. This result clearly shows the limitations of current automatic repair techniques, since many bug fixes involve adding code blocks (i.e., a list of statements). However, the majority of fixes produced by GenProg [1] are "one-liners" (i.e., changes only one line of code) [42]. As pointed out by Monperrus [9], many existing automatic program repair approaches are effective only in fixing bugs that require simple changes.

4.1.3. **$\mathbf{RQ_3}$** : Pan *et al.* [18] also discovered that there is a similarity of bug-fix patterns across projects. This indicates that developers may have trouble with individual code situations, and that frequencies of bug introduction are independent of application domain [18]. However, the main drawback of the bug-fix patterns approach stems from its automation. We therefore automatically detect these five bug-fix patterns, estimating their prevalence in the *Boa* data set presented in Section 2.

We use *Boa* language to detect common bug-fix patterns in the historical information of the projects. *Boa* provides domain-specific language features for mining source code [19]. *Boa's* capabilities are powerful, but limited in the precision it enables in detection of the aforementioned bug-fix patterns. For example, it cannot directly `diff` two files. Rather than finding exact counts of

DOI: 10.1002/smr

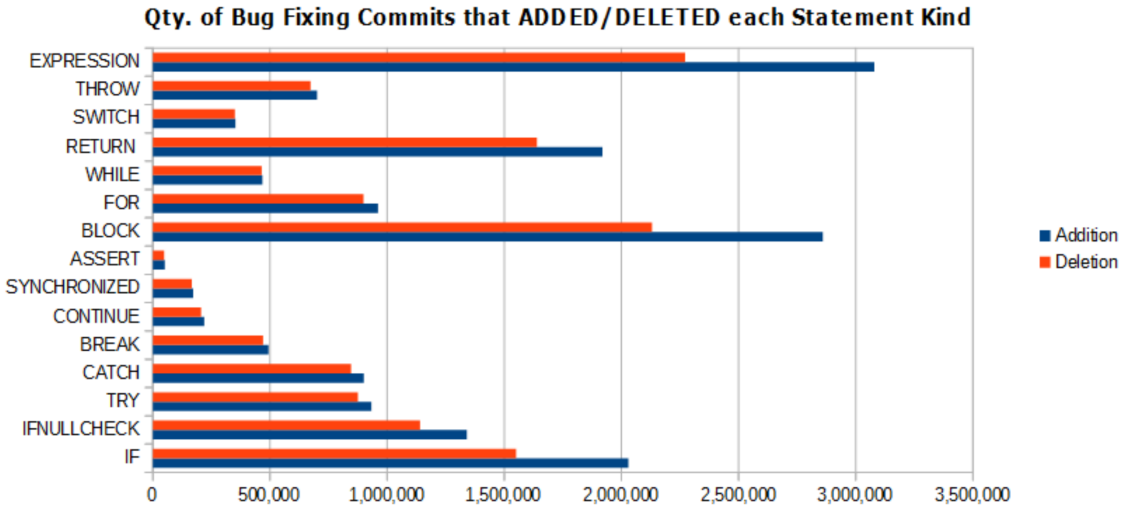

Figure 5. Number of Bug-Fix Commits that ADDED/DELETED each Statement Type.

bug-fix patterns, we approximate by processing pre- and post-fix files separately. Fortunately, these five patterns can be detected by *Boa*, as we describe below. For each pattern, we create a query written in *Boa* language. In the following paragraphs, we describe in natural language each of the five algorithms designed to detect the five bug-fix patterns described in Section 3.

1. **How many bug-fix commits change one or more `IF` Condition Expressions (IF-CC)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many `IF` conditions and expressions of these `IF` conditions appear. Then, if the number of `IF` conditions is the same between these two versions of the file (to ensure that it is a modification and not an addition or deletion), we check whether the number of expressions of these `IF` conditions is different between these two versions of the file. If it's true, the pattern was detected and the bug-fix commit is recorded. For more information of what expression types we consider, see the Section `ExpressionKind` of this page . *We found that 196,283 out of 4,590,405 (4.2759%) bug-fix commits change one or more* `IF` *condition expressions*.
2. **How many bug-fix commits change the parameter values of the method calls (MC-DAP)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many method calls appear and we also built 2 strings (i.e., one string for the pre-version and another string for the post-fix version of these file) containing the parameter values (i.e., string literals) of all method calls. Then, if the number of method calls is the same between these two versions of the file (to ensure that it is a modification), we compare if the two strings are different. If it's true, the pattern was detected and the bug-fix commit is recorded. *We found that 290,818 out of 4,590,405 (6.3353%) bug-fix commits change the parameter values of the method calls.*
3. **How many bug-fix commits change the number or parameter types of the method calls (MC-DNP)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many method calls and method parameters appear. Then, if the number of method calls is the same between these two versions of the file (to ensure that it is a modification), we check if the number of method parameters is different. If it's true, the pattern was detected and the bug-fix commit is recorded. For this pattern, due *Boa* limitations, it was not possible to identify the types of method parameters present in the

http://boa.cs.iastate.edu/docs/dsl-types.php (verified 03/08/2017)

DOI: 10.1002/smr

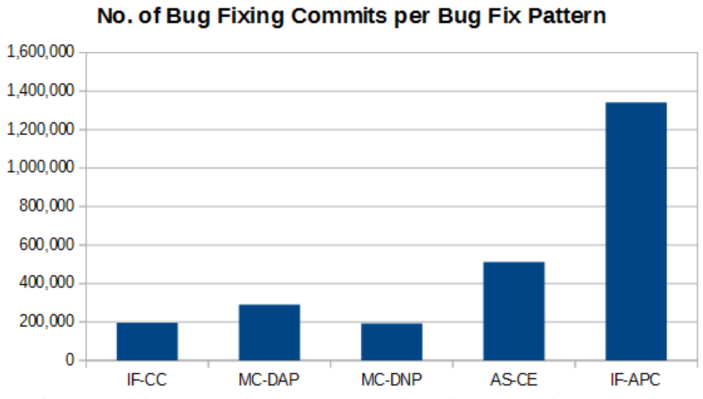

Figure 6. Number of bug-fix commits per bug-fix pattern.

method calls. *We found that 192,375 out of 4,590,405 (4.1908%) bug-fix commits change the number of parameters of the method calls.*

4. **How many bug-fix commits change one or more assignment expressions (AS-CE)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many assignment statements and expressions of these assignments appear. Then, if the number of assignment statements is the same (to ensure that it is a modification), we check if the number of expressions between these two versions of the file is different. If it's true, the pattern was detected and the bug-fix commit is recorded. For more information of what expression types we consider, see the Section `ExpressionKind` of this page. *We found that 511,299 out of 4,590,405 (11.1384%) bug-fix commits change one or more assignment expressions.*
5. **How many bug-fix commits added a null check precondition (IF-APC)?** To answer this question and to detect this pattern, for both pre- and post-fix versions of a buggy file, we count how many null checks appear. Then, if the number of null checks in the current version of the file is greater than in the previous version of these file, the pattern was detected and the bug-fix commit is recorded. *We found that 1,340,488 out of 4,590,405 (29.2019%) bug-fix commits added an* `IF` *null check precondition.*

Summary of $\mathbf{RQ_3}$. Figure 6 shows a bar chart with the number of bug-fix commits distributed among the five studied bug-fix patterns. The bug-fix pattern that appears more frequently is IF-APC (29.2019% of the analyzed bug-fix commits). Observe that several bug-fix commits match this bug-fix pattern in order to avoid *NullPointerException* errors.

4.2. *Study II: Bug fixes present in the Defects4J data set*

In order to conduct the second study, we manually reviewed the source code diffs of reproducible bugs present in the *Defects4J* data set to study the prevalence of the 5 most common bug-fix patterns identified in the work of Pan *et al.* [18]. Moreover, we counted how many bugs are If-related or Method call (the most common categories of bug-fix patterns identified in the work of Pan *et al.* [18]). Table III shows the results we obtained for each program.

Table IV shows a summary of the results of $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$. As shown in this table, there are consistencies between the two data sets (i.e., *Boa* and *Defects4J*):

1. The IF-APC pattern is the most prevalent in both;
2. The MC-DNP pattern is the least prevalent in both;
3. In general, bug fixes related to the `IF` statement type are more frequent than those related to API method calls in both.

DOI: 10.1002/smr

Table III. Programs and bug fixes per category (If-related and API Method Call) and per bug-fix pattern.

Program	#Bug Fixes	If-related	API Call	IF-CC	MC-DAP	MC-DNP	AS-CE	IF-APC
Closure	133	67	15	24	6	1	5	43
Lang	65	39	12	12	4	3	3	27
Math	106	31	11	9	4	1	16	22
Time	27	15	3	1	2	1	2	14
Mockito	38	14	6	1	1	3	1	13
Total	369	166/369	47/369	47	17	9	27	119

Table IV. Summary of the results of $\mathbf{RQ_3}$ and $\mathbf{RQ_4}$.

Bug-fix pattern	*Boa*	*Defects4J*
IF-CC	4.2759%	12.7371%
MC-DAP	6.3353%	4.6070%
MC-DNP	4.1908%	2.4390%
AS-CE	11.1384%	7.3170%
IF-APC	29.2019%	32.2493%
IF-related	33.4778%	44.9864%
API Method Call	10.5261%	12.7371%

We deepen our manual analysis in order to discover which types of addition are most common to fix bugs. Table V shows the results of these analysis. As shown in this table, the addition types that most appear to fix bugs are: *Method Addition* and *Addition of Logical or Arithmetic Expression*. For more details, please see the columns "Method" and "Logic/Arithmetic Exp." of Table V.

Table V. Manual analysis results: addition types to fix each bug per program.

