Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts

Thomas Durieux, Fernanda Madeiral, Matias Martinez, Rui Abreu

Empirical Review of Java Program Repair Tools

A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts

Thomas Durieux
INESC-ID and IST, University of Lisbon, Portugal
thomas@durieux.me

Fernanda Madeiral
Federal University of Uberlândia, Brazil
fer.madeiral@gmail.com

Matias Martinez
University of Valenciennes, France
matomartinez@gmail.com

Rui Abreu
INESC-ID and IST, University of Lisbon, Portugal
rui@computer.org

ABSTRACT

In the past decade, research on test-suite-based automatic program repair has grown significantly. Each year, new approaches and implementations are featured in major software engineering venues. However, most of those approaches are evaluated on a single benchmark of bugs, which are also rarely reproduced by other researchers. In this paper, we present a large-scale experiment using 11 Java test-suite-based repair tools and 5 benchmarks of bugs. Our goal is to have a better understanding of the current state of automatic program repair tools on a large diversity of benchmarks. Our investigation is guided by the hypothesis that the repairability of repair tools might not be generalized across different benchmarks of bugs. We found that the 11 tools 1) are able to generate patches for 21% of the bugs from the 5 benchmarks, and 2) have better performance on Defects4J compared to other benchmarks, by generating patches for 47% of the bugs from Defects4J compared to 10-30% of bugs from the other benchmarks. Our experiment comprises 23,551 repair attempts in total, which we used to find the causes of non-patch generation. These causes are reported in this paper, which can help repair tool designers to improve their techniques and tools.

ACM Reference Format:
Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software bugs decrease the quality of software systems from the point of view of the software system users. Manually repairing bugs is well-known as being a difficult and time-consuming task. To address this activity automatically, a new field of research has emerged, named *automatic program repair*. Automatic program repair consists of automatically finding solutions 100% executable to software bugs, without human intervention [30, 31]. The most popular approach to automatically repair bugs is to create a patch using the test suite of the program as the specification of its expected behavior. This type of approach is known as *test-suite-based program repair* [30], which has been applied in several repair tools in the last decade [2, 6, 10, 13, 14, 17, 18, 23, 24, 27, 28, 37, 42, 43, 45–47, 50].

The repair tools are used in *empirical evaluations* so that the repairability of the repair approaches they implement is measured. These evaluations are reported in the literature in two ways: when a new repair approach is proposed (e.g. [50]), or when a dedicated full contribution on evaluating existing repair tools is reported (e.g. [26, 33, 48]). The evaluations consist of four main aspects in general: 1) [benchmark] the selection of benchmarks of bugs; 2) [execution] the collection of data by executing repair tools on the selected bugs; 3) [observed aspect] an investigation on the effectiveness of the repair approach regarding some criteria (e.g. *repairability*, *correctness*, and *repair time*); and finally 4) [comparison/discussion] the comparison of repair approaches and discussion.

A major problem with all previous evaluations, focusing on repair for Java programs, is that they are widely performed on the same benchmark of bugs: Defects4J [16]. In theory, this should not be a problem if Defects4J is not biased; however, no benchmark is perfect [21]. Benchmarks should reflect the representativeness of the bugs and the projects they come from in the real world. The extent of the representativeness of benchmarks for real-world bugs is unknown, because even the distribution of the real world bugs is unknown. Therefore, by using a single benchmark when evaluating repair tools, a bias can be introduced, which makes hard to generalize the performance of repair tools.

In this paper, we report on a large experiment conducted on 11 test-suite-based repair tools for Java using other benchmarks of bugs than Defects4J. The primary goal of this experiment is to investigate if the existing repair tools behave in a similar way across different benchmarks. If a repair tool performs significantly better on one benchmark than on others, we say that the repair tool *overfits the benchmark*. The secondary goal is to understand the causes of non-patch generation from a practical view, which, to the best of our knowledge, has not been subject of investigation by the repair community.

To achieve our goals, we designed our experiment considering three out of the four main aspects usually used to evaluate repair tools: a) on benchmark, we use 5 benchmarks (including Defects4J), totaling 2,141 bugs; b) on execution, we run 11 repair tools on the 2,141 bugs, using a framework we developed to automatize and

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

simplify the execution of repair tools on different benchmarks; c) on observed aspect, we analyze the repairability of the tools, focusing on their performance across the different benchmarks. We do not target the fourth aspect of evaluations, which is about comparing repair tools. Our goal is not to compare repair tools among themselves, but to compare the behavior of each tool among different benchmarks.

Through our experiment, we first observed that all 11 repair tools are able to generate a test-suite adequate patch for bugs from each of the 5 benchmarks. However, when analyzing the proportion of bugs patched by the repair tools per benchmark, we found that indeed the repair tools perform better on Defects4J than on the other benchmarks. Finally, we found six main reasons why repair tools do not succeed to generate patches for bugs. For instance, we observed that incorrect fault localization and multiple fault locations have a significant impact on patch generation. These reasons are valuable for repair tool designers and researchers to improve their tools.

To sum up, our contributions are:

- A large-scale experiment of 11 repair tools on 2,141 bugs from 5 benchmarks: this is the largest study on automatic program repair ever (i.e. 11 x 2,141 = 23,551 repair attempts);
- A repair execution framework, named RepairThemAll, that adds an abstraction around repair tools and benchmarks, which can be further extended to support additional repair tools and benchmarks;
- A novel study on the repairability of repair tools across multiple benchmarks, where the goal is to investigate if there exists the *benchmark overfitting* problem, i.e. that the repair tools perform significantly better on the extensively used benchmark Defects4J than on other benchmarks;
- A thorough study on the non-patch generation cases on the 23,551 repair attempts that did not result in patches.

