Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J

Victor Sobreira*, Thomas Durieux†, Fernanda Madeiral*, Martin Monperrus‡, and Marcelo de Almeida Maia*
*Federal University of Uberlândia, Brazil, {victor, fernanda.madeiral, marcelo.maia}@ufu.br
†INRIA & University of Lille, France, thomas.durieux@inria.fr
‡KTH Royal Institute of Technology, Sweden, martin.monperrus@csc.kth.se

***Abstract*—Well-designed and publicly available datasets of bugs are an invaluable asset to advance research fields such as fault localization and program repair as they allow directly and fairly comparison between competing techniques and also the replication of experiments. These datasets need to be deeply understood by researchers: the answer for questions like "which bugs can my technique handle?" and "for which bugs is my technique effective?" depends on the comprehension of properties related to bugs and their patches. However, such properties are usually not included in the datasets, and there is still no widely adopted methodology for characterizing bugs and patches. In this work, we deeply study 395 patches of the Defects4J dataset. Quantitative properties (patch size and spreading) were automatically extracted, whereas qualitative ones (repair actions and patterns) were manually extracted using a thematic analysis-based approach. We found that 1) the median size of Defects4J patches is four lines, and almost 30% of the patches contain only addition of lines; 2) 92% of the patches change only one file, and 38% has no spreading at all; 3) the top-3 most applied repair actions are addition of method calls, conditionals, and assignments, occurring in 77% of the patches; and 4) nine repair patterns were found for 95% of the patches, where the most prevalent, appearing in 43% of the patches, is on conditional blocks. These results are useful for researchers to perform advanced analysis on their techniques' results based on Defects4J. Moreover, our set of properties can be used to characterize and compare different bug datasets.**

I. Introduction

Bug fixing is a hard and time-consuming task as it involves debugging, i.e., the process of identifying and correcting the root cause of a failure [1]. In the last decades, research aiming to automate tasks such as fault localization [2], [3], [4] and program repair [5], [6], [7], [8], [9], [10], [11] emerged to support developers at fixing bugs. To evaluate the effectiveness of the proposed techniques, researchers either create their own datasets and define ad-hoc baselines or rely on publicly available datasets of bugs (e.g. [12], [13], [14], [15], [16]. The latter is essential to advance those research fields as publicly available datasets allow directly and fairly comparison between competing techniques and also the replication of experiments.

Researchers in fault localization and program repair fields need detailed information on the datasets they use: 1) to only select bugs that have the required properties according to the technique under consideration (sampling and inclusion criteria), and 2) to perform advanced analysis of the performance of the newly proposed techniques depending on certain properties of the bugs or patches (correlation analysis).

We focus on the analysis of Defects4J [14], a dataset containing 395 real bugs collected from six open-source Java projects. Although extensively used in recent research on fault localization [17], [18], [19] and program repair [20], [21], [8], Defects4J does not come with fine-grained information about bugs and their patches. We contribute to Defects4J with the extraction and study of both quantitative (e.g. metrics) and qualitative properties (e.g. patterns) regarding patches. This new data is very valuable to 1) interpret past published results based on Defects4J under the light of the extracted properties; 2) provide and guide future research using Defects4J with fine-grained information; 3) understand the representativeness of different kinds of patches in Defects4J to suggest improved versions or new datasets; and 4) compare existing or future datasets of bugs with Defects4J.

We answer to four research questions based on the quantitative and qualitative properties:

RQ #1: Patch size by number of added, removed and modified lines;
RQ #2: Patch spreading by number of lines between chunks and by number of modified files, classes and methods;
RQ #3: Prevalence of repair actions over code elements (e.g. method call addition);
RQ #4: Prevalence of repair patterns (e.g. wraps-with `if`).

This study has important implications for future research on program repair, in particular: there is a need for techniques that leverage patches only containing addition of code; there is a research avenue for repair algorithms that are specific to repair patterns.

To sum up, our contributions are:

- The *anatomy of the patches in Defects4J* containing an extensive set of patch properties, consolidated into a JSON file[1] and augmented with a web user-interface to facilitate exploration[2];
- A *bug dataset dissection methodology* to extract valuable quantitative and qualitative properties regarding patches from bug datasets. The methodology is based on diff and advanced patch analysis and combines automated and manual thematic analysis;
- A *taxonomy of repair actions and patterns*, resulted from manual analysis of patches according our methodology.

[1]https://github.com/program-repair/defects4j-dissection
[2]http://program-repair.org/defects4j-dissection/

978-1-5386-4969-5/18 © 2018 IEEE

The remainder of this paper is organized as follows. Section II presents our methodology, including research questions and data collection. Section III presents the answers to the research questions with results and analysis. Section IV presents lessons learned and Section V discuss threats to validity. Section VI presents the related work, and Section VII presents the conclusions.

II. Methodology

In order to characterize and understand patches in Defects4J, we defined the following research questions.

RQ #1: *What is the size distribution of Defects4J patches?* Patch size can help to quantify the complexity and difficulty to fix a bug. Small patches have a great potential for repair automation and, in fact, many techniques for automatic program repair works well when this kind of patch is required. The size of human-written patches to fix bugs of a dataset is an important property to know on which bugs a given technique may succeed or not.

RQ #2: *To what extent are Defects4J patches spread in source code?* Bug fixing ranges from a single line to multiple lines, and it can be sequential or spread over methods, classes and files. We consider three types of spreading: number of chunks, spreading of chunks and number of modified files, classes and methods. Similar to patch size, patch spreading gives insights on how well a given technique can handle bugs in a dataset used to evaluate such technique. Some past bug datasets, especially those artificially generated, contain many single line bugs (e.g. Siemens suite [22], available in SIR [12]) and support studies from past to fewer years ago such as [23]. There is no guarantee that techniques effective on those bugs are also effective on multi-line spread bugs. Therefore, information about this diversity is essential to sustain any claim based on a specific bug dataset.

RQ #3: *What is the composition of Defects4J patches in terms of repair actions (i.e. addition, removal and modification) over code elements (e.g. conditional and method call)?* Actions required on code elements to produce a patch can be simple as a single change of a relational operator, or complex as the addition, removal, and modification of several lines of code, on different code elements, in multiple points in the source code. To proceed with a bug fixing, it is important to know whether a given technique can handle just simple cases as the former or whether such technique is elaborated enough to handle the latter case. Exposing this information from a bug dataset highlights gaps and improvement opportunities, and also avoids misjudgements on a technique being able to handle any type of bug.

RQ #4: *What repair patterns can be found in Defects4J using a manual thematic analysis [24]?* Many patches share some common structures [25]. We are interested in identifying abstractions occurring recurrently in patches that can involve compositions of repair actions. These abstractions shared by groups of patches are called in this paper as *repair patterns*. Knowledge on what repair patterns are present in a dataset can help to develop techniques capable to handle certain types of bugs, for instance by extracting templates for code synthesis.