Program	Try/Catch	Return	Method	Switch Case	Logic/Arithmetic Exp.	Class
Closure Compiler	1	1	8	7	6	0
Commons Lang	2	2	5	1	5	0
Commons Math	2	5	6	1	9	0
Joda-Time	2	0	0	0	0	0
Mockito	0	0	2	0	0	4
Total	7	8	21	9	20	4

Current automatic software repair approaches like GenProg [43] and PAR [5] were designed to fix simple bugs. For instance, the majority of fixes produced by GenProg modify only one line of code. A recent study on GenProg repairs [8] shows that seemingly complex repairs generated from GenProg are in the overwhelming majority of cases in fact functionally equivalent to single line modification. In other words, the majority of GenProg repairs avoid bugs simply by deleting functionality [44]. Westley Weimer, one of the inventors of GenProg, said that *the majority of fixes produced by humans are quite simple* [42]. To investigate better this affirmation, we performed another qualitative analysis involving the human-written patches (i.e., bug fixes) present in the *Defects4J* data set. Our goal was to study the size of those patches in terms of the number of lines of code (LoC) that are added to fix a particular bug. Table VI shows the main descriptive statistics of the considered patches. As shown in this table, although all considered projects have at least one complex patch whose size is 26 lines of code or more, the median and mode of the patches considering all projects are, respectively, 3 and 1 LoC. In general, the results of Table VI are aligned with Weimer's affirmation. Furthermore, recent research into the nature of software changes [45] supports the following observation: *Small changes to the repository such as one-line additions often represent bug fixes*. However, as previously discussed in $\mathbf{RQ_2}$, many bug fixes involve adding code blocks (i.e., a list of statements). For these bugs, current automatic repair techniques are not effective. Thus, greater effort is required in this direction.

Summary of $\mathbf{RQ_4}$. Table III shows the results we obtained for each program and per bug-fix pattern. Our manual analysis showed that 45% of these bugs are If-related, 12.73% are Method

DOI: 10.1002/smr

Call, and 7.31% are related to assignment expressions. The remaining bugs (i.e., 34.96%) occur due to different causes (e.g., missing a Try/Catch statement). In our work, the bug-fix pattern that most appeared in both data sets was IF-APC (Addition of `IF` Precondition Check).

Table VI. The main descriptive statistics of the patches considered in Defects4J per program.

Patch size (LoC)	Closure	Lang	Math	Time	Mockito	All
Minimum	0	0	0	1	1	0
Median	3	3	3	7	4	3
Maximum	37	43	49	26	29	49
Variance	30.89	65.28	42.81	40.48	54.43	43.99
Mode	1	1	1	5	1	1
Average	4.82	6.359	5.533	8.346	7.108	5.794
Standard Deviation	5.558	8.08	6.543	6.362	7.378	6.632
Total	133	65	106	27	38	369

4.3. Study III: Automatic Discovery of Fine-grained Repair Actions in the Defects4J data set

The **Study III** of our work adopts one of the most popular algorithms for performing approximate or exact Near Neighbor Search in high dimensional spaces based on the concept of *Locality-Sensitive Hashing* (LSH) [46]. As previously stated, we use LSH to remove outliers from search space before clustering the data. The next subsection motivates the use of LSH in this work.

4.3.1. Motivation for the use of Locality-Sensitive Hashing The problem of finding duplicate documents in a list may look like a simple task - use a hash table, and the job is done quickly and the algorithm is fast. However, if we need to find not only exact duplicates, but also documents with differences such as typos or different words, the problem becomes much more complex. In our case, each patch (a.k.a. repair) represents a document composed by fine-grained repair actions. Thus, we have to exclude many repairs that are not repetitive across projects in order to reduce the search space in which we look for frequently occurring bug patterns. In other words, because bug patterns have consistent repairs, we can use LSH to measure the similarity between similar source code changes in project histories.

A fundamental data-mining problem is to examine data for "similar" items. The problem of finding textually similar documents can be turned into a set problem: it is possible to measure the similarity of sets by looking at the relative size of their intersection. This notion of similarity is called "Jaccard Similarity" [47]. An important class of problems that "Jaccard similarity" addresses well is that of finding textually similar documents in a large corpus or a collection of patches. First, let us observe that testing whether two documents are exact duplicates is easy; just compare the two documents character-by-character, and if they ever differ then they are not the same. However, in many applications, the documents are not identical, yet they share large portions of their text.

Another important problem that arises when we search for similar items of any kind is that there may be far too many pairs of items to test each pair for their degree of similarity, even if computing the similarity of any one pair can be made very easy. That concern motivates a technique called "locality-sensitive hashing," for *focusing our search on pairs that are most likely to be similar* (i.e., Near-Neighbor Search). We should understand that the aspect of similarity we are looking at here is character-level similarity, not "similar meaning", which requires us to examine the words in the documents.

Let's give an example to better motivate the use of the LSH technique in this paper. Suppose your goal is to detect duplicate questions in a long list of questions. First, we need to define a method of determining whether a question is a duplicate of another. For this, it will be necessary to use some distance metric for strings. The Jaccard index performs sufficiently for this use case. This metric is an intersection over a union. We count the amount of common elements from two sets, and divide

DOI: 10.1002/smr

by the number of elements that belong to either the first set, the second set, or both. For example, assume that we have the two following questions:

1. *"Who was the first ruler of Poland"*;
2. *"Who was the first king of Poland"*;

The size of the intersection is 6, while the size of the union is 6 + 1 + 1 = 8, thus the Jaccard index is equal to 6 / 8 = 0.75. We can conclude the more common words, the bigger the Jaccard index, the more probable it is that the two questions are a duplicate.

The task of finding nearest neighbors is very common. Approximate algorithms to accomplish this task has been an area of active research. LSH is one such algorithm and it has many applications, including:

- **Near-duplicate detection**: LSH is commonly used to deduplicate large quantities of documents, webpages, and other files;
- **Genome-wide association study**: Biologists often use LSH to identify similar gene expressions in genome databases;
- **Large-scale image search**: Google used LSH along with PageRank [48] to build their image search technology VisualRank [49];
- **Audio/video fingerprinting**: In multimedia technologies, LSH is widely used as a fingerprinting technique A/V data.

In the next subsection, we give a background of LSH to understand better the workings of this algorithm.

4.3.2. Background on Locality-Sensitive Hashing: LSH has been around for some time, dating back as far as 1999 [50] exploring its use for breaking the "curse of dimensionality" in nearest neighbor query problems. Since then, various applications of LSH have been proposed [47]. Figure 7 shows the overview of the preprocessing step of our approach (i.e., Locality-Sensitive Hashing of Patch Code) to eliminate outliers before clustering the data.

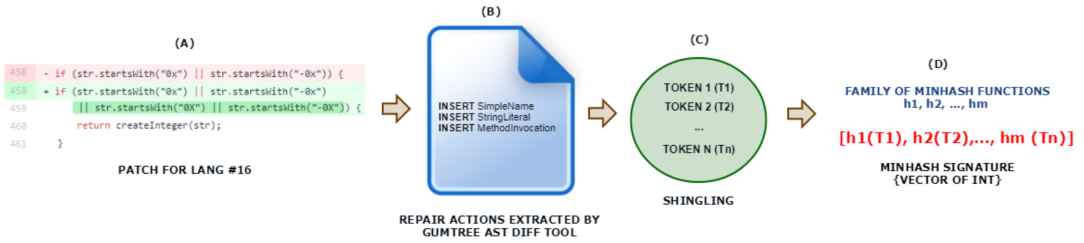

Figure 7. Overview of the Locality-Sensitive Hashing of Patch Code.

LSH has been previously used for similarity search over Twitter data [51]. Specifically, the paper applied LSH to Twitter data for the purpose of first story detection, i.e. those tweets that were highly dissimilar to all preceding tweets. In this paper, we have applied LSH in patch code (more specifically, we use LSH in the fine-grained source code changes that represent a patch).

Improved LSH algorithms [52] were proposed using a consistent weighted sampling method "where the probability of drawing identical samples for a pair of inputs is equal to their *Jaccard similarity*" [52]. The *Jaccard similarity* of sets S and T is the ratio of the size of the intersection of S and T to the size of their union (Equation 1). This metric is used to measure the similarity between two sets S and T (e.g., containing words from two documents).

$$Jaccard\_Sim(S,T) = \frac{|S \cap T|}{|S \cup T|} \tag{1}$$

The most effective way to represent documents as sets, for the purpose of identifying lexically similar documents is to construct for the document the set of short strings that appear within it. A

DOI: 10.1002/smr

document is a string of characters. Define a k-shingle for a document to be any substring of length k found within the document. Then, we may associate with each document the set of k-shingles that appear one or more times within that document [47]. We can pick k to be any constant we like. However, if we pick k too small (e.g., $k = 1$), then we would expect most sequences of k characters to appear in most documents. In this work, our corpus of documents is based on repair actions extracted from patches. Usually these documents are small. We note that each document (i.e., a document represents a patch) has the average size of an email message without attachments ($\approx$ 59 kilobytes in size according to a 2003 study). We have adopted the same criterion followed by Leskovec and Rajaraman [47] to define the shingle size, i.e., since our body of documents is as small as a body of documents composed of emails, picking $k = 5$ for the shingle size is sufficient.

LSH is a technique for fitting very large feature spaces into unusually small places. Likewise even smaller feature spaces can also benefit from the use of LSH by drastically reducing required search times and disk space requirements. Instead of storing and searching against all available raw data or even random samples of all raw data, we can use LSH techniques to create very compact signatures which replace storing all of the features typically required for such searches. The important property of signatures is that we can estimate the *Jaccard similarity* of two sets from the signatures alone. Moreover, using the signatures produced by LSH exponentially reduces both storage space and processing time requirements for similar item searches [47].