The remainder of this paper is organized as follows. Section 2 presents the literature on the evaluation of test-suite-based repair tools for Java, which grounds the motivation of our experiment. Section 3 presents the design of our study, including the research questions and the data collection and analysis. Section 4 presents the results, followed by the discussion in Section 5. Finally, Section 6 presents the related works, and Section 7 presents the final remarks.

2 STATE OF AFFAIRS ON TEST-SUITE-BASED AUTOMATIC REPAIR TOOLS FOR JAVA

Automatic repair tools meet benchmarks of bugs when they are evaluated. In this section, we present a review of the literature on the existing evaluations of repair tools, which is based on a two-step protocol: 1) gathering repair tools, and 2) gathering information on their evaluations, focusing on the used benchmarks and the number of bugs given as input to the repair tools.

To gather repair tools, we searched the existing living review on automatic program repair [32] for test-suite-based repair tools for Java, which is the focus of this work: we found 24 repair tools that meet this criterion. Then, we gathered scientific papers containing evaluations of these tools. There are two types of papers that are interesting for us: the first type consists of the presentation of a new repair approach, which also includes an evaluation conducted using a tool that implements the approach (e.g. [47]); and the second type consists of an empirical evaluation carried out on already created tools, which is a specific work to evaluate repair tools by running them on benchmarks of bugs (e.g. [26]). We gathered 18 papers from the first type of papers (more than one tool can be presented in the same paper) and two papers from the second type.

Table 1: Test-suite-based program repair tools for Java.

Repair tool	Benchmark used in evaluation	# Bugs	# Patched[a]	# Fixed[b]
Generate-and-validate				
ACS [46]	Defects4J	224	23	17
ARJA [50]	Defects4J	224	59	18
	QuixBugs [48]	40	4	2
CapGen [42]	Defects4J	224	25	22
	IntroClassJava	297	–	25
Cardumen [28]	Defects4J	356	77	–
	QuixBugs [48]	40	5	3
DeepRepair [43]	Defects4J	374	51	–
Elixir [37]	Defects4J	82	41	26
	Bugs.jar	127	39	22
GenProg-A [50]	Defects4J	224	36	–
HDRepair [18]	Defects4J	90	–	23
Jaid [2]	Defects4J	138	31	25
jGenProg [27]	Defects4J	224	29	–
	Defects4J [26]	224	27	5
	QuixBugs [48]	40	2	0
jKali [27]	Defects4J	224	22	–
	Defects4J [26]	224	22	1
	QuixBugs [48]	40	2	1
jMutRepair [27]	Defects4J	224	17	–
	QuixBugs [48]	40	3	1
Kali-A [50]	Defects4J	224	33	–
LSRepair [23]	Defects4J	395	38	19
PAR [17]	PARDataset	119	27	–
RSRepair-A [50]	Defects4J	224	44	–
	QuixBugs [48]	40	4	2
SimFix [14]	Defects4J	357	56	34
SketchFix [13]	Defects4J	357	26	19
SOFix [24]	Defects4J	224	–	23
ssFix [45]	Defects4J	357	60	20
xPAR [18]	Defects4J	90	–	4
Semantics-driven				
DynaMoth [10]	Defects4J	224	27	–
	QuixBugs [48]	40	2	1
Nopol [47]	ConditionDataset	22	17	13
	Defects4J [26]	224	35	5
	QuixBugs [48]	40	3	1
Metaprogramming-based				
NPEFix [6]	NPEDataset	16	14	–
	QuixBugs [48]	40	2	1

[a] A *patched* bug means that a repair tool fixed it with a test-suite adequate patch.
[b] A *fixed* bug means that a repair tool fixed it with a test-suite adequate patch that was confirmed to be correct.

Table 1 summarizes our review on the existing evaluations of the 24 repair tools based on the 20 scientific papers. The first column presents the repair tools, which are grouped by the well-known categories *generate-and-validate* and *semantics-based* approaches, plus *metaprogramming-based*. We named the latter category for repair tools that first create a metaprogram of the program under repair and then explore it at runtime, which in the end uses the runtime information to generate patches.

Each repair tool is associated with one or more benchmarks used in its evaluation in the table. When a repair tool has been evaluated in more than one benchmark (or more than once in the same benchmark), we place first the benchmark used in the paper that presented the tool (i.e. first evaluation), followed by the other benchmarks with the reference for the posterior studies.

For instance, in the paper that jGenProg [27] is presented, there is an evaluation on Defects4J: this evaluation has no citation in the second column of the table because the evaluation is in jGenProg's paper. Later, it was evaluated again on Defects4J [26] and also on QuixBugs [48], which contain citations of the empirical evaluation papers in the table. The table also presents additional information on the evaluations, which are the number of bugs given as input to the repair tools, and the number of bugs for which the tools generated a test-suite adequate patch (i.e. patched bugs) and a correct patch (i.e. fixed bugs), reported by the gathered papers.

In total, we found 38 evaluations of the 24 repair tools. Out of 24, 22 repair tools were evaluated on (a subset of) bugs from Defects4J, and nine of them were recently evaluated on the QuixBugs benchmark [22]. In some exceptional cases, the benchmarks Bugs.jar [36] and IntroClassJava [11] were also used. However, the number of existing evaluations in terms of number of repair tools versus number of benchmarks is low compared to all possible combinations. There are some benchmarks that were rarely used or never used so far: this is partially explained by the fact that some benchmarks were recently published (e.g. Bears [25]), thus they were not available when some repair tools were published.