A. Subject Dataset: Defects4J

Defects4J [14] is a dataset containing 395 real bugs (version 1.1), built to support software testing research. Bugs in Defects4J were collected from six open-source Java projects: JFreeChart (26 bugs), Closure Compiler (133 bugs), Apache Commons Lang (65 bugs), Apache Commons Math (106 bugs), Mockito Testing Framework (38 bugs) and Joda Time (27 bugs). For each bug, Defects4J delivers the buggy program version and its associated fixed version. Moreover, Defects4J bugs are 1) related to source code (i.e. fixes within the build system, configuration files, documentation, or tests are not included), 2) reproducible (each bug contains at least one test that exposes the bug), and 3) isolated (patches do not include unrelated changes to the bugs such as features or refactorings). In this paper, to refer to Defects4J patches, we use a simple notation with project name followed by bug id, for instance Closure-12 and Math-3.

B. Data Collection

For each bug, we first produced a *diff* view between the buggy program version and its associated fixed version. We used these views as source for data extraction and analysis, and they are available through hyper-links when patches are cited in this paper. Data collection procedures to answer each research question are described below.

1) Patch Size: We produced scripts to compute how many source code lines were added, removed or modified by a patch, based on the *diff* views. Addition and removal of lines vary between 1) consecutive lines (Chart-3) and sparsed lines (Chart-2), and 2) full statements (Closure-80), partial statements (i.e. closing brackets as in Chart-26) and line continuation (Closure-59). Lines are considered modified when sequences of removed lines are straight followed by added lines (or vice-versa). Thus, to count each modified line, a pair of added and removed lines is needed. Listing 1 shows an example of patch with one modified line (line 635), two non-paired removed lines (the old 636 and 639 lines), and none non-paired added line. By summing these lines, we have the metric **patch size** in number of lines, which in the example is 3 lines.

2) Patch Spreading: We calculated five metrics of patch spreading also through scripting. The first metric is **number of chunks** in a patch. A chunk is a sequence of continuous changes in a file, consisting of the combination of addition, removal, and modification of lines. A patch can be composed of one or more chunks, and this information can give us insights on how a patch is spread through the source code: a patch with a single chunk has no spreading, and the more chunks, the more the patch is spread. Listing 1 has two chunks: the first one is composed of the lines 635 and 636, and the second one is composed of the old line 639. The second metric is **spreading of chunks** in a patch. To measure chunk spreading, we consider the number of lines interleaving chunks

in a patch. In a patch with only one chunk, this value is naturally zero, because it represents a continuous sequence of changes. In a patch with two chunks, at least one line separates the chunks. For more chunks, naturally, this value tends to increase. An exception is for patches involving more than one file. For this case, we sum the spreading of chunks of all files to get the final spreading of the patch. For example, a patch with two modified files has zero spreading if the patch has just two chunks, one in each file. In Listing 1, between the old line 636 (end of the first chunk) and the old line 639 (beginning of the second chunk) there is only two lines, which is the value of the metric spreading of chunks in this case. It is worth to mention that empty and comment lines were discarded for chunk spreading calculations. These lines have no influence on program behavior, and considering them would make more sense for code readability, for example. The remaining metrics for patch spreading are **number of modified files**, **classes** and **methods**. We consider only source code files.

```
635      - JsName name = getName(ns.name, false);
     635 + JsName name = getName(ns.name, true);
636      - if (name != null) {
637 636      refNodes.add(new ClassDefiningFunctionNode(
638 637          name, n, parent, parent.getParent()));
639      - }
```

Listing 1. Patch for bug Closure-40.

3) Repair Actions: Diff views were reviewed manually, aiming to characterize their composition in terms of **repair actions** over code elements. Repair actions are the basic building blocks for patches. For example, Listing 1 has the actions "modification" of "method call parameter value" (line 635) and "removal" of "conditional (`if`) branch" (old lines 636 and 639). Repair actions provide fine-grained information beyond simple counting of addition, removal and modification of lines. An initial list was produced based on several potential repair actions. This list was augmented with other actions found during subsequent manual analysis of the patches. For each new repair action found, patches were reviewed to update the annotations on them. Repair actions without occurrences in Defects4J were discarded.

4) Repair Patterns: With knowledge about the content of patches acquired in repair action analysis, it was observed a recurrence of more abstract structures in patches, resembling patterns. For example, the modified line 635 of Listing 1 illustrates the repair pattern "Constant Change", while the removed lines 636 and 639 illustrate the repair pattern "Unwraps-from `if`" (both discussed in Section III-D). To confirm the existence of these **repair patterns**, a process based on Thematic Analysis (TA) was conducted. Originally, TA is "a method for identifying, analyzing, and reporting patterns (themes) within data" [24]. TA is a manual analysis that involves six steps: 1) familiarizing with the data (done with many reading and re-reading of patches to understand its composition); 2) identifying initial codes (in our context, a "code" is a single repair action); 3) searching for themes (combinations of repair actions re-appearing over many patches were identified, counted and named as repair patterns); 4) reviewing themes (at first glance some found themes appear to be relevant but after passing all patches it was not sustained, other themes were merged because they were similar, some themes were hierarchically organized and some were discarded); 5) defining and naming themes (although many themes were named at early steps, some of them were reviewed and renamed to better reflect their meaning, and criteria to recognize each instance were defined and exposed in Section III-D); 6) producing the report (this paper and complementary online material reflect this step, compiling the main results of this analysis).

For both repair actions and patterns, two main activities were performed: 1) the identification of existing repair actions and patterns in the patches, which resulted in a taxonomy of repair actions and repair patterns, and 2) the annotation of the patches in Defects4J using such taxonomy. These two activities were performed manually by the first author of this paper. Then, two other authors validated all the annotated patches by reviewing which repair actions and patterns from the taxonomy a given patch contains, and some adjustments were done.

C. Data Availability

All the data collected on Defects4J with this methodology is consolidated into a JSON file, which is publicly available in an open-science repository[3]. We also have created a web user-interface to present that data for researchers to easily browse, filter, and understand the patches of Defects4J:

http://program-repair.org/defects4j-dissection/

The web user-interface is augmented with runtime information on the Defects4J bugs, which consists of the exceptions threw when running (failing) tests on the buggy program versions, and repair tools that have fixed Defects4J bugs.

III. Results and Analysis

In this section, we present the results and the answers to our four research questions.

A. Size of the Defects4J Patches (*RQ #1*)

Patch size results are presented as follows: we first show the repartition and intersection of patches among three sets, i.e. added, removed and modified lines; then the results for each of these sets; and finally the total size of the patches.