A form of LSH called *Minhashing* [53] was used to compute very compact signatures from the documents. It reduces the feature space size using a family of random hashing functions to hash each individual piece of raw input data retaining only the minimum values produced by each unique hashing function. These signatures are computed for each document by *Minhashing* the document a number of times. Given a word-document matrix in which each column represents a document and each row indicates the presence/absence (1/0) of a word in a document; *Minhashing* would be choosing for each column, from a permutation of the rows, the row number of the first row which has a value 1 in that column, a process that is typically repeated a number of times. If we do this m times, each with a different permutation, the size of the signature would be m (Fig. 7 (d)). To make this practical, the random permutations of the matrix can be simulated by the use of m randomly minhash functions (Fig. 7 (d)). Given a row r in the word-document matrix, we use hash function h() to simulate the permutation of r to the position h(r). For example, let r be the third row, and let a hash function be h(x) = 2x + 1 mod 5, then h(r) = 2 * 3 + 1 mod 5 = 2 (r is permuted to the second row according to this hash function).

The original works on LSH were [50] and [54]. Andoni and Indyk [46] summarize proposals in this field. To compare our documents for similarity using LSH, it was necessary to implement a *Minhashing* approach. This approach is highly scalable and can be extended to much larger data sets of bugs (e.g., thousands of software errors). We followed the five basic steps shown below to implement such approach:

1. The first step is to create a family of m unique hashing functions. To accomplish this task, we have used FNV (**Fowler–Noll–Vo**) hash [55], a hashing method which uses XOR bit shifting to create the seeded hash values. This hashing method has a very low chance of collision;
2. Each word or text token identified during tokenization [56] will be hashed by each unique hashing function. We use a technique called *Shingling* for text tokenization;
3. The minimum hash value produced by each unique hashing function for all words within each document processed will be retained within a minimum hash signature representing the unique characteristics of each document processed;
4. The minimum hash signatures for each document can be intersected to produce an accurate approximation of *Jaccard similarity* [52][47];
5. Longer minimum hash signatures (i.e., additional unique hashing functions) will produce more accurate approximations of *Jaccard similarity* [47].

LSH differs from conventional and cryptographic hash functions because it aims to maximize the probability of a "collision" for similar items [47]. LSH hashes input items so that similar items map to the same "buckets" with high probability [46]. Hence, LSH can detect similar or near-duplicate

DOI: 10.1002/smr

patches in a large data set of bug fixes. In our work, we used LSH to discard the patches that do not occur repeatedly because they do not represent a pattern. This preprocessing step is critical to the elimination of outliers from the data that will be clustered in the next step of our approach.

4.3.3. Methodology of Study III: In order to conduct the **Study III**, we performed **five** steps, which are described below:

1. **Reducing the Search Space**: One way is to search for repair actions that are frequently repaired by developers. This can be done by inspecting source code changes in project histories. There is, however, a problem with this method. An inspection of all the bug repairs in one project's history by a human might take several days for a project with a few thousand commits. This means a manual inspection of a sufficient set of projects representative for a programming language is not feasible. We must therefore reduce the search space in which we look for frequently occurring repair actions. Because some repair actions occur repeatedly across projects, we can reduce the search space by grouping bug-fix commits based on their repair actions. These actions can be observed automatically by extracting the source code changes of a bug-fix commit [57]. We focus on bug-fix commits rather than bug reports because developers often omit links from commits to bug reports or do not create bug reports in the first place [58].
2. **Removing Outliers from the Search Space:** The search space should now exclude many patches that are not repetitive across projects because they do not represent a frequent repair action and they would include noise in the generated clusters. However, there may still be many repair actions that do not occur frequently or related ones that are fragmented throughout the search space, making manual inspection challenging. We used Locality-Sensitive Hashing (LSH) to address this problem efficiently. In other words, we turned the problem of textual similarity of documents into one of set intersection. We should understand that the aspect of similarity we are looking at here is character-level similarity, not "similar meaning", which requires us to examine the words in the documents and their uses. However, textual similarity also has important uses. Many of these involve finding duplicates or near duplicates. We used a form of LSH called *Minhashing* [53] to compress large documents into small signatures and preserve the expected similarity of any pair of documents. In our work, a document represents a patch present in the *Defects4J* data set [17]. Each patch is composed by repair actions (i.e., fine-grained source code changes extracted using the state-of-the-art AST diff tool GumTree [21]). We use this tool because it implements an efficient AST differencing algorithm that takes into account move actions. With the use of this state-of-the-art AST diff tool, we extract all *fine-grained source code changes* that occur in a given patch. To be more precise, source code changes at the level of AST nodes: inserting nodes, removing them, moving them or updating their value. These fine-grained changes provide an additional level of precision compared to traditional, more coarse-grained, line-based diffs [59]. The main advantage in using the AST granularity is that the edit script directly refers to the structure of the code [21]. Thus, all displayed text tokens (i.e., repair actions) sampled from a document now become a small collection of integers (i.e., a signature) which will be stored and used for all subsequent similar patch comparisons. As stated earlier in Section 4.3.2, LSH hashes input items so that similar items map to the same "buckets" with high probability [54]. We can say two patches (i.e., documents) are similar if their sets of repair actions have a high *Jaccard similarity* [47] (i.e., a similarity above 80% in our case). The idea is that most of the dissimilar pairs (i.e., outliers) will never hash to the same "bucket", and therefore they will be discarded. In our study, 23 out of 395 (5.8227%) patches were considered outliers;
3. **Grouping Cross-project Repair Actions:** Our interest in this work lies in repair actions that are detectable across multiple projects. We use the following definition in this paper:

> **Definition 1 (Cross-project Repair Action)**. *An action in source code that fixes incorrect behavior, has a consistent repair, and occurs across multiple projects.*

DOI: 10.1002/smr

Our goal is to group bug-fix commits with the same repair action. Because we do not have a prior knowledge about what common repair actions exist, to achieve this goal we perform cluster analysis using machine learning. The challenge we face is selecting the best feature vector and clustering algorithm such that (1) the number of bug-fix commits a human must inspect is minimized and (2) the number of repair actions recalled by an inspection of the clusters is maximized. Ideally, each cluster would contain all instances of one repair action (perfect recall) and only instances of one repair action (perfect precision).

4. **Extracting Basic Repair Actions:** For a changed file, the state-of-the-art AST diff tool GumTree [21] compares the ASTs before and after the changes to derive the fine-grained changes. We build a Python script to extract the URLs that contain the buggy and fixed versions of a file for all bug-fix commits present in the *Defects4J* data set [17]. Our Python script and the presentation URLs are publicly available. In other words, for each bug-fix commit (c) in a project's history ($\mathbb{C}$), we obtain the set of all modified source code files in the bug-fix commit. This gives us a set $\mathbb{F} = \{f_{b1}, f_{r1}\} \ldots \{f_{bn}, f_{rn}\}$ of n {buggy file, repaired file} pairs. For each pair in $\mathbb{F}$, we compute basic repair actions that were made to the source code using *Abstract Syntax Tree* (AST) differencing [21]. Because it considers the program structure when computing the changes between f_b and f_r, AST differencing is more accurate than line level differencing. It also computes fine-grained changes by labeling each node in the AST; this fine granularity is useful for learning basic repair actions. The product of AST differencing is an AST for f_b (AST_b) and an AST for f_r (AST_r). For each $\{\text{AST}_b, \text{AST}_r\}$ pair, we extract the repair actions made to source code that occur in AST_b, but do not occur in AST_r, and vice versa. Algorithm 1 is the pseudo-code for extracting basic repair actions from a set of bug-fix commits. The output of Algorithm 1 is a set of repair actions that occurred in patches belonging to *Defects4J* projects. Each repair action present in this set is converted to a feature (a.k.a.. attribute), while each bug-fix commit is converted to an instance of a feature vector in the ARFF file [60]. For each bug-fix commit, Algorithm 2 computes the basic repair actions as well as the number of times each repair action occurs in a given patch (i.e., the frequency of the repair action). The outputs of Algorithms 1 and 2 are used as inputs for the Algorithm 3. Algorithm 3 is the pseudo-code for generating the ARFF file. Figure 8 shows an overview of the steps followed to generate an instance of a feature vector in the WEKA file.

ALGORITHM 1: Basic Repair Action Extraction

Input: $\mathbb{C}$ (bug-fix commits);
Output: $allRepairActions : Set < RepairAction >$;
$allRepairActions \leftarrow new\ HashSet < RepairAction > ()$;
$repairActionsPerDiff : Set < RepairAction >$;
$generator : ActionGenerator$;
foreach $c \in \mathbb{C}$ **do**
  $\mathbb{F} \leftarrow ModifiedFiles(c)$;
  foreach $\{f_b, f_r\} \in \mathbb{F}$ **do**
    $\{AST_b, AST_r\} \leftarrow ASTDiff(f_b, f_r)$;
    $generator \leftarrow new\ ActionGenerator(AST_b, AST_r)$;
    $repairActionsPerDiff \leftarrow generator.getRepairActions()$;
    foreach $repairAction \in repairActionsPerDiff$ **do**
      $allRepairActions.add(repairAction)$;

5. **Clustering and Ranking:** Our *goal* is to discover what common repair actions exist given a set of bug-fix commits. Depending on the problem, we can have a much larger number of bug-fix commits. Because we do not have a priori knowledge about what repair actions exist, we perform cluster analysis using machine learning to achieve this goal. Thus, we group bug-fix commits with the same repair action by clustering the WEKA file obtained in **Step 4**. The used clustering algorithm is DBSCAN [61], because (1) it is a density-based algorithm, i.e., it groups feature vectors that are closely related, and (2) unlike other clustering algorithms, it does not require the number of clusters to be provided in advance as an input. The WEKA