We also observe that three repair tools were originally evaluated on datasets that were not presented in the literature in a research paper dedicated for them (i.e. PARDataset [17], ConditionDataset [47] and NPEDataset [6]). This is the case of the first evaluations of PAR, Nopol, and NPEFix. However, these repair tools were later evaluated on formally proposed benchmarks, except for PAR, which is not publicly available. PAR was later reimplemented, resulting in the tool xPAR [18], which was then evaluated on Defects4J.

3 STUDY DESIGN

The extensive usage of Defects4J at evaluating repair tools motivates our study. Our goal is to investigate if the repair tools have a similar performance on other benchmarks of bugs. In this section, we present the design of our study, including the research questions, the systematic selection of repair tools and benchmarks of bugs, and the data collection and analysis.

3.1 Research Questions

RQ1. [Repairability] To what extent do test-suite-based repair tools generate patches for bugs from a diversity of benchmarks?
This research question guides us towards the investigation on the ability of the existing repair tools to generate test-suite adequate patches for bugs from the selected benchmarks.

RQ2. [Benchmark overfitting] Is the repair tools' repairability similar across benchmarks?
The repair tools have been extensively evaluated on Defects4J. Our goal in this research question is to investigate if they repair bugs from other benchmarks to a similar extent than they repair bugs from Defects4J.

RQ3. [Non-patch generation] What are the causes that lead repair attempts to not generate patches?
Existing evaluations focus on the successful cases, i.e. the bugs that a given repair tool generated patches for. However, to the best of our knowledge, there is no study that investigates the unsuccessful cases, i.e. a repair tool tried to repair a bug but no patch was generated. Our goal in this research question is to find the causes of non-patch generation so that the repair community can focus on practical limitations and improve their repair tools.

Table 2: Selected repair tools based on our inclusion criteria.

	Non-fulfillment Criteria	Repair Tools
Excluded (13)	Not public (C1)	Elixir, PAR, SOFix, xPAR
	Not working (C2)	ACS, CapGen, DeepRepair
	Only compatible with Defects4J (C3)	LSRepair, SimFix
	Faulty class/method required (C4)	HDRepair, Jaid, SketchFix
	Others	ssFix
Included (11)	ARJA, Cardumen, DynaMoth, jGenProg, GenProg-A, jKali, Kali-A, jMutRepair, Nopol, NPEFix, RSRepair-A	

3.2 Subject Repair Tools

To include a Java test-suite-based program repair tool in our study, it must meet the following four inclusion criteria:

- *Criterion #1.* The repair tool ought to be publicly available: our study involves the execution of repair tools, therefore tools that are not publicly available are excluded. We exclude tools with this criterion when 1) the paper where the tool was described does not include a link for the tool, 2) we cannot find the tool on the internet, and 3) we received an answer by email from the authors of the tool explaining why the tool is not available (e.g. Elixir has a confidentiality issue) or no answer at all.
- *Criterion #2.* The repair tool ought to be possible to run: some tools are publicly available, but they are not possible to run for diverse issues (e.g. ACS uses GitHub, which recently changed its interface and does not allow programmed queries).
- *Criterion #3.* The repair tool ought to be possible to run on bugs from different benchmarks beyond the one used in its original evaluation: we cannot run tools in other benchmarks if they are hardcoded to specific ones (e.g. SimFix is currently working only for Defects4J).
- *Criterion #4.* The repair tool ought to require only the source code of the program under repair and its test suite used as oracle. These two elements are the two inputs specified in the problem statement of test-suite-based automatic program repair [31].

After checking on all 24 repair tools presented in Table 1, we found 12 tools that meet the inclusion criteria outlined. One of them, ssFix, was further excluded because we had issues to run it, so we ended up with 11 repair tools for our experiment. Table 2 presents the excluded and included tools, and for the excluded ones, it also shows the criterion they did not meet. Note that, among the included tools, we have eight generate-and-validate tools, the two semantics-based tools, and the only metaprogramming-based tool. Therefore, we cover the three approach categories. We briefly describe each selected repair tool in the remainder of this section.

jGenProg [27] and GenProg-A [50] are Java implementations of GenProg [41], which is for C programs. GenProg is a redundancy-based repair approach [29] that generates patches using existing code (aka the *ingredient*) from the system under repair, i.e., it does not synthesize new code. GenProg works at the statement level,

and the repair operations are insertion, removal, and replacement of statements.

jKali [27] and Kali-A [50] are Java implementations of Kali [35]. Kali was conceived to show that most of the patches synthesized by GenProg over the ManyBugs benchmark [20] consist of avoiding the execution of code. The operators implemented in Kali are removal of statements, modification of `if` conditions to *true* and *false*, and insertion of `return` statements.

jMutRepair [27] is an implementation of the mutation-based repair approach presented by Debroy and Wong [4] for Java. It considers three kinds of mutation operators: relational (e.g. ==), logical (e.g. &&) and unary (i.e. addition or removal of the negation operator !). jMutRepair performs mutations on those operators in suspicious `if` condition statements.

Nopol [47] is a semantics-based repair tool dedicated to repair buggy `if` conditions and to add missing `if` preconditions. Nopol uses the so-called angelic values to determine the expected behavior of suspicious statements: an angelic value is an arbitrary value that makes all failing test cases from the program under repair pass. Nopol then collects those values at runtime and encodes them into a Satisfiability Modulo Theory (SMT) formula to find an expression that matches the behavior of the angelic value. When the SMT formula is satisfiable, Nopol translates the SMT solution into a source code patch.

DynaMoth [10] is a repair tool integrated into Nopol that also targets buggy and missing `if` conditions. The difference between DynaMoth and Nopol is that instead of using an SMT formula to generate the patch, it uses the Java Debug Interface to access the runtime context and collects variable and method calls. Then, DynaMoth combines those variables and method calls to generate more complex expressions until it finds one that has the expected behavior. This allows the generation of patches containing method calls with parameters, for instance.