1) Repartition and Intersection of Patches: Figure 1 presents the Venn diagram on the intersections among the three types of changes on lines (addition, removal and modification), where each number is the number of patches that contain added, removed and/or modified lines. For example, there are 107 patches containing only modified lines, and there are 106 patches containing both added and modified lines. We note three interesting facts. First, 118 out of 395 patches (29.87%) contain only added code. This contradicts a common intuition that patches contain mainly code modification. Second, nine patches fix bugs only by removing code, which illustrates that the correct behavior for few cases is already present in the program. Third, 234 patches (59.24%) are exclusive for one of the three types of changes on lines.

[3]https://github.com/program-repair/defects4j-dissection

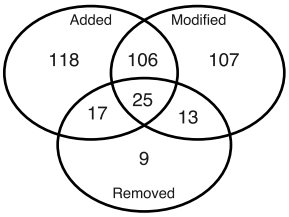

Fig. 1. Venn diagram of patches that contain added, removed and/or modified lines.

2) Added Lines: Addition of lines ranges from 0 to 48 lines as shown in Table I. 25% of the patches have no added line, and at most two lines are added in half of the patches. 95% of the patches have added lines ranging from 0 to 19. Beyond 19 lines, the additions occur in outlier patches.

3) Removed Lines: Numbers on removed lines dropped down considerably when compared with added lines (see the two first lines in Table I). For 95% of the patches, no more than six lines are removed.

4) Modified Lines: As shown in Table I, 25% of the patches have no modified line, half of the patches have at most one modified line, and 95% of the patches have at most four modified lines.

5) Patch Size: The total size of a patch is the sum of added, removed and modified lines in the patch. As shown in Table I, a patch involves at least one line and at most 54 lines. For 25% of the patches, at most two lines are involved. To cover 95% of the patches, at most 22 lines should be considered.

RQ #1: What is the size distribution of Defects4J patches?
Findings: The median size of Defects4J patches is four lines. Addition of lines predominates over removal and modification. In Defects4J, large patches are rare: only 5% of the patches involve more than 22 lines, and the maximum is 54 lines.
Implications: Current repair techniques are only capable of creating small patches. Our results show that Defects4J is an appropriate dataset for program repair because it mostly contains small patches. This confirms the results of [20]. Since around 30% of the patches contain only code addition, research on repair systems should leverage this assumption and define addition-based patches as an important repair search space.

*B. Spreading of the Defects4J Patches (**RQ #2**)*

In this section, we analyze the spreading of the patches by the number of 1) chunks, 2) lines between chunks, 3) modified files, 4) modified classes, and 5) modified methods.

1) Number of Chunks: In Defects4J, the patches contain between one and 20 chunks (see Table I). Half of the patches contain two or less chunks, and 95% of the patches contain at most eight chunks.

2) Spreading of Chunks: 25% of the patches in Defects4J have no spreading of chunks. Half of the patches have a spreading of no more than one line. To cover 95% of the patches, it is enough to consider spreading of 214 lines (see Table I). Considering that 95% of the patches have maximum size of 22 lines and maximum spreading of 214 lines, it is reasonable to state that 95% of the patches in Defects4J are bounded to blocks of 236 lines overall.

TABLE I
DESCRIPTIVE STATISTICS FOR PATCH SIZE AND SPREADING.

	Min	25%	50%	75%	90%	95%	Max
# Added lines	0	0	2	6	12	19	48
# Removed lines	0	0	0	0	2	6	24
# Modified lines	0	0	1	2	3	4	27
Patch size	1	2	4	9	18	22	54
# Chunks	1	1	2	3	5	8	20
Spreading	0	0	1	18.5	88.2	213.5	1,332
# Files	1	1	1	1	1	2	7
# Classes	1	1	1	1	1	2	7
# Methods	0	1	1	2	2	3	20

3) Modified Files: 92.41% of the patches modify only one file, and 7.09% of the patches modify two files. Therefore, for Defects4J bugs, techniques need to access at most 2 files to deal with 99.5% of the bugs, and if optimized to work with one file, they still cover more than 90% of the bugs. Just in exceptional cases, patches modify more than two files, which is the case of Mockito-19 (five files) and Math-6 (seven files).

4) Modified Classes: We observed the number of modified classes is highly related to the number of modified files in a patch (see Table I). Only eight patches modify more classes than files.

5) Modified Methods: We observed two interesting facts when analyzing the number of modified methods in the patches. First, there are two patches that do not modify methods, they only change class and field declaration. Second, 27% of the patches (107) change more than one method, and 47% of these patches are related to the patterns *Copy/Paste* and *Missing Null-Check* (see Section III-D).

RQ #2: To what extent are Defects4J patches spread in source code?
Findings: 151 patches (38.23%) are composed of a single continuous chunk. In terms of chunk spreading, 207 patches (52.41%) have only one code line separating the chunks. Only two patches affect more than two files.
Implications: The majority of program repair techniques perform single-point repair by modifying a single location in the code (e.g. GenProg [5], Nopol [9], Astor [8], Elixir [26], ssFix [27], HDRepair [21], PAR [6]). This corresponds well to the 151 single-chunk human-written patches in Defects4J. However, the remaining 244 multi-chunk patches show that there is a need for multi-point program repair, such as Angelix [7]. As only two patches affect more than two files, single-file fault localization is appropriate for program repair techniques targeting bugs similar to those contained in Defects4J.

*C. Repair Actions in the Defects4J Patches (**RQ #3**)*

In this section, we present the repair actions over code elements found in the Defects4J patches.

1) Assignment: we consider assignment statements containing the simple assignment operator (=), unary increment

(x++)/decrement (x--) operators and assignments compound of arithmetic operators (e.g. x+=1). Repair actions related to assignments are:

- Assignment Addition: an assignment to a variable is considered added when it appears in the lines added by the patch and there is no assignment involving such variable in the removed lines. Line 2401 of Closure-133 shows an assignment addition for the variable unreadToken;
- Assignment Removal: an assignment to a variable is considered removed when it appears in the lines removed by the patch and there is no assignment involving such variable in the added lines. Line 145 of Chart-12 shows the removal of the assignment for the variable this.dataset;
- Assignment Modification: an assignment to a variable is considered modified in a patch when its expression changed, i.e., when it appears in the removed and added lines. In Time-7, the value assigned to the variable defaultYear changed.

2) Conditional: constructions considered regarding conditional branches are simple if, if-else (including compact form cond?a:b, i.e. if-else-expression), simple else and case in switch structure. The basic repair actions regarding conditionals are:

- Conditional Branch Addition: Mockito-8 has an addition of a simple if in line 79;
- Conditional Branch Removal: Closure-11 has an else-if branch removal in lines 1314 and 1315.