DOI: 10.1002/smr

ALGORITHM 2: Frequency of Repair Actions Per Patch

```
Input: ℂ (bug-fix commits);
Out: patches : Map < Integer, Map < RepairAction, Integer >>;
patches ←new HashMap <> ();
actionFreqMap : Map < RepairAction, Integer >;
repairActionsPerDiff : Set < RepairAction >;
generator : ActionGenerator;
patchId ← 0;
frequencyAction ← 0;
foreach c ∈ ℂ do
    actionFreqMap ←new HashMap <> ();
    patchId ← patchId + 1;
    𝔽 ← ModifiedFiles(c);
    foreach {f_b, f_r} ∈ 𝔽 do
        {AST_b, AST_r} ← ASTDiff(f_b, f_r);
        generator ← new ActionGenerator(AST_b, AST_r);
        repairActionsPerDiff ← generator.getActions();
        foreach repairAction ∈ repairActionsPerDiff do
            frequencyAction ← Frequency(repairAction);
            actionFreqMap.put(repairAction, frequencyAction);
    patches.put(patchId, actionFreqMap);
```

ALGORITHM 3: Generation of ARFF File

```
Input: allRepairActions : Set < RepairAction >;
patches : Map < Integer, Map < RepairAction, Integer >>;
PATH : String;
fileName : String;
Output: WEKA File (Attribute-Relation File Format);
writer : FileWriter;
writer ← new FileWriter(PATH);
actionFreqMap : Map < RepairAction, Integer >;
qtyRepairActions ← allRepairActions.size();
count ← 0;
frequency ← 0;
writer.write('@RELATION' + fileName);
foreach repairAction ∈ allRepairActions do
    writer.write('@ATTRIBUTE' + repairAction + 'REAL');
    writer.write(NEW_LINE);
writer.write('@DATA');
writer.write(NEW_LINE);
foreach patchId ∈ patches do
    count ← 0;
    actionFreqMap ← patches.get(patchId);
    foreach repairAction ∈ allRepairActions do
        if actionFreqMap.containsKey(repairAction) then
            frequency ← actionFreqMap.get(repairAction);
            writer.write(frequency);
        else
            writer.write('0');
        if count ≠ (qtyRepairActions − 1) then
            writer.write(',');
        count ← count + 1;
    writer.write(NEW_LINE);
writer.close();
```

software contains a basic implementation of DBSCAN that runs in quadratic time and linear memory. We use Manhattan distance as our distance function, because it computes shorter distances between bug-fix commits with the same basic repair actions. Repair actions are ranked by the number of times that they occurred in each bug-fix commit, considering all

DOI: 10.1002/smr

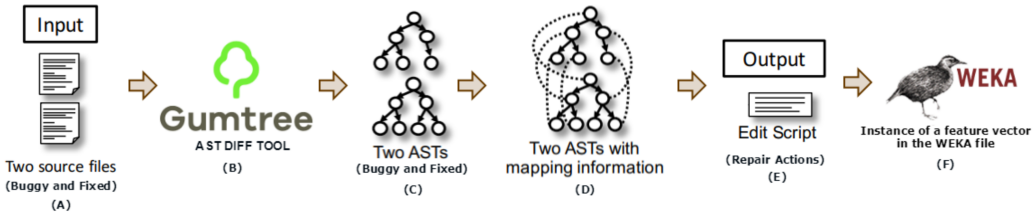

Figure 8. Overview of the steps followed to generate an instance of a feature vector in the ARFF file.

bug-fix commits belonging to *Defects4J* projects. In total, we discover 155 repair actions from those bug-fix commits.

In order to remove infrequent repair actions (i.e., noise features), we performed Feature Selection for Clustering to select important features for the underlying clusters [62]. In this paper, we use a filter method available on WEKA software to evaluate feature subsets and choose the best subset for clustering by considering their effect on the underlying clusters [62]. A filter method, by definition, is independent of clustering algorithms, and thus completely avoids the issue about lack of unanimity in the choice of clustering criterion [62]. The filtering process selected 114 attributes (41 attributes were considered irrelevant to the clustering process). The WEKA file used in this study is available online.

After this pre-processing step, we cluster the WEKA file aforementioned to obtain all instances for a given repair action. DBSCAN [61] is a density-based algorithm which discovers clusters with arbitrary shape. However, it requires the specification of two input parameters which are hard to guess [63]: *epsilon*, which specifies how close points (i.e., in our case a point corresponds to a bug-fix commit) should be to each other to be considered a part of a cluster; and *minPoints*, which specifies how many neighbors a point should have to be included into a cluster. Both parameters have a significant influence on the clustering results. We combined Binary Differential Evolution [64] and DBSCAN algorithm to simultaneously quickly and automatically specify appropriate parameter values [65]. For the parameter values, *epsilon* = 0.9 and *minPoints* = 6, DBSCAN yields a clustering comprising 10 clusters. The total of clustered and unclustered instances were respectively, 318 and 54.

4.4. Study III: Cluster Evaluation and Results

External evaluation measures try capture the extent to which points from the same partition appear in the same cluster, and the extent to which points from different partitions are grouped in different clusters [66].

As the name implies, external measures assume that the correct or ground-truth clustering is known *a priori*. The true cluster labels play the role of external information that is used to evaluate a given clustering [66]. The ground-truth clustering is given as $\mathbb{T} = \{T_1, T_2, \ldots, T_k\}$, where the cluster T_j consists of all the points with label j. For clarity, henceforth, we will refer to $\mathbb{T}$ as the ground-truth partitioning (a.k.a. ground-truth classification), and to each T_j as a partition.

We assume that the ground truth clustering is given by the 10 most frequent AST-level repair actions occurring in 395 bug fixes from *Defects4J* data set before running the clustering algorithm. Thus, all repair actions are ranked by the number of times that they occurred in each bug-fix commit, considering all bug-fix commits belonging to *Defects4J* projects. In total, we discover 155 repair actions from those bug-fix commits. Table VII shows the 10 most frequent AST-level repair actions. The column "Number of Patches" of Table VII shows, for each repair action, the number of analyzed patches that it appears.

For clustering evaluation, we use a metric called *Precision* (a.k.a.. Purity) [67]. It is an external evaluation criterion of cluster quality and represents the percent of the total number of objects (data points) that were classified "correctly", in the unit range [0..1]. In our context, this metric represents the number of bug-fix commits associated with the most frequent repair action in a cluster divided

DOI: 10.1002/smr

Table VII. The 10 most frequent AST-level repair actions occurring in 395 real bug fixes.

Repair Action	Number of Patches
INSERT Simple Name	349
INSERT Method Invocation	264
INSERT Infix Expression	228
INSERT Block	207
INSERT IF Statement	190
INSERT Expression Statement	125
INSERT Number Literal	122
INSERT Variable Declaration Fragment	111
INSERT Return Statement	98
DELETE Method Invocation	91

by the cluster size. The overall precision (Equation 2) of the clustering solution is obtained by taking a weighted sum of the individual cluster purities:

$$Precision = \frac{1}{N}\sum_{i=1}^{k}\max_{j}|C_i \cap T_j| \tag{2}$$

where N = number of objects (data points), k = number of clusters, C_i is a cluster in $\mathbb{C}$, and T_j is the classification which has the maximum count for cluster C_i. In general, the larger the values of purity, the better the clustering solution is.

When we say "correctly" that implies that each cluster C_i has identified a group of objects as the same class that the ground-truth $\mathbb{T}$ has indicated. We use the ground-truth classification T_i of those objects as the measure of assignment correctness, however to do so we must know which cluster C_i maps to which ground-truth classification T_i. If it were 100% accurate then each C_i would map to exactly one T_i, but in reality our C_i contains some points whose ground-truth classified them as several other classifications. Naturally then we can see that the highest clustering quality will be obtained by using the C_i to T_i mapping which has the most number of correct classifications i.e. $C_i \cap T_i$. That is where the *max* comes from in the Equation 2.

The *Recall* of cluster C_i [67] (Equation 3) is defined as:

$$Recall = \frac{n_i}{|T_j|} = \frac{n_i}{m_j} \tag{3}$$

where: n_i = number of elements within the cluster C_i, $m_j = |T_j|$ (i.e., number of elements within the ground-truth classification). It measures the fraction of point in ground-truth classification T_j shared in common with cluster C_i.

The *F-measure* (a.k.a. F1 score or F-score) is the harmonic mean of the precision and recall values for each cluster C_i [67]. This metric reaches its best value at 1 (perfect precision and recall) and worst at 0. The F-measure for cluster C_i (Equation 4) is therefore given below:

$$F - measure = \frac{2 \cdot precision_i \cdot recall_i}{precision_i + recall_i} \tag{4}$$

We have created a confusion matrix (a.k.a. matching matrix) [68] to calculate the values for **Precision**, **Recall**, and **F-Measure**. This can be done by looping through each cluster C_i and counting how many objects were classified as each class T_i. Table VIII shows the confusion matrix obtained.

Table IX shows the evaluation summary for those clusters. The column "Precision" of Table IX shows the individual purities obtained for each cluster. As shown in this table, the overall *Precision* and *Recall* values were 0.62 and 0.64, respectively. Ideally, each cluster would contain all instances of one repair action (perfect recall = 1) and only instances of one repair action (perfect precision = 1).

As shown in Table IX, we found a most prevalent repair action for each cluster. However, these clusters also contain more than one repair action.

DOI: 10.1002/smr

Table VIII. Confusion Matrix for Clustering Solution. T_1: Delete Method Invocation, T_2: Insert Infix Expression, T_3: Insert IF Statement, T_4: Insert Simple Name, T_5: Insert Return Statement, T_6: Insert Variable Declaration Fragment, T_7: Insert Method Invocation, T_8: Insert Number Literal, T_9: Insert Block, T_{10}: Insert Expression Statement, n_i: Number of elements within the cluster C_i, m_j: Number of elements within the ground-truth classification T_j.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	n_i
C_1	0	78	0	0	0	0	145	0	0	0	223
C_2	0	0	15	0	0	0	0	3	0	10	28
C_3	0	0	0	0	0	0	1	2	0	3	6
C_4	0	7	0	0	0	0	0	1	0	0	8
C_5	0	0	7	0	0	0	0	9	2	0	18
C_6	0	0	3	0	0	0	0	0	3	2	8
C_7	0	0	0	6	0	0	2	0	0	0	8
C_8	3	0	0	0	1	0	0	0	0	2	6
C_9	0	3	0	0	3	0	0	0	0	0	6
C_{10}	0	0	0	0	3	4	0	0	0	0	7
m_j	3	88	25	6	7	4	148	15	5	17	$n = 318$

Table IX. Evaluation Summary. Total = Number of instances within the cluster; Instances = Number of instances within the cluster associated with the repair action; Precision = Percentage of instances within the cluster associated with the most frequent repair action; Recall = Percentage of instances within the cluster associated with a repair action in relation to the total number of instances for this same repair action.