NPEFix [6] is different from the generate-and-validate and semantics-based tools, it is a metaprogramming-based tool. It means that NPEFix modifies the program under repair to include several repair strategies that can be activated during the runtime. NPEFix repairs programs that crash due to a null pointer exception. NPEFix runs the failing test-case several times and activates a different repair strategy for each execution. In the end, knowing the repair strategies that have worked, together with information of the context that they worked, a patch is created. Note that NPEFix works in a similar way than semantics-based tools in this last step: if a patch is found, it means the patch is already satisfactory.

ARJA [50] is a genetic programming approach that optimizes the exploration of the search space by combining three different approaches: a patch representation for decoupling properly the search subspaces of likely-buggy locations, operation types and ingredient statements; a multi-objective search optimization for minimizing the weighted failure rate and for searching simpler patches; and a method/variable scope matching for filtering the replacement/inserted code to improve compilation rate.

Cardumen [28] is a test-suite-based repair approach that works at the level of expressions. It synthesizes new expressions (that are used to replace suspicious expressions) as follows. First, it mines templates (i.e., piece of code at the level of expression, where the variables are replaced by placeholders) from the code under repair.

Table 3: Selected benchmarks of bugs and their sizes.

Benchmark	# Projects	# Bugs	LOC (Java)
Bears [25]	72	251	62,597
Bugs.jar [36]	8	1158	212,889
Defects4J [16]	6	395	129,592
IntroClassJava [11]	6	297	230
QuixBugs [22]	40	40	190
Total	130	2,141	146,428

Then, for creating a candidate patch that replaces a suspicious expression *se*, Cardumen selects a compatible template (i.e. the evaluation of the template and the *se* return compatible types) and creates a new expression from it by replacing all its placeholders with variables frequently used in the context of *se*.

RSRepair-A [50] is a Java implementation of RSRepair [34]. RSRepair repairs is a test-suite-based repair approach for C that has been created to compare the performance between genetic programming (GenProg) and random search in the case of automatic program repair. It showed that in 23/24 RSRepair finds patches faster than GenProg.

3.3 Subject Benchmarks of Bugs

To select benchmarks of bugs for our study, we defined the following three inclusion criteria:

- *Criterion #1*: The benchmark must contain bugs in the Java language: this criterion excludes benchmarks such as ManyBugs [21], IntroClass [21], Codeflaws [40] and BugsJS [12].
- *Criterion #2*: The benchmark must be peer-reviewed, presented in the literature in a research paper dedicated for it: this criterion excludes benchmarks such as PARDataset [17], ConditionDataset [47], and NPEDataset [6].
- *Criterion #3*: The benchmark must include, for each bug, at least one failing test case: this criterion excludes benchmarks such as iBugs [3].

After searching the literature for benchmarks that meet our criteria, we ended up with 5 benchmarks. Table 3 summarizes them by presenting their sizes in number of projects, bugs and lines of code. We present a brief description of them as follows.

Defects4J [16] contains 395 bugs from six widely used open source Java projects with an average size of 129,592 lines of Java code. The bugs were extracted by the identification of bug fixing commits with the support of the bug tracking system and the execution of tests on the bug fixing program version and its reverse patch (buggy version). Despite the fact this benchmark was first proposed to the software testing community, it has been used for several works on automatic program repair.

Bugs.jar [36] contains 1,158 bugs from eight Apache projects with an average size of 212,889 lines of Java code. It was created using the same strategy than Defects4J. Its main contribution is the high number of bugs.

Bears [25] contains 251 bugs from 72 different GitHub projects with an average size of 62,597 lines of Java code. It was created by mining software repositories based on commit building state from Travis Continuous Integration. Bears has the largest diversity of project compared to previous bug benchmarks.

IntroClassJava [11] contains 297 bugs from six different student projects. It is a transpiled version to Java of the bugs from the C benchmark IntroClass [21]. In the transpiled version, the projects have on average 230 lines of code.

QuixBugs [22] contains 40 single line bugs from 40 programs, which are translated into both Java and Python languages. Each program corresponds to the implementation of one algorithm such as Quicksort and contains on average 190 lines of code. This is the first multi-lingual program repair benchmark.

3.4 Building a Repair Execution Framework

To run the repair tools on different benchmarks of bugs, we created an execution framework named RepairThemAll, which provides an abstraction around repair tools and benchmarks. Figure 1 illustrates the overview of the framework. It is composed of three main components: 1) *repair tool plug-in*, where there is the abstraction around repair tools, allowing the addition and removal of tools, 2) *benchmark plug-in*, where there is the abstraction around benchmarks, also allowing the addition and removal of benchmarks, and 3) *repair runner*, which works as a façade for the execution of repair tools on specific bugs.

For the creation of the framework, we performed three main tasks, one for each component. First, for *repair tool plug-in*, we identify the common parameters that are required by the repair tools, which we refer to as *abstract parameters*. We then map these abstract parameters to the *actual parameters* of each repair tool. This is necessary because the repair tools use different parameter names and input formats. For instance, an abstract parameter is the source code folder path, and the actual parameter for source code folder path in ARJA is `DsrcJavaDir`, but in jGenProg is `srcjavafolder`. We identify eight common parameters for the repair tools: (1) source code path, (2) test path, (3) binary path of the source code, (4) binary path of the tests, (5) the classpath, (6) the java version (compliance level), (7) the failing test class name, and (8) workspace directory. RepairThemAll also supports the setting of actual parameters existing in the repair tools to tune specific executions, i.e., actual parameters that are not mapped to any abstract parameter.