Moreover, the conditional expression can be modified, which can happen in three ways:

- Conditional Expression Modification: Chart-1 shows an example of a simple change in the conditional expression composition in line 1797;
- Conditional Expression Expansion: Closure-99 shows an expansion of a conditional expression composition in line 92;
- Conditional Expression Reduction: Chart-5 shows a reduction of a conditional expression composition in line 552.

3) Loop: we considerer the loop constructions for, while and do-while. Changes related to these are:

- Loop Addition: addition of a new loop (Closure-129);
- Loop Removal: removal of an existing loop (Math-56);
- Loop Modification: modification of conditional test (Lang-19, line 1050 to 1054) or initialization variables (Math-41, line 520).

4) Method Call: changes related to method calls are present in the majority of the patches. These changes manifest in the following forms:

- Method Call Addition: call addition (Chart-5, line 545) or parameter addition, i.e., the call is replaced by overloaded version with more parameters (Closure-3, line 155);
- Method Call Removal: call removal (Lang-40, line 1048) or parameter removal, i.e., the call is replaced by overloaded version with less parameters (Math-66, line 62);
- Method Call Modification: method call replacement (Closure-4, lines 190 and 202), method call moving (Closure-102, lines 89 and 94), parameter value modification (Lang-59, line 884), or parameter value swapping (Time-4, line 464).

5) Method Definition: changes related to method definitions and signatures can be observed by:

- Method Definition Addition: complete method definition addition (Closure-8, line 209 to 211) or parameter addition (Closure-3, line 280);
- Method Definition Removal: complete method definition removal (Closure-46, lines 140 to 155) or parameter removal (Math-66, lines 94 and 95);
- Method Definition Modification: method renaming (Mockito-21, line 20), changes in parameter types (Lang-30, 1443 and 1497), return type (Lang-29) and modifier (Mockito-21, line 20), and changes related to addition and removal of overriding method (Closure-28).

6) Object Instantiation: the instantiation of objects is observed by the keyword new, and the changes related to it are:

- Object Instantiation Addition: Mockito-36 shows an instantiation addition in line 204;
- Object Instantiation Removal: Math-58 shows an instantiation removal in line 121;
- Object Instantiation Modification: Math-6 shows an instantiation modification in line 51.

7) Exception: repair actions related to exception handling and throwing are:

- Exception Addition: addition of try-catch block (Closure-83) or throw statement (Time-15, line 139);
- Exception Removal: removal of try-catch block (Math-60) or throw statement (Mockito-1).

8) Return: repair actions related to return statements are:

- Return Addition: Some patches add a return statement wrapped with an if condition, making them new exit points in the program control flow. Lang-49 illustrates this case;
- Return Removal: The opposite of the previous case also happens. Closure-11 shows the removal of a return statement wrapped with an if condition in line 1315;
- Return Expression Modification: Changes in return expression is also common in patches of Defects4J. Math-105 shows an example in line 264.

9) Variable: repair actions related to variable declaration and usage are:

- Variable Addition: addition of a new variable declaration (Lang-40, lines 1048 and 1049);
- Variable Removal: removal of an existing variable declaration (Math-56, line 237);
- Variable Modification: modification of the variable type (Chart-17, line 857), or modifier (Mockito-23, lines 44 and 45), or replacement of the usage of a variable by another one (e.g., Lang-59, line 884) or by a method call (Math-34, line 209), preserving the context where it is applied.

10) Type: patches including changes in types are:

- Type Addition: addition of type (Mockito-23, line 136);
- Type Modification: implementation of interface (Math-12).

TABLE II
REPAIR ACTIONS ACRONYMS AND GROUPING NAMES.

Acronym	Action	Group
asgn	A/R/M	Assignment
cnd	A/R/M	Conditional
lp	A/R/M	Loop
mc	A/R/M	Method Call
md	A/R/M	Method Definition
obj	A/R/M	Object Instantiation
ex	A/R	Exception
ret	A/R/M	Return
var	A/R/M	Variable
ty	A/M	Type

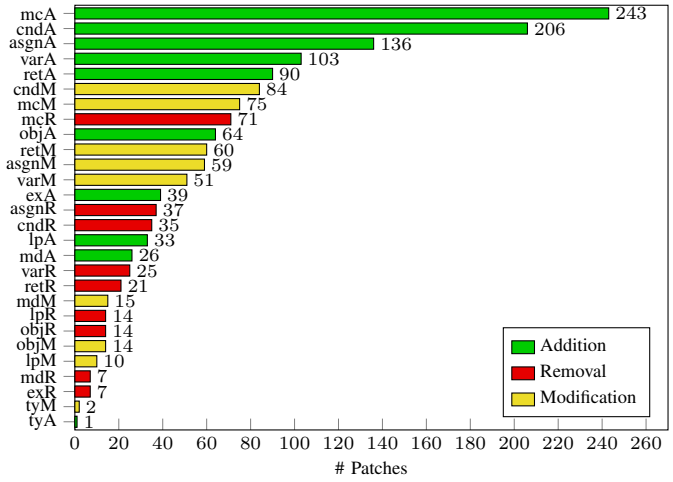

Fig. 2. Incidence of the repair actions in patches.

Figure 2 shows the ranking of the repair actions over code elements (vertical axis) concerning the number of patches (horizontal axis) where they occur. We grouped repair actions that belong to the same group (e.g. method call and method call parameter addition belong to the group *Method Call Addition*) to avoid a too fragmented graph. We also contracted repair action group names to reduce visual pollution. Table II shows, for each group (e.g. *Method Call*), its acronyms (e.g. mc) and suffix letters of the existing action types for it (A=Addition, R=Removal and M=Modification), which combined form the contracted name of a repair action group (e.g. "mcA" represents *Method Call Addition*). To make easier to understand the graph, green bars represent addition, red bars represent removal, and yellow bars represent modification actions.

Method Call Addition is the most prevalent repair action in the patches (243 patches), followed by *Conditional Branch Addition* (206 patches) and *Assignment Addition* (136 patches). Together and discounting co-occurrences, these three repair actions cover 77.21% of the patches. In all cases, adding structures surpass removing or modifying existing ones.

Figure 3 presents the distribution of number of actions per patches. The median (highlighted in red) shows that 50% of the patches have no more than three types of repair actions. Some outlier patches were found, containing 10-15 repair actions and a maximum of 19 repair actions.

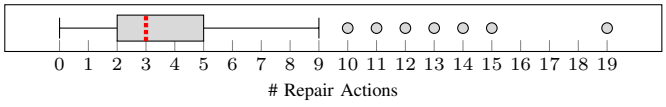

Fig. 3. Distribution of number of repair actions per patches.