Cluster	Total	Instances	Precision	Recall	F-Measure	Repair Action
C_1	223	145	0.65	0.97	0.77	INSERT Method Invocation
C_2	28	15	0.53	0.60	0.56	INSERT IF Statement
C_3	6	3	0.50	0.17	0.25	INSERT Expression Statement
C_4	8	7	0.87	0.07	0.13	INSERT Infix Expression
C_5	18	9	0.50	0.60	0.54	INSERT Number Literal
C_6	8	3	0.375	0.60	0.46	INSERT Block
C_7	8	6	0.75	1.00	0.85	INSERT Simple Name
C_8	6	3	0.50	1.00	0.66	DELETE Method Invocation
C_9	6	3	0.50	0.43	0.46	INSERT Return Statement
C_{10}	7	4	0.57	1.00	0.72	INSERT Variable Declaration Fragment
Overall	318	198	0.62	0.64	0.54	

1. **INSERT Method Invocation**: The bug fix inserts a new (API) method call. The listing below shows an insertion of the API method call `Math.max` in JFreeChart-13 snippet:

```
- new Range(0.0, constraint.getWidth() - w[2])
+ new Range(0.0, Math.max(constraint.getWidth() - w[2], 0.0))
```

2. **INSERT IF Statement**: This bug fix adds an `IF` predicate to ensure a precondition is met before an object is accessed or an operation is performed. Without the precondition check, there may be a `NullPointerException` error caused by the buggy code or invalid operation execution [18]. The listing below shows this repair action occurring in Time-27 snippet:

```
+ if (sep.iAfterParser == null && sep.iAfterPrinter == null)
```

3. **INSERT Expression Statement**: This bug fix inserts an expression statement. The listing below shows this repair action occurring in Closure-133 snippet:

```
+ unreadToken = NO_UNREAD_TOKEN;
```

4. **INSERT Infix Expression**: This bug fix inserts an infix expression (e.g., comparison operator within an `IF` condition). The listing below shows this repair action occurring in Math-48 snippet:

```
+ if (x == x1) {
+    throw new ConvergenceException();
```


DOI: 10.1002/smr

```
+ }
```

5. **INSERT Number Literal**: This bug fix inserts a number literal (e.g., a number in an arithmetic expression). The listing below shows this repair action occurring in Time-9 snippet:

```
- int hoursInMinutes = FieldUtils.safeMultiply(hoursOffset, 60);
+ int hoursInMinutes = hoursOffset * 60;
```

6. **INSERT Block**: This bug fix inserts a list of statements (e.g., a case in Switch Statement). The listing below shows this repair action occurring in Closure-3 snippet:

```
+ case Token.NAME:
+ Var var = scope.getOwnSlot(input.getString());
+ if (var != null && var.getParentNode().isCatch()) {
+   return true;
+ }
```

7. **INSERT Simple Name**: This bug fix inserts a simple name (e.g., changing the order of a field in the constructor of a class). The listing below shows this repair action occurring in Time-4 snippet:

```
- Partial newPartial = new Partial(iChronology, newTypes,
    newValues);
 + Partial newPartial = new Partial(newTypes, newValues,
     iChronology);
```

8. **DELETE Method Invocation**: This bug fix deletes a method call. The listing below shows this repair action occurring in Mockito-38 snippet:

```
- return StringDescription.toString(m).equals(arg.toString());
+ return StringDescription.toString(m).equals(arg == null? "null"
    : arg.toString());
```

9. **INSERT Return Statement**: This bug fix inserts a return statement. The listing below shows this repair action occurring in Math-13 snippet:

```
+ if (m instanceof DiagonalMatrix) {
+   final int dim = m.getRowDimension();
+   final RealMatrix sqrtM = new DiagonalMatrix(dim);
+   for (int i = 0; i < dim; i++) {
+     sqrtM.setEntry(i,i,FastMath.sqrt(m.getEntry(i, i)));
+   }
+   return sqrtM;
```

10. **INSERT Variable Declaration**: This bug fix declares a new variable. The listing below shows this repair action occurring in Math-40 snippet:

```
- targetY = -REDUCTION_FACTOR * yB;
+ final int p = agingA - MAXIMAL_AGING;
+ final double weightA = (1 << p) - 1;
+ final double weightB = p + 1;
+ targetY = (weightA * yA - weightB * REDUCTION_FACTOR * yB) /
    (weightA + weightB);
```

Summary of RQ$_5$. We performed the **five** steps aforementioned in Subsection 4.3.3 to automatically discover pervasive bug fix patterns in Java. Thus, we discover a total of 155 repair actions from patches and discuss 10 pervasive repair actions that occur across all analyzed Java

DOI: 10.1002/smr

projects. Moreover, the overall *Precision* and *Recall* values for the clustering approach were 0.62 and 0.64, respectively. Due to the fact that the repair actions are fine-grained source code changes, they would be suitable for automatic program repair techniques. For instance, the most popular repair actions could be used to improve automated patch generation techniques.

5. DISCUSSION OF STUDIES

In this section, we discuss the lessons learned in the three studies.

5.1. Lessons learned about Studies I and II:

The findings of our paper provide useful insights for automatic program repair tools in Java. It suggests that patterns proposed by the state-of-the-art approaches for Java are insufficient to cover the extent of bug fixes in the analyzed data sets mentioned above in Section 2. Our findings suggest that test-suite based program repair may need to consider addition of method or logic/arithmetic expressions to achieve human-comparability in patches. Our results showed that developers often forget to add `IF` preconditions in the code. Evidence of this is that the bug-fix pattern that most appeared in the analyzed bug-fix commits of both data sets (i.e., *Boa* and *Defects4J*) was IF-APC (Addition of `IF` Precondition Check). To the best of our knowledge, few works have addressed this "defect class" [9] (i.e., NOPOL [3], SPR [34]). However, a recent work showed that NOPOL can automatically fix only 35 out of 224 bugs present in the *Defects4J* data set [16]. Moreover, patches by SPR only contain primitive values and do not contain object-oriented expressions (e.g., fields and method calls) [3].

Another interesting aspect concerns the importance of fixing bugs in multiple programming languages or in non-source files (the same results were obtained by Zhong and Su [30]). As automatic program repair has been evaluated on only a limited number of programming languages, such as C and Java, it may require significant improvement to fix bugs in other programming languages. Concerning bugs in non-source files (e.g., configuration files like XML or properties), future research in software fault localization needs to be performed. Current fault localization approaches can deal with 30% of source files at the most [30].

5.2. Lessons learned about Study III:

Although our approach does not always produce high-purity clusters (i.e, clusters that clearly refer to a repair action), we obtained a high degree of purity for cluster C1 (77% F-measure for "Insert Method Invocation" repair action). Moreover, we have shown that the bug-fix pattern that most appeared in the analyzed bug-fix commits of both data sets (i.e., Boa and Defects4J) was "Addition of IF Precondition Check (IF-APC)" (a.k.a. "Insert IF Statement"). Our results are in line with the results achieved by Yue and colleagues [69]. They found that 72% of manually inspected repeated-fix groups focused on bugs in **if-statements** or **if-conditions**, meaning that **if-statement** is one of the biggest software pitfall.

One interesting question is, of the frequent repair actions we identify, *which are currently handled by existing automatic repair tools?* To the best of our knowledge, very few. For instance, NOPOL [3] targets the "INSERT IF Statement" repair action. NOPOL addresses `IF` conditional bugs (i.e., `if-then-else` statements). It repairs programs by either modifying an existing `IF` condition or adding a precondition (a.k.a.. a guard) to any statement or block in the code. The modified or inserted condition is synthesized via input-output based code synthesis with SMT [70] and predicate switching [71]. For other repair actions like "INSERT Method Invocation" or "INSERT Return Statement", there is a lack of tool support. Therefore, our results unveiled the need to support fixes like adding method invocations to automated program repair tools. In this paper, we have only presented a few of the more common repair actions in Java. Many more exist, although building a generic tool to look for some of them might be prohibitively difficult because of their project-specific characteristics.

DOI: 10.1002/smr

We compared the most common repair actions that we found with the bug-fix patterns revealed by Pan *et al.* [18]. Table X shows the intersection or disjoint sets between those patterns. As shown in this table, our repair actions can be mapped to 8 out of 27 bug-fix patterns (29.6296%) studied by Pan *et al* and shown in Table 2 of their paper.

Table X. Mapping between the bug-fix patterns discovered in our study and those revealed by Pan *et al*.

