2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement

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

Eduardo C. Campos
Faculty of Computing
Federal University of Uberlândia, Brazil
eccampos@ufu.br

Marcelo A. Maia
Faculty of Computing
Federal University of Uberlândia, Brazil
marcelo.maia@ufu.br

***Abstract*—*[Background]:* There are more bugs in real-world programs than human programmers can realistically address. Several approaches have been proposed to aid debugging. A recent research direction that has been increasingly gaining interest to address the reduction of costs associated with defect repair is automatic program repair. Recent work has shown that some kind of bugs are more suitable for automatic repair techniques. *[Aim]:* The detection and characterization of common bug-fix patterns in software repositories play an important role in advancing the field of automatic program repair. In this paper, we aim to characterize the occurrence of known bug-fix patterns in Java repositories at an unprecedented large scale. *[Method]:* The study was conducted for Java GitHub projects organized in two distinct data sets: the first one (i.e., Boa data set) contains more than 4 million bug-fix commits from 101,471 projects and the second one (i.e., Defects4J data set) contains 369 real bug fixes from five open-source projects. We used a domain-specific programming language called Boa in the first data set and conducted a manual analysis on the second data set in order to confront the results. *[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. Moreover, 76% of bug-fix commits associated with the IF-APC bug-fix pattern are isolated from the other four bug-fix patterns analyzed. *[Conclusion]:* Targeting on bugs that miss preconditions is a feasible alternative in automatic repair techniques that would produce a relevant payback.**

I. Introduction

There are more bugs[1] in real-world programs than human programmers can realistically address [17]. The battle against software bugs exists since software existed. Substantial effort is spent to fix bugs. For instance, Kim and Whitehead [15] 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" [17]. 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 [40]. Recent test-suite based repair approaches include the work by Le Goues *et al.* [17], Nguyen *et al.* [27], Kim *et al.* [14].

[1]In this paper, we use the terms "defect", "error", "fault", and "bug" as synonyms.

The cost of debugging and maintaining software has been continuously increasing [7]. A 2013 study estimated the global cost of debugging at $312 billion, with software developers spending half their time debugging [2]. 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 [17]. Currently, this recent research direction attracts much academic and industrial attention. Nonetheless, many people question the positive results. For instance, Qi *et al.* [34] 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 [17], PAR [14]) seem to be able to fix only simple bugs, due to several limitations [26].

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) [35].

While automatic program repair shows great promise, it is far from being widely adopted, and still many potential improvements remain to be made [7]. In a recent work, Martinez *et al.* [23] assessed the effectiveness of different automatic repair approaches on the real-world Java bugs of *Defects4J* [10]. They conducted a study with three automatic repair systems on 224 bugs present in this data set: jGenProg, an implementation of GenProg [17] for Java; jKali, an implementation of Kali [34] for Java; and NOPOL [40]. Their results showed that these repair systems together can synthesize a patch for only 21% of these bugs.

Generally speaking on debugging, the research community still does not have general consensus on which kinds of software bugs are most common [29]. The reason is not a deficit of research, but a lack of uniformity. Since 1975, efforts have been made to create taxonomies of common types of bug (e.g., [6], [1], [19], [20], [28], etc.). However, researchers often create new taxonomies of bug types instead of using an existing one. The end result (and current state of knowledge) is predictable: *the literature contains several bug classification taxonomies* [29].

978-1-5090-4039-1/17 $31.00 © 2017 IEEE
DOI 10.1109/ESEM.2017.55

IEEE computer society

The research community also has limited knowledge on the nature of bug fixes [26]. To the best of our knowledge, only two works analyzed the links between the nature of bug fixes and automatic program repair [24][42]. Furthermore, a large scale characterization of bug-fix patterns occurring in software repositories is still lacking. Thus, an empirical study to better understand the challenges for this research field is justified.

In this paper, we study a much larger data set [5] than those two previous works with 101,471 Java projects and more than 4 million 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* [8][4] (i.e., unrelated or loosely related code changes committed by developers in a single transaction). In order to make our results more dependable, we also conducted a qualitative analysis in the real-world Java bugs present in the *Defects4J* data set [10]. 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).

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) Decrease the bias 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 [5]. We used a domain-specific programming language called *Boa* [5] to analyze ultra-large-scale data efficiently.