RQ #3: What is the composition of Defects4J patches in terms of repair actions over code elements?
Findings: Addition of method calls, conditionals and assignments are the top-3 applied actions by patches appearing in 305 patches (77.21%). The additions surpass removals and modifications, which confirms the findings in the RQ #1 at line level.
Implications: Program repair research has mostly focused on conditional statements and assignments [9], [28], [29]. However, handling method calls with rich side-effects has been rather neglected. To our knowledge, only generate-and-validate techniques à la GenProg are capable of synthesizing such patches. Assuming Defects4J well represents the distribution of real patches, this calls for more research on repair strategies handling method calls.

*D. Repair Patterns in the Defects4J Patches (**RQ #4**)*

We found nine repair patterns from the patches in Defects4J, which are presented in this section in descending order of prevalence. Examples for these patterns are shown in Figure 4.

1) Conditional Block: As shown by the most frequent repair actions in the previous section, the majority of Defects4J bugs require more addition than removing or just changing an existing code. This leads us to the first type of repair pattern, which seems to fill a gap for missing conditional blocks. Following variants of this repair pattern can be found:

- Conditional Block Addition: involves the addition of a new conditional block (e.g. `if-then`) in the program (Lang-45, lines 616 to 618);
- Conditional Block Addition with Return Statement: involves the addition of a conditional block that also includes a return statement (Closure-5, lines 176 to 178);
- Conditional Block Addition with Exception Throwing: involves the addition of a conditional block that also throws an exception (Math-48, lines 189 to 191).

There are also patches where conditional blocks are removed as in Math-50.

2) Expression Fix: This repair pattern occurs in patches with actions impacting existing logic or arithmetic expressions.

Logic Expression Fix mainly occurs in conditional expression (in branches and loops), while *Conditional Block* and *Wraps-with* patterns also impact the code in branch body. Moreover, *Logic Expression Fix* also occurs in return expressions and expressions of assignments to boolean variables, and it has the following variants:

- Modification: occurs when an existing logic expression is modified, as in Chart-1, where the logic operator in the conditional expression was changed;
- Expansion: occurs when an existing logic expression is preserved and extra logic is added, as in Mockito-34;
- Reduction: occurs when an existing logic expression has a part of the logic removed, as in Closure-18.

```
// Conditional Block Addition (Lang-45)
616 + if (lower > str.length()) {
617 +     lower = str.length();
618 + }
// Conditional Block Addition with Return Statement (Closure-5)
176 + if (gramps.isDelProp()) {
177 +     return false;
178 + }
```

Pattern 1 – *Conditional Block*.

```
// Logic Expression Modification (Chart-1)
1797 - if (dataset != null) {
1797 + if (dataset == null) {
// Logic Expression Expansion (Mockito-34)
106 - if (m instanceof CapturesArguments) {
106 + if (m instanceof CapturesArguments && i.getArguments().length > k) {
// Arithmetic Expression Modification (Math-80)
1135 - int j = 4 * n - 1;
1135 + int j = 4 * (n - 1);
```

Pattern 2 – *Expression Fix*.

```
// Wraps-with if (Time-3)
662 + if (years != 0) {
663     setMillis(getChronology().years().add(getMillis(), years));
664 + }
// Wraps-with if-else-exp (Mockito-29)
29 - description.appendText(wanted.toString());
29 + description.appendText(wanted == null ? "null" : wanted.toString());
// Unwraps-from method call (Closure-9)
183 - String moduleName = guessCJSModuleName(normalizeSourceName(script.
      getSourceFileName()));
184 + String moduleName = guessCJSModuleName(script.getSourceFileName());
```

Pattern 3 – *Wraps-with*.

```
// Single Line (Mockito-34)
106 - if (m instanceof CapturesArguments) {
106 + if (m instanceof CapturesArguments && i.getArguments().length > k) {
```

Pattern 4 – *Single Line*.

```
// Wrong Variable Reference (Chart-11)
275 - PathIterator iterator2 = p1.getPathIterator(null);
275 + PathIterator iterator2 = p2.getPathIterator(null);
// Wrong Method Reference (Closure-10)
1417 - return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
1417 + return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
```

Pattern 5 – *Wrong Reference*.

```
// Missing Null-Check and Non-Null-Check (Chart-15)
1378 +     if (this.dataset == null) {
1379 +         return 0.0;
1380 +     }
     [...]
2054 +     if (this.dataset != null) {
2055          state.setTotal(DatasetUtilities.calculatePieDatasetTotal(
2056                  plot.getDataset()));
2057 +     }
```

Pattern 6 – *Missing Null-Check*.

```
// Copy/Paste of Method Call Replacement (Chart-19)
698 + if (axis == null) {
699 +     throw new IllegalArgumentException("Null 'axis' argument.");
700 + }
[...]
976 + if (axis == null) {
977 +     throw new IllegalArgumentException("Null 'axis' argument.");
978 + }
```

Pattern 7 – *Copy/Paste*.

```
// Change in string (Closure-65)
1015 - case '\0': sb.append("\\0"); break;
1015 + case '\0': sb.append("\\000"); break;
// Replacement of constant variable (Closure-14)
767 - cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
767 + cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
```

Pattern 8 – *Constant Change*.

```
// Code Moving (Closure-13)
126 - traverse(c);
126   Node next = c.getNext();
127 + traverse(c);
```

Pattern 9 – *Code Moving*.

Fig. 4. Code snippet examples for repair patterns.

Arithmetic Expression Fix mainly occurs in assignment and return statements, as in Math-80 and Math-2, respectively.

3) Wraps-with: This repair pattern resembles the *Conditional Block* pattern, but its intrinsic feature is the wrapping of an existing code with a conditional branch. Indeed, the wrapping structure goes beyond conditionals and can also involve try-catch blocks, method calls and loops. The following are the variants for *Wraps-with*:

- if: occurs when an existing code is wrapped with a conditional logic using an `if` expression. Time-3 illustrates this repair pattern applied ten times;
- if-else: occurs when an existing code is wrapped with an `if-else` expression, where the existing code can be placed in the `then` or in the `else` block. Mockito-29 illustrates this repair pattern, where the method call `wanted.toString()` was wrapped with an `if-else` expression of type `cond?a:b`;
- else: occurs when an existing code is wrapped with an `else` complementing previous written `if` or `if-else` (just Chart-21);
- try-catch: occurs when an existing code is wrapped with a new `try-catch` block (Closure-83);
- method: occurs when an expression is wrapped with a method call (Math-105);
- loop: occurs when a statement or block of code is wrapped with a loop, turning it from a simple sequence of code into a repeating one (Closure-124).