Pervasive Repair Action	Corresponding Pan *et al.'s* bug-fix pattern
INSERT Method Invocation	Addition of a method declaration (MD-ADD)
INSERT IF Statement	Addition of precondition check (IF-APC), Addition of Precondition Check with Jump (IF-APCJ), Addition of Post-condition Check (IF-APTC)
INSERT Expression Statement	No corresponding bug-fix pattern
INSERT Infix Expression	No corresponding bug-fix pattern
INSERT Number Literal	No corresponding bug-fix pattern
INSERT Block	Addition of operations in an operation sequence of field settings (SQ-AFO), Addition of operations in an operation sequence of method calls to an object (SQ-AMO)
INSERT Simple Name	No corresponding bug-fix pattern
DELETE Method Invocation	Removal of a method declaration (MD-RMV)
INSERT Return Statement	No corresponding bug-fix pattern
INSERT Variable Declaration Fragment	Addition of a class field (CF-ADD)

6. THREATS TO VALIDITY

In this section we discuss the limitations of our studies. Concerning the **Study III**, a bug somewhere in the implementation may invalidate our results. Another threat is that we only inspect repairs which appear in a commit. Bugs that are repaired before a commit is merged into the main branch are not captured by our method. Moreover, there is risk that our data set of six open-source projects is not representative of Java software as a whole. Also, our results may not generalize to other programming languages. Concerning the **Studies I and II**, the following threats were identified:

1. **Tangled code changes in bug-fix commits**. Although the *Boa* data set is representative and contains a multitude of real-world Java bugs, the respective bug-fix commits may suffer from *tangled code changes* [28][29] (i.e., unrelated or loosely related code changes committed by developers in a single transaction). To mitigate this threat, we conducted a qualitative analysis in the real-world Java bugs present in the *Defects4J* data set [17]. One of the advantages of using this data set is that it contains isolated bugs (i.e., the bug fixes do not contain unrelated code changes such as addition of features or refactorings).
2. **Correctness of Boa programs.** The correctness of our automated analysis in the *Boa* data set depends on both our *Boa* programs and its Domain-Specific Language (DSL). For example, we rely on *Boa* to identify bug-fix commits. However, precisely accomplishing this is an open problem and some false positives (i.e., commits that are not related to bug fixes and have been erroneously classified as being) may have been included in our automated analysis. We have conducted a *manual analysis* on the real and isolated bug fixes present in the *Defects4J* data set to assess this problem. Moreover, we released our *Boa* programs to mitigate the risk of implementation errors. Because *Boa* does not provide an easy mechanism to identify precise, statement-level diffs between commits, our template matching and analysis of code changes (by counting each statement type or expression type) only provide estimates of behavior. We consider our results as informative approximations;
3. **Systems are all open-source.** All systems examined in this paper are developed as open-source. Furthermore, most GitHub repositories are personal (i.e., 71.6% of the repositories have only one committer: its owner) and have very low activity [72, 73]. Hence they might not be representative of closed-source development since different development processes

DOI: 10.1002/smr

could lead to different bug-fix patterns. Despite being open-source, several of the analyzed projects have substantial industrial participation;

4. **Bug-fix patterns have incomplete coverage of bug fixes.** Concerning the coverage of bug fixes by bug-fix patterns, only 55.1423% and 59.3495% of bug fixes (belonging to *Boa* and *Defects4J* data sets, respectively) contain at least one identifiable bug-fix pattern, and hence there are many bug fix changes that are not accounted for by one of the five bug-fix patterns mentioned above in Subsection 3.1.
5. **False positives in bug identification.** The built-in function `isfixingrevision` identifies bug-fix commits by using a list of regular expressions to match against the revision's log (i.e., commit's log message). There is a limitation in this approach: it only uses the change log information, and change logs of some non-bug-fix changes may also match these list of regular expressions. A more precise way for identifying bugs is to use bug tracking information together with change logs. Due *Boa's* design, it was not possible to use the bug tracking information. This paper only used change logs for identifying bug-fix commits, which may cause some false positives in bug identification.

7. RELATED WORK

Empirical Studies on Repeated Code Changes. Researchers have observed that developers apply repeated code changes [74, 59, 69, 75, 76, 77, 78]. Such a group of similar code changes [74], can be performed for several reasons: fixing multiple occurrences of the same bug, adapting code to a changed API, migrating to a different library/framework, refactoring, performing routine code maintenance tasks, etc. For instance, Pan *et al.* [18] automatically extracted 27 repair templates on Java software, and observed that `if`-condition changes are the most frequently applied bug fixes. They use line level differencing to extract and reason about repair patterns for automated program repair. The repair patterns they identify are coarse grained and do not identify the root cause of the bugs. Campos and Maia [79] designed *Boa* programs [19] that automatically detect the five most common repair templates identified in the work of Pan *et al.* [18]. They found that the repair template that most appeared in the analyzed bug-fix commits of both data sets (i.e., *Boa* [19] and *Defects4J* [17]) was IF-APC (Addition of `IF` Precondition Check). Nguyen *et al.* [76] found that 17-45% of bug fixes were recurring. They extracted related objects' API usage in modified code before and after each fix, and clustered bug fixes based on the graphical representation of API usage modification. In this paper, our focus is not just to demonstrate the existence of repeated fixes. Instead, compared with all prior studies, we provide an automatic approach to discover frequently occurring and isolated repair actions in Java. These patterns are expressed in a finer granularity than Pan *et al.*'s work [18] and are more suitable to be used in an automatic program repair technique because they are fine granularity source code changes [14].

Mining change types. Hanam *et al.* [80] proposed a novel technique called BugAID, for discovering the most prevalent and detectable repair templates in JavaScript language. In our work, an approach similar to BugAID [80] was adopted (i.e., based on unsupervised machine learning and AST differencing of bug fixes in the code using the GumTree tool). However, our approach was designed for Java language rather than JavaScript language. Because each approach was designed to find bug-fix patterns for a particular programming language, it does not make much sense to compare the editing scripts between them. These two languages are syntactically different. Moreover, our approach differs from theirs since it includes a preprocessing step based on Locality-Sensitive Hashing to remove outliers from search space before clustering the data. Long *et al.* [81] proposed Genesis, the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches. Genesis was designed for three classes of defects in Java programs: null pointer (NP), out of bounds (OOB), and class cast (CC) defects. By automatically inferring transforms from successful human patches, Genesis makes it possible to leverage the combined expertise and patch generation strategies of developers worldwide to automatically patch bugs in new applications. Fluri *et al.* [82] use hierarchical clustering to discover unknown change types in three Java applications. This approach is similar to ours, however, the

DOI: 10.1002/smr

basic change types they use are limited to 41 basic change types identified in [83]. In our work, we use density-based spatial clustering and considered 155 basic change types (i.e., repair actions). Livshits and Zimmermann [84] discover application-specific repair templates (methods that should be repaired but are not) by using association rule mining on two Java projects. Kim *et al.* [5] introduced PAR, an algorithm that generates program patches using a set of ten manually written fix templates. Martinez and Monperrus [14] analyzed the links between the nature of bug fixes and automatic program repair. They give extensive empirical results on the nature of human bug fixes at a large scale and a fine granularity with abstract syntax tree differencing using ChangeDistiller [82]. They mined repair models from manual fixes, and the mined repair models improve random search. Our work also investigates the nature of human bug fixes at a large scale and a fine granularity with abstract syntax tree differencing. However, we use the state-of-the-art AST diff tool GumTree [21].

Automatic program repair. The subfield of automatic program repair is concerned with automatically fixing bugs, without human intervention. Since 2009, interest in this subfield has grown substantially, and currently there are at least twenty projects involving some form of program repair (e.g., AE [85], AutoFix-E [86], ClearView [87], GenProg [43], Kali [8], NOPOL [3], PACHIKA [88], PAR [5], Prophet [33], SPR [34], RSRepair [35], Semfix [4], TrpAutoRepair [89], etc.).

NOPOL [3] targets a specific fault class: `IF` conditional bugs (i.e., `if-then-else` statements). It repairs programs by either modifying an existing `IF` condition or adding a precondition (a.k.a.. a guard) to any statement or block in the code. The modified or inserted condition is synthesized via input-output based code synthesis with SMT [70] and predicate switching [71]. The evaluation was done on 22 real-world bugs from two large open-source projects.

Empirical Knowledge on Automatic Program Repair. Zhong and Su [30] designed and developed *BugStat*, a tool that extracts and analyzes bug fixes. They conducted an empirical study on more than 9,000 real-world bug fixes from six popular Java projects. Their results provide useful guidance and insights for improving the state-of-the-art of automatic program repair. We study a much larger data set [19] with 101,471 Java projects. Moreover, we designed *Boa* programs that automatically detect the five most common bug-fix patterns identified in the work of Pan *et al.* [18].

Martinez and Monperrus [14] analyzed the links between the nature of bug fixes and automatic program repair. Below, we present the main similarities and differences between the two works:

- We both share the idea that one can mine repair actions from software repositories. In other words, one can learn from past bug fixes the main repair actions (e.g., adding a method call);
- We both share the idea that repair actions are meant to be generic enough to be independent of the kinds of bug and the software domains;
- We both extract repair actions automatically based on AST differencing;
- Martinez and Monperrus [14] mined repair models (i.e., a repair model consists of a set of repair actions) from past bug fixes to guide the repair process. In contrast, we propose a novel automatic technique for unveiling frequent and isolated repair actions corresponding to realistic bug fixes in Java;
- From the external validity viewpoint, there is risk that the dataset of 14 Java projects used by Martinez and Monperrus [14] is not representative of Java software as a whole. Unlike their work, our work does not suffer from this problem because it was conducted on a large scale with 101,471 Java projects;
- There may be some bug fix changes that do not recur very often across projects. Since these bug fix changes are not repetitive, they do not constitute bug fix patterns and should be eliminated from the search space. For this, a pre-processing step was performed using Locality-Sensitive Hashing to eliminate these non-repetitive bug fixes before clustering them and revealing the bug fix patterns. On the contrary, Martinez and Monperrus [14] did not take this threat into account and did not perform any pre-processing step on bug fix changes.

Soto *et al.* [90] conducted a large-scale study of bug-fix commits in Java projects. Their findings provide useful insights for automatic program repair tools in Java. They created *Boa* programs to detect the PAR's bug-fix patterns [5] and provided an informative approximation of their prevalence

DOI: 10.1002/smr

in the *Boa* data set. We used the same data set in our study but we created *Boa* programs to detect the five most common bug-fix patterns identified in the work of Pan *et al.* [18]. Moreover, we do not limit our study to bug-fix patterns. We also investigated other aspects related to human-made bug fixes such as the statement types that appear more frequently in bug-fix commits and the file types that are usually changed to fix a bug.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented the idea that one can mine repair actions from bug-fix commits. In other words, one can learn from past bug fixes the main repair actions (e.g., adding a method call). Those repair actions are independent of the kinds of bug and the software application domains. We considered 395 real bugs from six open-source Java projects present in *Defects4J* data set. We discovered 155 AST-level repair actions and 10 frequent and isolated cross-project repair actions in Java language.