Then, for *benchmark plug-in*, we identify the *abstract operations* that should be performed to use the bugs from benchmarks (e.g. check out a buggy program given a bug id). We map these abstract operations to the *actual operations* of each benchmark when they are available. We define three required bug usage operations to be able to use the benchmark with repair tools: (1) to check out a specific bug (buggy source code files) at a given location, (2) to compile the buggy source code and the tests, and (3) to provide information on the bug to be given as input to repair tools (i.e. the eight parameters previously mentioned, except workspace directory). If the bug is from a multi-module project, the source code and test paths should be related to the module that contains the bug. Only Defects4J provides bug usage operations that fully covers the required abstract operations. Consequently, we had to build above the four other benchmarks the missing operations, e.g. to check out a bug from the QuixBugs benchmark.

Finally, for the *repair runner*, we design the input and output in a simple way so that one can easily interact with the RepairThemAll framework as well as interpret the results. For the input,

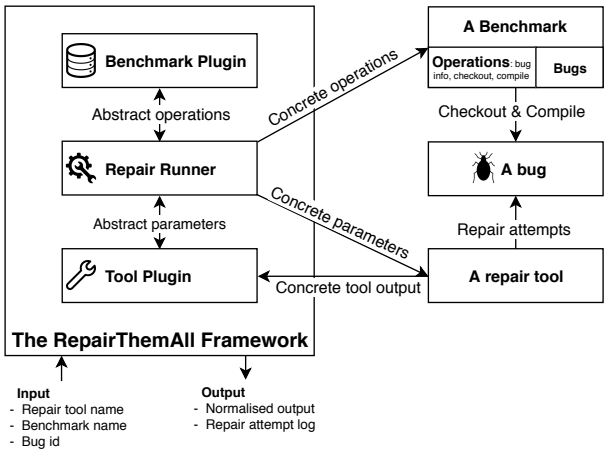

Figure 1: The RepairThemAll framework.

one can start an execution of repair tools on benchmarks as a simple command line: for instance, the command `./repair.py Nopol -benchmark Defects4J -id Chart-7` starts the execution of Nopol on the bug Chart 7 of Defects4J. At the end of this execution, the repair runner generates a standardized output divided into two files: the log of the repair attempt execution (`repair.log`), and the normalized JSON file (`results.json`) containing the location of the patches generated by the tools, if any, and the textual difference between the patches and their buggy program versions. This standardized output is due to the abstraction around the output format (output normalization) we create to simplify the analysis and the readability of the results from the different repair tools.

The RepairThemAll framework currently contains 11 repair tools and 5 benchmarks of bugs, and it allows the plug-in of other ones to help the repair community to compare different approaches. Moreover, the framework allows users to set repair tool executions such as the timeout and the limit of generated patches. It is publicly available at [7], which includes a tutorial with all the steps to use it and to integrate new repair tools and new benchmarks.

3.5 Data Collection and Analysis

To answer our research questions, we executed the 11 repair tools on the 5 benchmarks using RepairThemAll, resulting in patches that are further used for analysis. In this section, we describe the repair tools' setup (Section 3.5.1) and their execution (Section 3.5.2), and the analysis we performed on the repair attempts to determine the possible causes of non-patch generation (Section 3.5.3).

3.5.1 Repair tools' setup. For this experiment, we set the time budget to two hours per repair attempt: a repair attempt consists of the execution of one repair tool over one bug. We also configured the repair tools to stop the execution of repair attempts when they already generated one patch. However, ARJA, GenProg-A, Kali-A, RSRepair-A, and NPEFix do not have this option, they stop their repair attempts when they consume their own tentative budget, or by timeout. Moreover, we configured repair tools to run on one predefined random seed: due to the huge computational power required for this experiment, we were not able to run the repair tools with additional seeds. Finally, Table 4 presents the version of each repair tool that we used in this study. The logs of the repair attempts are available at [8].

Table 4: The used version of each repair tool.

Repair tool	Framework	GitHub repository	Commit id
ARJA, GenProg-A, Kali-A, RSRepair-A	Arja	yyxhdy/arja	e60b990f9
Cardumen, jGenProg, jKali, jMutRepair	Astor	SpoonLabs/astor	26ee3dfc8
DynaMoth, Nopol	Nopol	SpoonLabs/nopol	7ba58a78d
NPEFix	–	Spirals-Team/npefix	403445b9a

3.5.2 Large scale execution. To our knowledge, our experimental setup is the biggest one on patch generation studies, in terms of number of repair tools and bugs, and also in execution time. In total, we executed 11 repair tools on 2,141 bugs from 130 open-source projects that the selected 5 benchmarks provide. This represents 23,551 repair attempts, which took 314 days and 12.5 hours of combined execution, almost a year of continuous execution. This experiment would not be possible without the support of the cluster Grid'5000 [1] that provided us the required computing power to conduct this work.

3.5.3 Finding causes of non-patch generation. Prior studies on patch generation mainly focused on the ability of approaches to generate patches, and do not investigate the reasons why non-patch generation happens. The study on non-patch generation is important to make research progress so that authors of repair tools can improve their tools. Since there is a lack of knowledge on that subject, we are not able to automatically perform the detection of the reasons why patches are not generated for bugs, and therefore manual analysis is required. Due to the scale of our experiment setup including 23,551 different patch generation attempts, it is unrealistic to manually analyze each attempt log to understand what happened. We identify the major causes of non-patch generation by analyzing a sample of the repair attempt logs. We do not predefine the sample because we observe during preliminary investigation that identical behaviors happen for groups of repair attempts. For instance, we found that for all bugs from a specific project, all the repair tools have the same issue in the fault localization. For that reason, predefining a sample is not optimal, because we would analyze attempt logs that we already know what is the cause of non-patch generation.