There are cases where *Wraps-with* is more subtle, as in Mockito-29, line 29. It shows the wrapping of the method call `wanted.toString()` with an `if-else` expression of type `cond?a:b`.

The inverse of *Wraps-with* pattern is also found, called *Unwraps-from*. Closure-9 shows the unwrap of the method call `script.getSourceFileName()`. Chart-18 shows the most common case, *Unwraps-from if-else*.

4) Single Line: Patches with the *Single Line* pattern are patches with one line addition, one line removal or both (line modified). We also consider as *Single Line* the special cases when a single statement spans multiple lines (Closure-55) or is moving (Closure-13). Many of current techniques in fault localization and program repair fields works well where this kind of fixing is required since many of previous datasets are based on seeded faults or mutants [19]. These patches involve small repair actions, such as variable replacement by another variable (Chart-11) and conditional expression expansion (Mockito-34).

5) Wrong Reference: A wrong reference occurs when a variable or a method call is referenced by mistake instead of another variable or method call. In the patch, the wrong reference is replaced by another one. Examples are:

- Wrong Variable Reference: Chart-11 shows the patch for a wrong variable reference, which is replaced by another variable. In some cases, the wrong variable reference is replaced by a method call such as Lang-57;
- Wrong Method Reference: Closure-10 shows a wrong method reference, which is replaced by another method

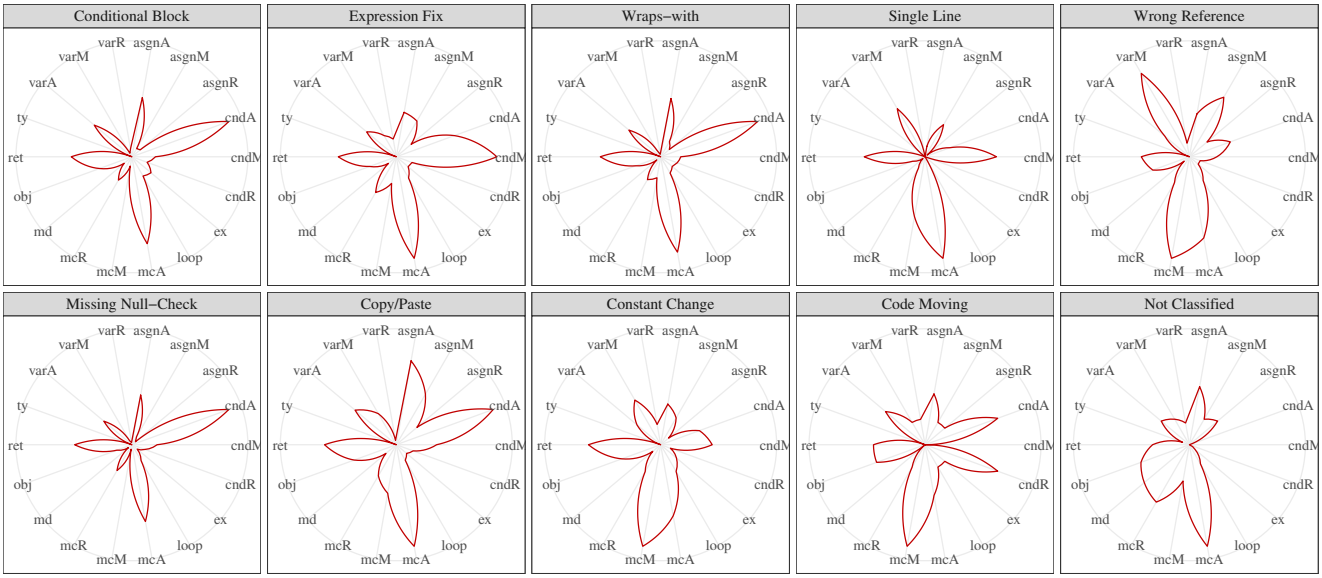

Fig. 5. Composition of the repair patterns in terms of repair actions.

reference. In some cases, the wrong method reference is less perceptive (e.g. overloaded method call replacements) such as in Math-70. In a few cases, the wrong method reference is replaced by a variable (Math-67).

6) Missing Null-Check: This repair pattern is related to the addition of conditional expressions or expansion of existing ones with null-checks. There are examples of *positive null-checks*, where a reference is checked for nullity (line 1378 of Chart-15), and *negative null-checks*, where a reference is checked for non-nullity (line 2054 in Chart-15).

7) Copy/Paste: Some patches repeat the same change in different points, resembling a copy-paste operation. Chart-19 shows the addition of the same *Conditional Block with Exception Throwing* in two different methods. Math-71 illustrates a case with non-exact fixing code applied to two different files.

8) Constant Change: This pattern is dedicated to changes of constant values in the code. A constant value is either a literal, i.e. a value fixed in the code, or a constant variable, i.e. a variable with final value that cannot be modified by the program during execution. Closure-14 shows an example of a constant variable being replaced by another constant variable. For literals, we found patches where string (Closure-65), boolean (Math-22), integer (Lang-19), and floating-point number (Mockito-26) were modified.

9) Code Moving: Some patches involve moving code lines around, without extra changes to these lines. Although there are not many examples of this repair pattern, it deserves attention because some patches consist basically of this type of change, and it may consist of single line as in Closure-13, or multiple lines as in Closure-117.

Figure 5 shows the overall composition of the repair patterns. Each panel corresponds to one of the nine repair patterns. The radial axis corresponds to the repair actions presented in the previous section that co-occur in patches where the repair pattern was found. For instance, in *Expression Fix* panel, it is clear that *Conditional Modification (cndM)* and *Method Call Addition (mcA)* repair actions are the most prevalent in patches containing the *Expression Fix* pattern, while *Type (ty)* actions almost do not appear for this repair pattern. Some repair patterns show similarities, e.g. *Conditional Block*, *Wraps-with* and *Missing Null-Check*. In fact, one of the main differences between *Conditional Block* and *Wraps-with* is the presence of non-patch (or wrapped) code between wraps, which does not influence in repair pattern composition. Furthermore, *Missing Null-Check* is present in many conditionals in patches where *Wraps-with* and *Conditional Block* are involved. Other repair patterns have distinctive silhouette, e.g. *Wrong Reference*, pointing out the more recurrent and distinctive actions in these repair patterns. The 22 patches where no repair pattern was found are shown in last panel (*Not Classified*), which points out that unclassified patches are more related to *Method Call Addition* action.

Figure 6 presents the ranking of the repair patterns (vertical axis) concerning the number of patches (horizontal axis) where they occur. *Conditional Block* is the most prevalent repair pattern found in the patches, followed by *Expression Fix* and *Wraps-with*. On the other hand, *Constant Change* and *Code Moving* are the less prevalent among the repair patterns.