The repair actions extracted from existing human-written patches could be leveraged to improve automated patch generation techniques. Our findings are useful for improving tools and techniques to prevent common bugs in Java, making developers aware of common mistakes involved with this programming language.

Moreover, this paper explored the underlying patterns in bug fixes mined from software project change histories. We rely on *Boa* to automatically identify bug-fix commits and to detect the five most common bug-fix patterns identified by Pan *et al.* [18]. The findings of our study provide useful insights for automatic repair tools in Java.

Our future work will concentrate on the following topics:

Automatic repair systems. An example of follow-up work would be to propose an approach to automatic repair `IF` null check preconditions (i.e., IF-APC bug-fix pattern). There are a number of program repair techniques (e.g., [43]) but not one of them is dedicated to fix null pointer exceptions.

Additional bug-fix patterns. We could conduct a more sophisticated analysis to discover additional bug-fix patterns, thereby increasing coverage of bug fixes by bug-fix patterns and potentially altering the observed pattern frequencies.

Bug localization techniques. We can explore how to locate bugs in non-source files (e.g., configuration files) or source files of different programming languages present in a Java project and how to fix them with advanced techniques.

9. ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] Le Goues C, Dewey-Vogt M, Forrest S, Weimer W. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. *Proc. of ICSE '12*, IEEE Press, 2012; 3–13.

[2] Kim S, Whitehead EJ Jr. How Long Did It Take to Fix Bugs? *Proc. MSR '06*, ACM, 2006; 173–174.

[3] Xuan J, Martinez M, DeMarco F, Clément M, Lamelas S, Durieux T, Le Berre D, Monperrus M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 2016; doi:10.1109/TSE.2016.2560811.

[4] Nguyen HDT, Qi D, Roychoudhury A, Chandra S. SemFix: Program Repair via Semantic Analysis. *Proc. ICSE '13*, IEEE Press; 772–781.

[5] Kim D, Nam J, Song J, Kim S. Automatic Patch Generation Learned from Human-written Patches. *Proc. of the ICSE '13*, IEEE Press: Piscataway, NJ, USA, 2013; 802–811.

[6] Goues CL, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 2015; **41**(12):1236–1256.

[7] Britton T, Jeng L, Carver G, Cheak P, Katzenellenbogen T. Reversible debugging software. *Technical Report*, University of Cambridge 2013.

[8] Qi Z, Long F, Achour S, Rinard M. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. *Proc. of the International Symposium on Software Testing and Analysis (ISSTA' 15)*, ACM, 2015; 24–36.

[9] Monperrus M. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. *Proc. ICSE' 2014*, ACM, 2014; 234–242.

[10] Soto M, Goues CL. Using a probabilistic model to predict bug fixes. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 00, 2018; 221–231.

[11] Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering* Aug 2016; **42**(8):707–740, doi:10.1109/TSE.2016.2521368.

[12] Fry ZP, Weimer W. A Human Study of Fault Localization Accuracy. *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, IEEE Computer Society: Washington, DC, USA, 2010; 1–10.

[13] Soto M, Le Goues C. Common statement kind changes to inform automatic program repair. *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Zaidman A, Kamei Y, Hill E (eds.), ACM, 2018; 102–105.

[14] Martinez M, Monperrus M. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering* Feb 2015; **20**(1):176–205.

[15] Smith EK, Barr ET, Le Goues C, Brun Y. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. *Proc. 10th ESEC/FSE*, ACM, 2015; 532–543.

[16] Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 2016; :1–29.

[17] Just R, Jalali D, Ernst MD. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. *Proc. ISSTA' 2014*, ACM, 2014; 437–440.

[18] Pan K, Kim S, Whitehead EJ Jr. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.* 2009; **14**(3):286–315.

[19] Dyer R, Nguyen HA, Rajan H, Nguyen TN. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.* Dec 2015; **25**(1):7:1–7:34.

[20] Campos EC, d A Maia M. Common Bug-Fix Patterns: A Large-Scale Observational Study. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017; 404–413.

DOI: 10.1002/smr

[21] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and Accurate Source Code Differencing. *Proc. 29th ASE*, ACM, 2014; 313–324.

[22] Dyer R, Rajan H, Nguyen HA, Nguyen TN. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM: New York, NY, USA, 2014; 779–790.

[23] Gosling J, Joy B, Steele GL. *Java(TM) Language Specification*. 1st edition edn., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1996.

[24] Gosling J, Joy B, Steele GL, Bracha G. *Java(TM) Language Specification*. 2nd edition edn., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2000.

[25] Gosling J, Joy B, Steele G, Bracha G. *Java(TM) Language Specification*. 3rd edition edn., Addison-Wesley Professional, 2005.

[26] Gosling J, Joy B, Steele G, Bracha G, Buckley A. *Java(TM) Language Specification*. Java SE 7 edition edn., Prentice Hall, 2013.

[27] Kong X, Zhang L, Wong WE, Li B. Experience report: How do techniques, programs, and tests impact automated program repair? *Proc. 26th ISSRE' 2015*, 2015; 194–204.

[28] Herzig K, Zeller A. The Impact of Tangled Code Changes. *Proc. 10th MSR '13*, IEEE Press, 2013; 121–130.

[29] Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S. Untangling fine-grained code changes. *Proc. SANER' 15*; 341–350.

[30] Zhong H, Su Z. An Empirical Study on Real Bug Fixes. *Proc. of ICSE '15*, IEEE Press; 913–923.

[31] Le Goues C, Wemer W, Forrest S. Representations and operators for improving evolutionary software repair. *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, 2012; 959–966, doi:10.1145/2330163.2330296.

[32] Le XBD, Lo D, Goues CL. History Driven Program Repair. *Proc. of 23rd SANER*, vol. 1, 2016; 213–224.

[33] Long F, Rinard M. Automatic Patch Generation by Learning Correct Code. *POPL '16*, ACM, 2016; 298–312.

[34] Long F, Rinard M. Staged Program Repair with Condition Synthesis. *Proc. ESEC/FSE*, ACM, 2015; 166–178.

[35] Qi Y, Mao X, Lei Y, Dai Z, Wang C. The Strength of Random Search on Automated Program Repair. *Proc. ICSE' 2014*, ACM; 254–265.

[36] Samimi H, Schäfer M, Artzi S, Millstein T, Tip F, Hendren L. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. *Proc. of ICSE '12*, IEEE Press: Piscataway, NJ, USA, 2012; 277–287.

[37] Weimer W, Fry ZP, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. *Proc. of 28th ASE*, 2013; 356–366.

[38] Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, IEEE Computer Society: Washington, DC, USA, 2009; 364–374, doi:10.1109/ICSE.2009.5070536.

[39] Martinez M, Weimer W, Monperrus M. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, ACM: New York, NY, USA, 2014; 492–495.

[40] Barr ET, Brun Y, Devanbu P, Harman M, Sarro F. The Plastic Surgery Hypothesis. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, ACM: New York, NY, USA, 2014; 306–317.

[41] Fluri B, Wuersch M, PInzger M, Gall H. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* Nov 2007; **33**(11):725–743, doi:10.1109/TSE.2007.70731.

[42] Tichy W. Automated Bug Fixing: An Interview with Westley Weimer and Martin Monperrus. *Ubiquity* 2015; :1:1–1:11.

[43] Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 2012; **38**(1):54–72.

[44] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. *Proc. of the International Conference on Software Engineering (ICSE '16)*, ACM, 2016; 691–701.

[45] Purushothaman R, Perry DE. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* June 2005; **31**(6):511–526, doi:10.1109/TSE.2005.74.

[46] Andoni A, Indyk P. Near-optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Commun. ACM* Jan 2008; **51**(1):117–122.

[47] Leskovec J, Rajaraman A, Ullman J. Finding similar items. *Mining of Massive Datasets*. chap. 3, Cambridge University Press: New York, NY, USA, 2014; 73–130.

[48] Brin S, Page L. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems* Apr 1998; **30**(1-7):107–117.

[49] Jing Y, Baluja S. VisualRank: Applying PageRank to Large-Scale Image Search. *IEEE Trans. Pattern Anal. Mach. Intell.* Nov 2008; **30**(11):1877–1890.

[50] Gionis A, Indyk P, Motwani R. Similarity Search in High Dimensions via Hashing. *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1999; 518–529.

[51] Petrović S, Osborne M, Lavrenko V. Streaming First Story Detection with Application to Twitter. *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, Association for Computational Linguistics: Stroudsburg, PA, USA, 2010; 181–189.

[52] Ioffe S. Improved Consistent Sampling, Weighted Minhash and L1 Sketching. *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, IEEE Computer Society: Washington, DC, USA, 2010; 246–255.

[53] Broder AZ, Charikar M, Frieze AM, Mitzenmacher M. Min-wise Independent Permutations (Extended Abstract). *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, ACM: New York, NY, USA, 1998; 327–336.

[54] Indyk P, Motwani R. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, ACM: New York, NY, USA, 1998; 604–613.

[55] Fowler G, Noll LC, Vo P. Fnv hash. *Technical Report* 1991. Http://www.isthe.com/chongo/tech/comp/fnv/.

[56] Grefenstette G. *Tokenization*. Springer Netherlands: Dordrecht, 1999; 117–133.

[57] Meng N, Kim M, McKinley KS. LASE: Locating and Applying Systematic Edits by Learning from Examples. *Proc. ICSE '13*, IEEE Press, 2013; 502–511.

[58] Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P. Fair and Balanced?: Bias in Bug-fix Datasets. *Proc. 7th ESEC/FSE*, ACM, 2009; 121–130.