4 RESULTS

The results of our empirical study, as well as the answers to our research questions, are presented in this section.

4.1 [RQ1] Repairability of the 11 Repair Tools

In this research question, we analyze the repairability of the 11 repair tools on the total of 2,141 bugs. For that, we calculated the number of patched bugs and also the number of bugs that are commonly patched by tool.

Figure 2 presents the repairability of the repair tools in descending order by the number of patched bugs. For each tool, it shows the number of unique bugs patched by the tool (dark grey), the number of patched bugs that other repair tools also patched (light grey), and the total number of patched bugs with the proportion over all 2,141 included in this study. For instance, Nopol synthesizes patches for

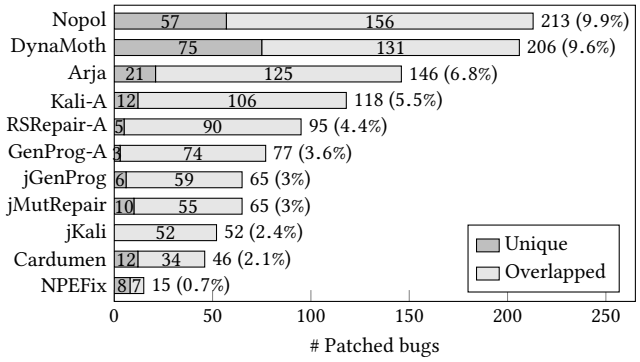

Figure 2: Repairability of the 11 repair tools on 2,141 bugs.[1]

213 bugs in total (9.9% of all bugs), where 57 are uniquely patched by Nopol, and 156 are patched by Nopol and other tools.

We observe that Nopol, DynaMoth, and ARJA are the three tools that generate test-suite adequate patches for the highest number of bugs, with respectively 213, 206 and 146 patched bugs in total. NPEFix, on the other hand, generates patches for the fewest number of bugs (15). It can be explained by the narrow repair scope of this tool, i.e. bugs exposed by null pointer exception.

On bugs uniquely patched by tools, we observe that only jKali failed to generate patches for bugs that are not patched by other tools, and that DynaMoth is the tool that patches the highest number of unique bugs. However, NPEFix is the tool that has the highest proportion of unique patched bugs, i.e. 53% of the 15 bugs patched by NPEFix are unique.

The overlapping between each pair of repair tools in number of bugs is presented in Table 5. In the case where the column name and the line name are the same (main diagonal), it presents the number of uniquely bugs patched by the tool. For instance, 20 bugs have been uniquely patched by ARJA, which repairs other 66 bugs that are also patched by GenProg-A.

We observe a large overlap between repair tools that share the same patch generation framework, i.e. the framework where the repair tools are implemented (see Table 4). For instance, ARJA has an overlap of 45% with GenProg-A, 56% with Kali-A, and 55% with RSRepair-A, all implemented in the Arja framework. However, ARJA has an overlap ranging from 2% to 36% with the other repair tools. DynaMoth has an overlap of 55% with Nopol, but only 0% to 26% with the other tools. Each tool implemented in the Astor framework (e.g. jGenProg) has a big overlap with other tools in Astor. Moreover, the tools in Astor also present high overlapping with the tools in the Arja framework, which are tools sharing similar repair approaches.

Answer to RQ1. To what extent do test-suite-based repair tools generate patches for bugs from a diversity of benchmarks? The 11 repair tools are able to generate patches for bugs ranging from 15 to 213 bugs, from a total of 2,141 bugs. They are complementary to each other, because 10/11 repair tools fix unique bugs (all but jKali). We also observe that the overlapped repairability of the tools is impacted by their similar implemented repair approaches, and also by the patch generation framework where they are implemented.

[1]The full list of the patched bugs and the textual patches are available in [9].

Table 5: The number of overlapped patched bugs per repair tool. Each row r presents the percentage of overlapped patched bugs of one tool t_r with the rest of the tools. For instance, 45% of the bugs patched by ARJA (row 2) are also patched by GenProg-A (column 3). On the contrary, 85% of the bugs patched by GenProg-A (row 3) are also patched by ARJA (column 2).

	ARJA	GenProg-A	Kali-A	RSRepair-A	Cardumen	jGenProg	jKali	jMutRepair	Nopol	DynaMoth	NPEFix
ARJA	**13% (20)**	45% (66)	56% (82)	55% (81)	15% (23)	30% (44)	27% (40)	19% (29)	36% (53)	32% (48)	2% (4)
GenProg-A	85% (66)	**3% (3)**	63% (49)	81% (63)	22% (17)	40% (31)	37% (29)	23% (18)	42% (33)	40% (31)	2% (2)
Kali-A	69% (82)	41% (49)	**9% (11)**	46% (55)	16% (20)	28% (34)	37% (44)	23% (28)	47% (56)	45% (54)	1% (2)
RSRepair-A	85% (81)	66% (63)	57% (55)	**5% (5)**	17% (17)	38% (37)	31% (30)	21% (20)	37% (36)	36% (35)	2% (2)
Cardumen	50% (23)	36% (17)	43% (20)	36% (17)	**26% (12)**	65% (30)	45% (21)	32% (15)	21% (10)	26% (12)	4% (2)
jGenProg	67% (44)	47% (31)	52% (34)	56% (37)	46% (30)	**9% (6)**	55% (36)	41% (27)	29% (19)	36% (24)	3% (2)
jKali	76% (40)	55% (29)	84% (44)	57% (30)	40% (21)	69% (36)	**0% (0)**	57% (30)	53% (28)	67% (35)	1% (1)
jMutRepair	44% (29)	27% (18)	43% (28)	30% (20)	23% (15)	41% (27)	46% (30)	**15% (10)**	58% (38)	30% (20)	1% (1)
Nopol	24% (53)	15% (33)	26% (56)	16% (36)	4% (10)	8% (19)	13% (28)	17% (38)	**26% (57)**	53% (114)	<1% (2)
DynaMoth	23% (48)	15% (31)	26% (54)	16% (35)	5% (12)	11% (24)	16% (35)	9% (20)	55% (114)	**36% (75)**	<1% (1)
NPEFix	26% (4)	13% (2)	13% (2)	13% (2)	13% (2)	13% (2)	6% (1)	6% (1)	13% (2)	6% (1)	**53% (8)**