The paper makes four contributions:

1) 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);
2) 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 (e.g., configuration files). Our results confirm the findings obtained by Zhong and Su [42] (please see the Findings 1, 2, 13, and 14 for more details). This is relevant because current automatic repair approaches have been evaluated on only source files belonging to a limited number of programming languages (such as C and Java);
3) We identified that the statement kinds `EXPRESSION`, `BLOCK`, `IF` and `RETURN` are frequently added to fix bugs. 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. To the best of our knowledge, few works have proposed inserting code at the block level [13] or considering multi-location repairs [25];
4) We conducted a manual analysis involving 369 real bug fixes from five open-source projects present in the *Defects4J* data set. This analysis showed that the majority of these bugs (i.e., 45%) are If-related (`IF`). In a recent work, Martinez *et al.* [24] have shown that `IF` conditions are among the most error-prone program elements in Java programs.

The remainder of this paper is organized as follows. Section II presents the data sets we use to perform the studies. Section III presents the research questions and the bug-fix patterns considered in the studies. Section IV details the studies and provides the results that are discussed in Section V. Related work is surveyed and shown in Section VI. Section VII concludes this paper and proposes future work.

II. Data sets and Characteristics

A. Boa data set

In this paper, we use the *September 2015/GitHub* data set offered by *Boa* [5], including 554,864 Java projects with 23,226,512 commits. *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 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 it is not common to see open-source Java projects hosted on GitHub with a large number of bug-fix commits (e.g., 51 or more bug-fix commits).

Programming Language: 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 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.

Kinds of changed files: Table I shows the kinds and descriptions of changed files present in the *Boa* data set and

```
counts: output sum of int;
p: Project = input;

exists (i: int; match(`^java$`, lowercase(p.programming_languages[i])))
foreach (j: int; p.code_repositories[j].kind == RepositoryKind.GIT)
  foreach (k: int; isfixingrevision(p.code_repositories[j].revisions[k].log))
    counts << 1;
```

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

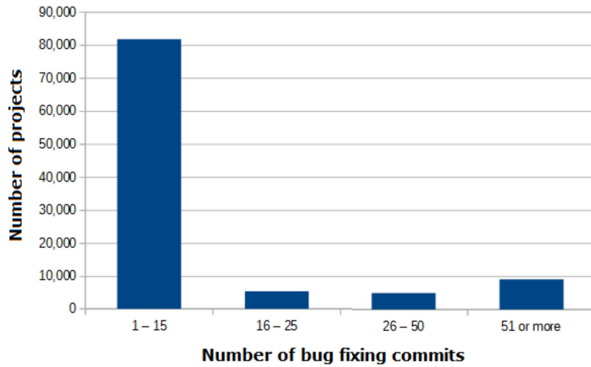

Fig. 2. Distribution of bug-fix commits among Java GitHub projects.

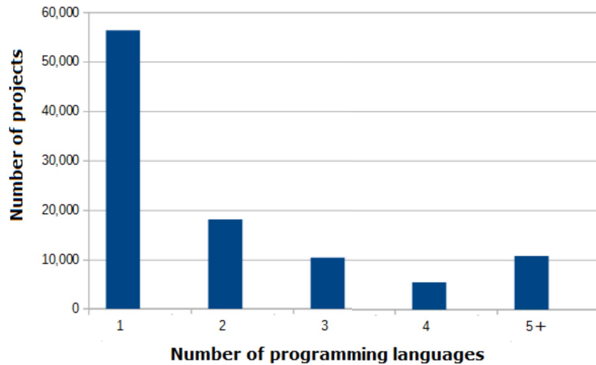

Fig. 3. Number of programming languages in each Java project using GIT.

the number of changed files per file kind. We consider a file *changed* if it is new, modified, or deleted in a commit. In total, 52,052,571 files were changed.

B. *Defects4J data set*

In this paper, we also use *Defects4J* [10], 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 [2], Closure Compiler [3], Commons Lang [4], Commons Math [5], Mockito [6] and Joda-Time [7]. 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 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.* [27]; Kong *et al.* [16]);
- **Scale**: It contains bugs that reside in large software projects (as opposed to bugs in student programs as in Smith *et al.* [35]);
- **Isolated Bugs**: A fundamental challenge when collecting bugs is deciding what constitutes a bug, and what does not [10]. When interacting with version control systems, developers frequently *group separate changes into a single commit*. Herzig *et al.* [8] studied this problem and named it as *tangled code changes* [4]. 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).

[2]JFreeChart, http://jfree.org/jfreechart/
[3]Closure Compiler, http://code.google.com/closure/compiler/
[4]Apache Commons Lang, http://commons.apache.org/lang
[5]Apache Commons Math, http://commons.apache.org/math
[6]Mockito, http://site.mockito.org/
[7]Joda-Time, http://joda.org/joda-time/

TABLE I
KINDS OF CHANGED FILES PRESENT IN THE *Boa* DATA SET (JLS: JAVA LANGUAGE SPECIFICATION).