[59] Molderez T, Stevens R, De Roover C. Mining Change Histories for Unknown Systematic Edits. *Proc. of MSR '17*, IEEE Press: Piscataway, NJ, USA, 2017; 248–256.

[60] Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* Nov 2009; **11**(1):10–18.

[61] Ester M, Kriegel HP, Sander J, Xu X. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proc. KDD'96*, AAAI Press, 1996; 226–231.

[62] Dash M, Choi K, Scheuermann P, Liu H. Feature Selection for Clustering - A Filter Solution. *Proc. ICDM '02*, IEEE Computer Society: Washington, DC, USA, 2002; 115–.

[63] Smiti A, Elouedi Z. DBSCAN-GM: An improved clustering method based on Gaussian Means and DBSCAN techniques. *2012 IEEE 16th International Conference on Intelligent Engineering Systems (INES)*, 2012; 573–578.

[64] Pampara G, Engelbrecht AP, Franken N. Binary Differential Evolution. *2006 IEEE International Conference on Evolutionary Computation*, 2006; 1873–1879.

[65] Karami A, Johansson R. Choosing DBSCAN Parameters Automatically using Differential Evolution. *International Journal of Computer Applications* April 2014; **91**(7):1–11.

[66] Zaki MJ, Jr WM. *Clustering Validation (Chapter 17)*. Cambridge University Press: New York, NY, USA, 2014.

[67] Conrad JG, Al-Kofahi K, Zhao Y, Karypis G. Effective Document Clustering for Large Heterogeneous Law Firm Collections. *Proceedings of the 10th International Conference on Artificial Intelligence and Law*, ICAIL '05, ACM: New York, NY, USA, 2005; 177–187.

[68] Stehman SV. Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment* 1997; **62**(1):77 – 89.

[69] Yue R, Meng N, Wang Q. A Characterization Study of Repeated Bug Fixes. *Proc. of ICSME' 17*, 2017; 422–432.

[70] Jha S, Gulwani S, Seshia SA, Tiwari A. Oracle-guided Component-based Program Synthesis. *Proc. ICSE '10*, ACM, 2010; 215–224.

[71] Zhang X, Gupta N, Gupta R. Locating Faults Through Automated Predicate Switching. *Proc. ICSE '06*, ACM, 2006; 272–281.

[72] Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. The Promises and Perils of Mining GitHub. *Proc. MSR' 2014*, ACM, 2014; 92–101.

[73] Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D. An In-depth Study of the Promises and Perils of Mining GitHub. *Empirical Softw. Engg.* Oct 2016; **21**(5):2035–2071.

DOI: 10.1002/smr

[74] Kim M, Notkin D. Discovering and Representing Systematic Code Changes. *Proc. of 31st ICSE*, IEEE Computer Society: Washington, DC, USA, 2009; 309–319.

[75] Kim S, Pan K, Whitehead EEJ Jr. Memories of Bug Fixes. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, ACM: New York, NY, USA, 2006; 35–45.

[76] Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi J, Nguyen TN. Recurring Bug Fixes in Object-oriented Programs. *Proc. of ICSE '10*, ACM: New York, NY, USA, 2010; 315–324.

[77] Park J, Kim M, Ray B, Bae DH. An empirical study of supplementary bug fixes. *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012; 40–49.

[78] Ray B, Kim M. A Case Study of Cross-system Porting in Forked Projects. *Proc. of FSE '12*, ACM: New York, NY, USA, 2012; 53:1–53:11.

[79] Campos EC, d A Maia M. Common Bug-Fix Patterns: A Large-Scale Observational Study. *Proc. of ESEM' 17*, ACM, 2017; 404–413.

[80] Hanam Q, Brito FSdM, Mesbah A. Discovering Bug Patterns in JavaScript. *Proc. 24th FSE*, ACM, 2016; 144–156.

[81] Long F, Amidon P, Rinard M. Automatic Inference of Code Transforms for Patch Generation. *Proc. of ESEC/FSE' 17*, ACM: New York, NY, USA, 2017; 727–739.

[82] Fluri B, Giger E, Gall HC. Discovering Patterns of Change Types. *Proc. ASE*, IEEE Computer Society, 2008; 463–466.

[83] Fluri B, Gall HC. Classifying Change Types for Qualifying Change Couplings. *Proc. 14th ICPC*, 2006; 35–45.

[84] Livshits B, Zimmermann T. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. *Proceedings of the ESEC/FSE-13*, ACM, 2005; 296–305.

[85] Weimer W, Fry ZP, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. *Proc. 28th ASE*, 2013; 356–366.

[86] Wei Y, Pei Y, Furia CA, Silva LS, Buchholz S, Meyer B, Zeller A. Automated Fixing of Programs with Contracts. *Proc. 19th ISSTA*, ACM, 2010; 61–72.

[87] Perkins JH, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G, *et al.*. Automatically Patching Errors in Deployed Software. *Proc. SOSP '09*; 87–102.

[88] Dallmeier V, Zeller A, Meyer B. Generating Fixes from Object Behavior Anomalies. *Proc. ASE '09*; 550–554.

[89] Qi Y, Mao X, Lei Y. Efficient Automated Program Repair Through Fault-Recorded Testing Prioritization. *Proc. ICSM '13*, 2013; 180–189.

[90] Soto M, Thung F, Wong CP, Le Goues C, Lo D. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. *Proc. 13th MSR' 2016*, ACM, 2016; 512–515.

Appendices

Table XI shows code examples for each statement type (investigated in $\mathbf{RQ_2}$) that *Boa* language can identify.

DOI: 10.1002/smr

Statement Type	Code Example
ASSERT	

	`private Value getConstantNumber() {`
	`  assert (isToken(Token.NUMERIC));`
	`  return _scanner.getNumber();`
	`}`
	
	`private boolean isToken(Token token) {`
	`  return peek() == token;`
	`}`

BLOCK	

	`Boolean isOtherAttachmentsExists = false;`
	`AttachedFileDataType attachedFileDataType;`
	`ProjectNarrativeAttachments projectNarrativeAttachments =`
	`  ProjectNarrativeAttachments.Factory.newInstance();`
	`AbstractAttachments abstractAttachments =`
	`  AbstractAttachments.Factory.newInstance();`

BREAK	

	`private boolean checkBishopMoves(boolean isWhite) {`
	`  byte i = 1;`
	`  for (i = 1; (posY + i < BOARD_SIZE) && (posX + i <`
	`  BOARD_SIZE); i++){`
	`  attack = board[posY+i][posX+i];`
	`  if (attack != '.') {`
	`  if (isEnemyKing(attack, isWhite)) return true;`
	`  break;`
	`  }`
	`  }`
	`  ...`
	`}`

CATCH	

	`try {`
	`  performDirectCodeGen(inputArgs1);`
	`  Assert.fail("Expected an Exception of type: " +`
	`  CodeGenFailedException.class.getName());`
	`} catch (CodeGenFailedException e) {`
	`  String expected1 = "Properties file not found in the`
	`  location";`
	`  Assert.assertThat(e.getMessage(),`
	`  containsString(expected1));`
	`}`

DOI: 10.1002/smr

CONTINUE

```
for (String line: lines) {
  Matcher keyMatch = keyPat.matcher(line);
  if (keyMatch.matches()) {
     foundInfo(f);
     declaredKeys.add(keyMatch.group(1));
     continue;
   }
  ...
}
```

EXPRESSION

```
TextMessagePerf textMessage = new TextMessagePerf();
textMessage.msgId = meta.getId();
textMessage.fromuid = meta.getFrom();
textMessage.time = (long) meta.getTime() * 1000;
```

FOR

```
Collection<Alarm> list = alamsControler.getAlarmsList();
log.info("Alarm collection size is " + list.size());
for (Alarm elem : list) {
  log.info(elem.toString());
}
alamsControler.flush();
```

IF

```
if (budgetLASalaryTempKey.equals(budgetLASalaryKey)) {
  if (startDate.after(tempReportTypeVO.getStartDate())) {
    startDate = tempReportTypeVO.getStartDate();
  }
}
```

RETURN

```
final String zuctw = qncName.trim().toUpperCase();
if (zuctw.length() == 0) return null;
try {
  return Charset.forName(zuctw);
} catch (final UnsupportedCharsetException exUC) {
  return null;
}
```


DOI: 10.1002/smr

SYNCHRONIZED

```
public void commandReceived(Object message) {
   private final Object mutex = new Object();
   log.info("Message received by session");
   synchronized (mutex) {
     response = message;
     mutex.notify();
   }
}
```

THROW

```
public void setRegion(int left, int top, int width, int
    height) {
   if (top < 0 || left < 0) {
    throw new IllegalArgumentException("Left and top
        must be nonnegative");
   }
   if (height < 1 || width < 1) {
    throw new IllegalArgumentException("Height and
        width must be at least 1");
   }
}
```

TRY

```
Map<String, Object> args = new HashMap<String, Object>();
APICall task = new APICall();
ArrayList<String> responses = null;
try {
  responses = task.execute(args).get();
} catch (InterruptedException e) {
  e.printStackTrace();
} catch (ExecutionException e) {
  e.printStackTrace();
}
```

SWITCH

```
public void handleMessage(Message msg) {
  switch (msg.what) {
  case 1: rolateHandler.sendEmptyMessage(2);
          break;
  case 2:
        if (count < getDegree()) {
         rolateHandler.sendEmptyMessage(2);
       } else {
         isFinish = true;
       }
       break;
  case 3: BeginRolate(matrix, (XbigY ? count : 0));
          break;
  }
}
```


DOI: 10.1002/smr

WHILE	`while (rowsLeft > 0) { nRows = Math.min(32767 / (this.width * (this.bytesPerPixel + 1)), rowsLeft); int[] pixels = new int[this.width * nRows]; pg = new PixelGrabber(this.image, 0, startRow, this.width, nRows, pixels, 0, this.width); pg.grabPixels(); }`

Table XI. Code Examples that *Boa* language can identify for each Statement Type.

DOI: 10.1002/smr