4.2 [RQ2] Benchmark Overfitting

In this research question, we compare the repairability of the repair tools on the bugs from the extensively used benchmark Defects4J with their repairability on the other benchmarks included in this study, which are Bears, Bugs.jar, IntroClassJava, and QuixBugs.

Table 6 shows the number of bugs that have been patched by each repair tool per benchmark. We first observe that Defects4J is the benchmark with the highest number of unique patched bugs (187), which represents 47.34% of all Defects4J bugs. The next most patched benchmarks are QuixBugs with 30% and IntroClassJava with 20.87% of their bugs. This difference can also be observed in the total number of generated patches per benchmark: Defects4J is still dominating the ranking with 550 generated patches, even that it contains fewer bugs than Bugs.jar (395 versus 1,158 bugs).

To test if the repairability of the repair tools is independent of Defects4J, we applied the Chi-square test on the number of patched bugs for Defects4J compared to the other benchmarks. The null hypothesis of our test is that *the number of patched bugs by a given tool is independent of Defects4J*. We observed in Table 6 that the *p-value* is smaller than the significance level $\alpha < .05$ for all repair tools. Hence, we reject the null hypothesis for those 11 tools, and we conclude that the number of patched bugs by them is dependent of Defects4J. Therefore, *repair tools overfit Defects4J*.

The repairability of the repair tools on Defects4J cannot be only explained by the repair approaches. We raised three hypotheses that can potentially explain the repairability difference between Defects4J and the other benchmarks: 1) there is a technical problem in the repair tools, 2) the bug fix isolation performed on Defects4J has an impact on repairing Defects4J bugs, and 3) the distribution of the bug types in Defects4J is different from the other benchmarks.

1. [Technical problems in the repair tools] In RQ1, we observed the importance of the implementation of the tools for the repairability. One hypothesis that can explain the fact that repair tools overfit Defects4J is that the authors of the repair tools have debugged and tuned their frameworks for Defects4J and, consequently, improved significantly the repairability of their tools for this specific benchmark. For instance, they may have paid attention to not let the dependencies of the repair tools to interfere with the classpath of the Defects4J bugs, in order to preserve the behavior of test executions on the Defects4J bugs. However, this issue can affect the bugs of other benchmarks.

2. [Bug fix isolation performed on Defects4J] The second hypothesis is related to the way that Defects4J has been created. A bug fixing commit might include other changes that are not related to the actual bug fix. Then given a bug fixing commit, the authors of Defects4J recreated the buggy and patched program versions so that the diff between the two versions contains only changes related to the bug fix: this is called bug fix isolation. The resulted isolated bug fixes facilitate studies on patches [39]. However, such a procedure can potentially have an impact on the repairability of the repair tools. For instance, by comparing the developer patch [15] with the Defects4J patch [5] for the bug Closure-51, we observe that the method `isNegativeZero` has been introduced in the buggy program version, which contains part of the logic for fixing the bug. The presence of this method in the buggy program version can simplify the generation of patches by the repair tools or introduce an ingredient for genetic programming repair approaches.

3. [Bug type distribution in the benchmarks] Our final hypothesis is related to the distribution of the bugs in the different benchmarks. Defects4J might contain more bugs that can be patched by the repair tools compared to the other benchmarks. For that reason, the bug type distribution of each benchmark should be further analyzed and correlated with the repairability of the tools.

To our understanding, the first hypothesis is more plausible since we observe in RQ1 that the implementation of the repair tools has an impact on their repairability. However, additional studies should be designed to identify which hypothesis, or a combination of hypotheses, has an impact on the repairability of the repair tools on Defects4J compared to other benchmarks.

Answer to RQ2. Is the repair tools' repairability similar across benchmarks? There is a difference in the repairability of the 11 repair tools across benchmarks. Indeed, the repairability of all tools is significantly higher for bugs from Defects4J compared to the other four benchmarks, therefore we conclude that they overfit Defects4J. In addition, we raised three hypotheses that might explain this difference. The confirmation of those hypotheses are full contributions themselves, therefore our study opens the opportunity for several future investigations.

Table 6: Number of bugs patched by at least one test-adequate patch and the p-value of the Chi-square test of independence between the number of patched bugs from Defects4J compared to the other benchmarks.