File Kind	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

III. 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).

A. Bug-fix patterns

Pan *et al.* [29] 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, the most common individual 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 [29]. Example:

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

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 [29]. 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 [29]. 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 [29]. 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 [29]. Example:

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

B. Research Questions

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

> **RQ$_1$**: *Which file types are usually changed to fix a bug?*

The **first part of 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.

> **RQ$_2$**: *Which kinds of statements are often added or deleted by developers to fix bugs?*

The **second part of 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.

> **RQ$_3$**: *What is the prevalence of the 5 most common bug-fix patterns identified in the work of Pan et al. [29] in the bug fixes present in the Boa data set?*

The **third part of 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 [14].

> $\mathbf{RQ_4}$: *What is the prevalence of the 5 most common bug-fix patterns identified in the work of Pan et al. [29] in the bug fixes present in the Defects4J data set?*

The research question $\mathbf{RQ_4}$ is similar to $\mathbf{RQ_3}$. The basic difference between them is: the answer for $\mathbf{RQ_4}$ is based on manual inspection of real bug fixes from projects present in the *Defects4J* data set, while the answer for $\mathbf{RQ_3}$ 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., **third part of Study I**).

IV. 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 last research question (i.e., $\mathbf{RQ_4}$).

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

We study bug-fix commits in 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.* [29] 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 II-A.

We divide this first study into three parts (i.e., *Part I*, *Part II*, and *Part III*), so that each part of the study addresses a different research question (i.e., $\mathbf{RQ_1}$, $\mathbf{RQ_2}$, and $\mathbf{RQ_3}$, respectively).

1) Part I: Figure 4 shows the number of bug-fix commits per File Kind. As shown in Figure 4, the 2 kinds 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 kinds 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" [42]); 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 [42]. 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

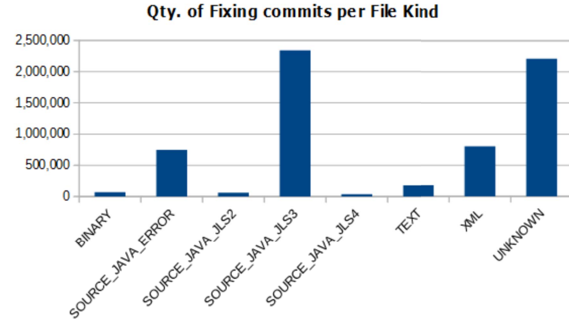

Fig. 4. Number of bug-fix commits per File Kind.

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.

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 [42] (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 [22], GenProg [17]) or Java language (e.g., NOPOL [40], PAR [14]). 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.

2) Part II: In order to find out which kinds of statements appear more frequently in bug-fix commits, we use *Boa* to compute the number of bug-fix commits that added/deleted a particular statement kind in order to solve the corresponding bug. We investigate the following 14 statement kinds present in the *Boa* Programming Guide [8]: `ASSERT`, `BLOCK`, `BREAK`, `CATCH`, `CONTINUE`, `EXPRESSION`, `FOR`, `IF`, `RETURN`, `SYNCHRONIZED`, `THROW`, `TRY`, `SWITCH`, and `WHILE`. The statement kind `BLOCK` is somewhat different because it was designed by *Boa* inventors to characterize a statement that contains a list with two or more statements within it (e.g., the statements in the method body). Concerning the statement kind `EXPRESSION`, it encompasses arithmetic or logical expres-

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

sions, expressions with binary operators, etc. For a complete list, please see the Section `ExpressionKind` present in the *Boa* Programming Guide [9].

For example, 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 kinds of statements aforementioned, we performed a similar algorithm, but considering the number of times each statement kind 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 kinds that are added or deleted to fix a particular bug. For instance, we can identify the prevalence of some statement kinds 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 kind is 6. This result clearly shows the limitations of current automatic repair techniques, since many bug fixes involve adding code blocks. However, the majority of fixes produced by GenProg [17] are "one-liners" (i.e., changes only one line of code) [37]. As pointed out by Monperrus [26], many existing automatic program repair approaches are effective only in fixing bugs that require simple changes.

3) Part III: Pan *et al.* [29] 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 [29]. 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 II.

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 [5]. *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 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 III.