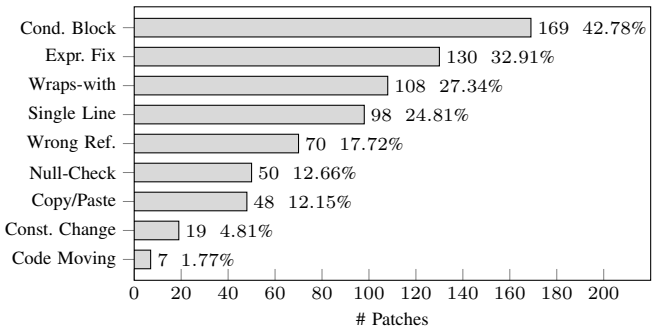

Fig. 6. Incidence of the repair patterns in patches.

Figure 7 presents the distribution of number of patterns per patches. The median (highlighted in red) and the upper quartile are the same, showing that most of the patches (75%) have no more than two repair patterns. Some outlier patches were found, containing between four and seven repair patterns.

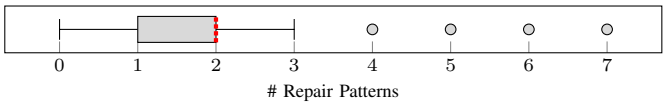

Fig. 7. Distribution of number of repair patterns per patches.

RQ #4: What repair patterns can be found in Defects4J using a manual thematic analysis?
Findings: Nine potential repair patterns were identified, which span 373 patches (94.43%). This show the clear presence of recurring patch techniques. *Conditional Block* repair pattern is the most prevalent, appearing in 169 patches (42.78%).
Implications: Some of the identified repair patterns can guide, for example, the development of program repair tools for specific bug types. For instance, a variant of the repair pattern *Conditional Block* is the *Conditional (if) Block Addition with Exception Throwing*. It appears in 33 patches and can be synthesized, as confirmed by the recent work on ACS [11].

IV. Lessons Learned

For most bugs in Defects4J, developers added more code than they removed or rewrote existing code. This is first perceived in findings of the RQ #1, where we studied patch size in terms of added, removed and modified lines. The findings of the RQ #3 also ground this point when we found that most of repair actions are related to addition of code. Researchers and tool builders should also focus on techniques capable to synthesize code, beyond the ones that just modify or remove code, especially for automatic repair. For fault location, the focus should be extended to find the location of missing code and not only to find the location of wrong code (which would mainly lead to modify or remove code).

Delimiting the applicability of a technique by exposing the characteristics of a dataset used to evaluate it is important. For instance, works as [30] already shown that the performance of a program repair tool changes considerably depending on the types of bug such tool is applied, even for Defects4J projects. Therefore, when reporting results of fault location or automatic repair techniques, it must be clear that not all types of bug would be handled and the dataset used would not contain some bug types, e.g. Defects4J does not cover well bugs spread through many files as pointed out by the findings of the RQ #2. Techniques previously evaluated on seeded bugs should be also evaluated on real bugs to confirm their efficacy, as the nature of the former may not reflect the latter. This also shows how fragile can be a technique that relies only on bugs from small or poor designed datasets, not paying attention at overfitting on it. If real bugs are not well represented in the dataset, the technique may be useless in practice.

The found repair patterns can help to categorize and group patches, leading to a reduced effort while trying to understand what kind of solution can be applied to an existing bug. They can help to segment the dataset, avoiding the need to think about a specific solution for every bug. As pointed out by findings of the RQ #4, the repair patterns are present in most of the Defects4J patches, and this recurrence may help to optimize the search for fixes with potential to be shared between many bugs.

V. Threats to Validity

Although this study provides a deep overview of human patches applied to bugs in six open source projects, our findings may be restricted to the projects in Defects4J. Defects4J can be considered a small dataset of bugs and may not represent well the existing types of bugs and the frequency they occur in the real world. To investigate whether our findings can be generalized to other projects, a follow-up of this work is needed. Although Defects4J projects are Java libraries, they are not necessarily related and, even then, metrics and insights found are consistent between these projects.

The nature of bugs in Defects4J should be also considered when developing solutions for fault location, automatic repair or other applications. Patches in Defects4J are isolated and may not reflect usual commits accompanied by unrelated changes and other kinds of noisy data that an technique should handle. Non-source code bugs were not covered in this work, since there is no such type of bug in Defects4J by design. We found two duplicated bugs in Defects4J dataset (i.e., Closure-$\{62 = 63, 92 = 93\}$). The duplicates were not removed to conduct our analysis and we consider that the impact is negligible over our findings and implications. However, to avoid bias in favor of some bug types, these duplicates may be discarded for other studies or applications.

The taxonomy of repair actions and patterns and the annotation of the patches in Defects4J using such taxonomy are results from manual analysis of patches. Although the patches were carefully analyzed by the first author of this paper and reviewed by other two authors, as any manual work, this one is not free of small mistakes or misinterpretation. Despite the value of manually analyzing human patches, this is a very difficult and time-consuming task. For this reason, by doing this, insights on how to automate the collection of the properties manually extracted from patches were obtained.

We considered all the repair patterns independently in each patch. Since different repair patterns are counted separately, pattern composition in terms of repair actions may be affected by other repair patterns (or other unrelated repair actions) present in the same patch. As a consequence, the identification of a pattern in a patch only implies this pattern is part of such patch. The patch can still involve a much more rich context with other patterns and repair actions beyond an identified pattern. A deeper study on correlations and co-occurrences of patterns would be necessary to obtain an accurate and more detailed description on patterns composition in terms of repair actions (as loosely illustrated in Figure 5).

VI. Related Works

A. Analysis on Defects4J bugs

In a recent work, Motwani et al. [31] annotated each bug in Defects4J with eleven abstract parameters regarding five defect characteristics: defect importance, complexity, independence, test effectiveness, and characteristics of the human-written patch. One example of abstract parameter is the number of lines edited in a patch, which is used to compute the defect complexity. Similar to our work, they annotated Defects4J bugs with patch size and number of modified files. On the characteristics of the patches, they annotated the bugs with nine code modification types, such as whether the patch contains addition of method calls, which are similar to our repair actions. However, our taxonomy of repair actions is more comprehensive and fine-grained, since we arranged the actions in groups considering more detailed changes. For instance, instead of having the information that a patch changed arguments in a method call, we have the information that an argument was added or removed, or that an argument value was changed or swapped with another one in a method call. Moreover, Motwani et al. considered other information than us, such as the number of relevant test cases, which makes our work and their work complementary to each other.