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 kind of expressions we consider, see the Section `ExpressionKind` of this page [10]. *We found that 196,283 out of 4,590,405 (4.2759%) bug-fix commits change one or more* `IF` *condition expressions (Execution time: 27m 16s).*
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) 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 (Execution time: 1h 15m 2s).*
3) **How many bug-fix commits change the number or type of parameters 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 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 (Execution time: 39m 33s).*

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

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

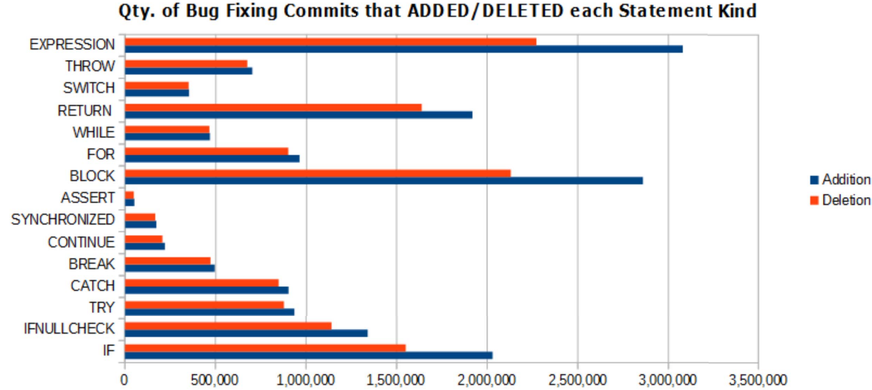

Fig. 5. Number of Bug-Fix Commits that ADDED/DELETED each Statement Kind.

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 kind of expressions we consider, see the Section `ExpressionKind` of this page [11]. *We found that 511,299 out of 4,590,405 (11.1384%) bug-fix commits change one or more assignment expressions (Execution time: 30m 26s).*
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, 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 (Execution time: 22m 26s).*

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.

B. 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

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

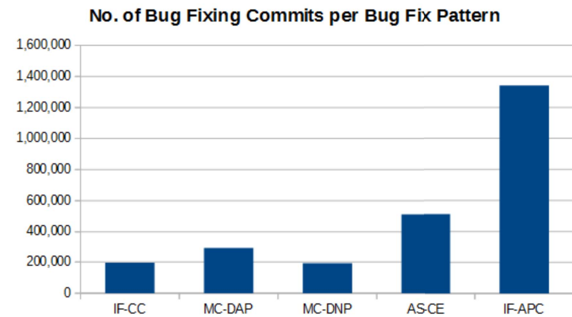

Fig. 6. Number of bug-fix commits per bug-fix pattern.

Defects4J data set to study the prevalence of the 5 most common bug-fix patterns identified in the work of Pan *et al.* [29]. 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.* [29]). Table III shows the results we obtained for each program.

We deepen our manual analysis in order to discover which types of addition are most common to fix bugs. Table IV 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 IV.

Current automatic software repair approaches like GenProg [18] and PAR [14] were designed to fix simple bugs. For instance, the majority of fixes produced by GenProg modify only one line of code. Westley Weimer, one of the inventors of GenProg, said that *the majority of fixes produced by humans are quite simple* [37]. 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 V shows the main descriptive statistics

TABLE III
PROGRAMS AND NUMBER OF REAL BUGS PER CATEGORY (IF-RELATED AND METHOD CALL) AND PER BUG-FIX PATTERN.

Program	#Bug Fixes	If-related	Method Call	IF-CC	MC-DAP	MC-DNP	AS-CE	IF-APC
Closure Compiler	133	67	15	24	6	1	5	43
Commons Lang	65	39	12	12	4	3	3	27
Commons Math	106	31	11	9	4	1	16	22
Joda-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
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

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 V are aligned with Weimer's affirmation. Furthermore, recent research into the nature of software changes [31] 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 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).

V. DISCUSSION

Lessons learned. The findings of our study 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 II. 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" [26] (i.e., NOPOL [40], SPR [21]). However, a recent work showed that NOPOL can automatically fix only 35 out of 224 bugs present in the *Defects4J* data set [23]. Moreover, patches by SPR only contain primitive values and do not contain object-oriented expressions (e.g., fields and method calls) [40].

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 [42]). 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 [42].

A. Threats to Internal Validity

Correctness of Boa programs. The correctness of our analysis depends on both our *Boa* programs and its Domain-Specific Types [12]. For example, we rely on *Boa* to identify bug-fix commits. However, precisely accomplishing this task is an open problem. To mitigate the risk of implementation errors, we released our *Boa* programs [13]. 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 kind or expression kind) only provide estimates of behavior. We consider our results as informative approximations.

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

[12] http://boa.cs.iastate.edu/docs/dsl-types.php#Expression
[13] https://github.com/eduardocunha11/BoaPrograms (verified 03/08/2017)

TABLE V
THE MAIN DESCRIPTIVE STATISTICS OF THE PATCHES CONSIDERED IN DEFECTS4J PER PROGRAM.

Patch size (LoC)	Closure Compiler	Commons Lang	Commons Math	Joda-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

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.

B. Threats to External Validity

Bug-fix patterns have incomplete coverage of bug fixes. Some bug-fix commits are associated with more than one bug-fix pattern studied. Concerning the coverage of bug fixes by bug-fix patterns, only 40.1423% and 42.3495% 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 III-A.

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 [11], [12]. Hence they might not be representative of closed-source development since different development processes could lead to different bug-fix patterns. Despite being open-source, several of the analyzed projects have substantial industrial participation.

VI. RELATED WORK

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 [39], AutoFix-E [38], ClearView [30], GenProg [18], Kali [34], NOPOL [40], PACHIKA [3], PAR [14], Prophet [22], SPR [21], RSRepair [33], Semfix [27], TrpAutoRepair [32], etc.).

NOPOL [40] 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 (aka. 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 [9] and predicate switching [41]. The evaluation was done on 22 real-world bugs from two large open-source projects.

SPR [21] addresses repairing conditional bugs, as well as other types of bugs, like missing non-IF statements. SPR combines *staged program repair* and *condition synthesis*. These techniques enable SPR to work productively with a set of program transformation schemas to generate and efficiently search a rich space of program repairs.

Empirical Knowledge on Automatic Program Repair. Zhong and Su [42] 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 [5] 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.* [29].

Martinez and Monperrus [24] analyzed the links between the nature of bug fixes and automatic program repair. Furthermore, the empirical study focuses on only one aspect of automatic program repair, namely the search space of fixing bugs. They mined repair models from manual fixes, and the mined repair models improve random search. Our study provides findings to better understand and improve these approaches. For example, we confirm that many bugs reside in source files of different programming languages or in non-source files like configuration files ([42]).

Soto *et al.* [36] 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 [14] and provided an informative approximation of their prevalence 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.* [29]. 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 kinds of statements that appear more frequently in bug-fix commits and the kinds of files that are usually changed to fix a bug.

VII. Conclusion and Future Work

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.* [29]. 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., [18]) but not one of them is dedicated to fix null pointer exceptions.

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.

Acknowledgment

We would like to thank the Brazilian agencies FAPEMIG, CAPES and CNPq for partially funding this research.

References

[1] V. R. Basili and B. T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Commun. ACM*, 27(1):42–52, Jan. 1984.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, 2013.

[3] V. Dallmeier, A. Zeller, and B. Meyer. Generating Fixes from Object Behavior Anomalies. In *Proc. ASE '09*, pages 550–554, 2009.

[4] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Proc. SANER' 2015*, pages 341–350.

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

[6] A. Endres. An Analysis of Errors and Their Causes in System Programs. In *Proceedings of the International Conference on Reliable Software*, pages 327–336. ACM, 1975.

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

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

[9] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *Proc. ICSE '10*, pages 215–224. ACM, 2010.

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

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

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

[13] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing Programs with Semantic Code Search (T). In *Proc. ASE' 2015*, pages 295–306. IEEE Computer Society, 2015.

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

[15] S. Kim and E. J. Whitehead, Jr. How Long Did It Take to Fix Bugs? In *Proc. MSR '06*, pages 173–174. ACM, 2006.

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

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

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

[19] M. Leszak, D. E. Perry, and D. Stoll. A Case Study in Root Cause Defect Analysis. In *Proc. ICSE '00*, pages 428–437. ACM, 2000.

[20] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proc. ASID '06*, pages 25–33. ACM, 2006.

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

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

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

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

[25] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proc. ICSE '16*, pages 691–701. ACM, 2016.

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

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *Proc. ICSE '13*, pages 772–781. IEEE Press, 2013.

[28] T. J. Ostrand and E. J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *Journal of Systems and Software*, 4(4):289–300, Nov. 1984.

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

[30] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically Patching Errors in Deployed Software. In *Proc. SOSP '09*, pages 87–102, 2009.

[31] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.

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

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The Strength of Random Search on Automated Program Repair. In *Proc. ICSE' 14*, pages 254–265. ACM, 2014.

[34] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proc. ISSTA' 2015*, pages 24–36. ACM, 2015.

[35] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proc. 10th ESEC/FSE*, pages 532–543. ACM, 2015.

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

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

[38] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *Proc. 19th ISSTA*, pages 61–72. ACM, 2010.

[39] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proc. 28th ASE*, pages 356–366, 2013.

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

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

[42] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. In *Proc. 37th ICSE' 15*, pages 913–923. IEEE Press, 2015.