B. Patch Analysis of Bug Datasets

Several bug datasets have been proposed to support empirical studies on techniques and tools related to software bugs. Usually, these datasets do not include detailed information on the *bugs* and their *patches* if any (e.g., Siemens suite [22] and SIR [12]), or they include simple information on the *bugs* (e.g., BugBench [32]), like bug type. In this section, we present notable and recent bug datasets where information about the *patches* are delivered, which is close to our work on Defects4J.

iBugs [13] (390 Java bugs) contains bugs annotated with size and syntactic properties on their patches. iBugs' size properties include similar patch size and spreading metrics as our work. iBugs' syntactic properties consist of fingerprints describing which syntactic tokens the patch changed, such as keywords, method calls, and expressions, augmented with information on variable usage, operators and literals. These fingerprints are similar to our repair actions, but our taxonomy is organized in a different way. For instance, the groups of token "keyword" and "expression" in iBugs represent different changes on `if`; we have the repair action group "Conditional" that is dedicated to changes on conditionals. Moreover, our analysis includes repair patterns.

ManyBugs [15] (185 C bugs), besides information on the bugs, delivers manually evaluated information about patches. For each patch, ManyBugs' authors note whenever some changes happened, such as whenever functions, loops, conditional and function calls were added, and whenever arguments to a function or function signature were changed. This is close to our repair actions, but our repair actions are more fine-grained. Similar to our work, they also calculated the number of lines changed (size) and number of files changed (spreading), but different from our work, ManyBugs does not provide number of chunks and repair patterns.

Codeflaws [16] (3902 C bugs) delivers bugs annotated with syntactic differences between buggy and patch code at AST level. Like in iBugs, Codeflaws' syntactic differences are similar to our repair actions, but we use a more comprehensive taxonomy; for example, in Codeflaws, conditionals and loops are considered together in one group, "control flow". Moreover, Codeflaws delivers neither information on patch size and spreading, nor repair patterns.

C. Patch Analysis on Other Resources

Pan et al. [33] and Soto et al. [34] identified patterns in human patches. Pan et al. [33] manually analyzed seven open-source projects and found 27 bug fix patterns covering from 46 to 64% bug fixes. They observed the most common bug fix patterns are related to method call and if condition (both are around 20% bug fixes), which is consistent with our findings, since *Method Call Addition* and *Conditional Branch Addition* are the most prevalent repair actions in patches.

Soto et al. [34] focused on identifying how many human patches contain the repair patterns presented by [6]. They analyzed 4,590,679 bug fix commits and found that less than 15% commits contain one of these patterns. The differences between these two works and this work are the focus on a different dataset, and the collection of additional metrics to characterize human patches, e.g. patch size and spreading.

VII. Conclusions

Research fields related to software bugs require well-designed, publicly available and real bug-based datasets. Defects4J aims at targeting these goals, but lacked an in-depth study to inform the potential users on its contents and characterization of bugs. To fill this gap, we analyzed the anatomy of the Defects4J patches considering four properties: patch size and spreading, and repair actions and patterns. We found that 95% of the patches in Defects4J involve at most 22 lines, have low spreading in lines (at most 214 lines) and in files (at most two files). Most repair actions involve addition of method calls, conditional branches and assignments (77.21%). Nine repair patterns were found in 94.43% of the patches, and *Conditional Block* is the most prevalent one (42.78%).

Our findings have important implications for those interested in the usage of bug datasets: Defects4J is an appropriate dataset to program repair research; repair based on addition of code should receive as much attention as repair based on modification of code; multi-point program repair is not just a trend but a need for proposed techniques; single file repair responds to the most part of bugs in Defects4J; automatic patch generation can rely on high prevalence of repair actions (e.g. *Method Call Addition*) and patterns (e.g. *Conditional Block*) found. This study is to help researchers to take better and informed decisions around bug dataset choice and comparison.

Acknowledgement

We acknowledge CNRS, CAPES (SticAmSud Program), CNPq and FAPEMIG for partially funding this research.

REFERENCES

[1] A. Zeller, "Automated Debugging: Are We Close?" *Computer*, vol. 34, no. 11, pp. 26–31, Nov 2001.

[2] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. New York, NY, USA: ACM, 2002, pp. 467–477.

[3] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. New York, NY, USA: ACM, 2005, pp. 273–282.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[5] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[6] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-Written Patches," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York, NY, USA: ACM, 2016, pp. 691–701.

[8] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. New York, NY, USA: ACM, 2016, pp. 441–444.

[9] 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*, vol. 43, no. 1, pp. 34–55, 2017.

[10] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming," in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*, 2017, pp. 349–358.

[11] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise Condition Synthesis for Program Repair," in *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 416–426.

[12] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[13] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. New York, NY, USA: ACM, 2007, pp. 433–436.

[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. New York, NY, USA: ACM, 2014, pp. 437–440.

[15] C. Le 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*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[16] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools," in *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 180–182.

[17] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A Learning-to-Rank Based Fault Localization Approach using Likely Invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. New York, NY, USA: ACM, 2016, pp. 177–188.

[18] G. Laghari, A. Murgia, and S. Demeyer, "Fine-Tuning Spectrum Based Fault Localisation with Frequent Method Item Sets," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. New York, NY, USA: ACM, 2016, pp. 274–285.

[19] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 609–620.

[20] 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*, vol. 22, no. 4, pp. 1936–1964, 2017.

[21] X.-B. D. Le, D. Lo, and C. Le Goues, "History Driven Program Repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, March 2016, pp. 213–224.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 191–200.

[23] S. Zhang and C. Zhang, "Software Bug Localization with Markov Logic," in *Proceedings of the 36th International Conference on Software Engineering (ICSE Companion '14)*. New York, NY, USA: ACM, 2014, pp. 424–427.

[24] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, pp. 77–101, 2008.

[25] M. Martinez and M. Monperrus, "Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing," *Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.

[26] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective Object-Oriented Program Repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 648–659.

[27] Q. Xin and S. P. Reiss, "Leveraging Syntax-Related Code for Automated Program Repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 660–670.

[28] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[29] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. New York, NY, USA: ACM, 2015, pp. 166–178.

[30] H. Yokoyama, Y. Higo, and S. Kusumoto, "Evaluating Automated Program Repair Using Characteristics of Defects," in *Proceedings of the 8th International Workshop on Empirical Software Engineering in Practice (IWESEP '17)*, March 2017, pp. 47–52.

[31] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Software Engineering (EMSE)*, 2018.

[32] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for Evaluating Bug Detection Tools," in *Workshop on the Evaluation of Software Defect Detection Tools (co-located with PLDI '05)*, 2005.

[33] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Jun. 2009.

[34] 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 *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. New York, NY, USA: ACM, 2016, pp. 512–515.