Repair tool \ Benchmark	Bears (251)	Bugs.jar (1,158)	Defects4J (395)	IntroClassJava (297)	QuixBugs (40)	Total (2,141)	p-value
ARJA	12 (4%)	21 (1%)	86 (21%)	23 (7%)	4 (10%)	146 (7%)	< 0.00001
GenProg-A	1 (<1%)	9 (<1%)	45 (11%)	18 (6%)	4 (10%)	77 (3%)	< 0.00001
Kali-A	15 (5%)	24 (2%)	72 (18%)	5 (1%)	2 (5%)	118 (5%)	< 0.00001
RSRepair-A	1 (<1%)	6 (<1%)	62 (15%)	22 (7%)	4 (10%)	95 (4%)	< 0.00001
Cardumen	13 (5%)	12 (1%)	17 (4%)	0 (0%)	4 (10%)	46 (2%)	0.00107
jGenProg	13 (5%)	14 (1%)	31 (7%)	4 (1%)	3 (7%)	65 (3%)	< 0.00001
jKali	10 (3%)	8 (<1%)	27 (6%)	5 (1%)	2 (5%)	52 (2%)	< 0.00001
jMutRepair	7 (2%)	11 (<1%)	20 (5%)	24 (8%)	3 (7%)	65 (3%)	0.009309
Nopol	1 (<1%)	72 (6%)	107 (27%)	32 (10%)	1 (2%)	213 (10%)	< 0.00001
DynaMoth	0 (0%)	124 (10%)	74 (18%)	6 (2%)	2 (5%)	206 (10%)	< 0.00001
NPEFix	1 (<1%)	3 (<1%)	9 (2%)	0 (0%)	2 (5%)	15 (<1%)	< 0.00001
Total	74	304	550	139	31	1,098	
Total unique	25 (9.96%)	173 (14.93%)	187 (47.34%)	62 (20.87%)	12 (30%)	459 (21.44%)	

Table 7: Percentage of repair attempts that failed by error.

Repair tool \ Benchmark	Bears	Bugs.jar	Defects4J	IntroClassJava	QuixBugs	Average
ARJA	24.70	49.56	1.26	0	0	29.93
GenProg-A	88.04	78.06	7.08	0	2.5	53.90
Kali-A	24.70	50.08	4.81	0	0	30.87
RSRepair-A	87.25	79.27	6.83	0	2.5	54.41
Cardumen	47.41	70.46	48.60	0	5.0	52.73
jGenProg	45.01	63.29	12.65	0	5.0	41.94
jKali	44.62	64.42	12.40	0	5.0	42.45
jMutRepair	72.11	66.66	15.44	13.46	22.5	49.64
Nopol	28.68	60.27	45.31	0	2.5	44.37
DynaMoth	27.09	46.97	4.30	0	0	29.37
NPEFix	89.24	86.18	73.16	0	2.5	70.62
Average	52.62	65.02	21.08	1.22	4.31	45.48

Table 8: Percentage of repair attempts that failed by timeout.

Repair tool \ Benchmark	Bears	Bugs.jar	Defects4J	IntroClassJava	QuixBugs	Average
ARJA	19.52	18.56	6.07	0	0	13.45
GenProg-A	6.37	7.08	9.62	0	5.0	6.44
Kali-A	1.19	2.76	0	0	0	1.63
RSRepair-A	7.17	6.99	8.86	0	5.0	6.35
Cardumen	4.38	61.57	19.74	0	2.5	37.50
jGenProg	48.20	28.23	71.39	83.83	85.0	47.31
jKali	0.79	4.05	1.77	0	0	2.61
jMutRepair	0.39	3.45	1.01	0	0	2.10
Nopol	0.39	0.51	0	0	0	0.32
DynaMoth	0	0.69	2.27	0	0	0.79
NPEFix	0	4.49	0.75	0	0	2.56
Average	8.04	12.58	11.04	7.62	8.86	11.01

4.3 [RQ3] Causes of Non-patch Generation

In this final research question, we analyze the repair attempts that did not result in patches, and we identify the causes of non-patch generation. The goal of this research question is to provide highlights to the automatic repair community on the causes of non-patch generation so that authors of repair tools can improve their tools.

Table 7 and Table 8 present the percentage of repair attempts that finished due to an error and by timeout, respectively. They show that the repair attempts in error or timeout represent the majority of all repair attempts (56.49%). The Bugs.jar benchmark is the main contributor to this percentage. The size and complexity of the Bugs.jar projects show the limitation of the current automatic patch generation tools. Moreover, Table 7 shows that NPEFix is the tool with the highest error rate, but this tool crashes when no null pointer exception is found in the execution of the failing test case that exposes a bug. Regarding the timeout in Table 8, jGenProg and Cardumen are more subject to reach timeout.

We then manually analyzed the execution trace of the repair attempts [8] to identify the causes of non-patch generation. The methodology for this analysis is described in Section 3.5.3, and by following it we identified six causes of non-patch generation.

1. [The repair tool cannot repair the bug] A logical problem is that the repair tools do not have a patch that fixes the bug in their search space. For instance, NPEFix is not able to generate patches for bugs that are not related to null pointer exception. jGenProg is not able to generate a patch when the repair ingredient is not in the source code of the application, which happens frequently for small programs like the ones in QuixBugs. New repair approaches should be created to handle this cause of non-patch generation.

2. [Incorrect fault localization] When the fault localization does not succeed to identify the location of the bug, the repair tools do not succeed to generate a patch for it. This can be due to a limitation of the fault localization approach or to the suspiciousness threshold that the repair tools use. Moreover, we identified that test cases that should pass are failing, and consequently there is a misleading

fault localization. For instance, the fault localization fails on all bugs from the INRIA/spoon project (from the Bears benchmark) because the fault localization does not succeed to load a test resource, and consequently the passing test cases fail.

3. [Multiple fault locations] Developers frequently fix a bug at more than one location in the source code: we refer to this type of bug as multi-location bug. However, most current repair tools and fault localization tools do not support multi-location bugs. For instance, the bug Math-1 from Defects4J has to be fixed in the exact same way at two different locations, and the two locations are specified by two failing test cases. The current tools consider that the two failing test cases specify the same bug at the same location, and consequently do not succeed to generate a multi-location patch.

4. [Too small time budget] We observe that some of the repair attempts finish the execution by consuming all the time budget. Considering the size of this experiment, it is not realistic to increase drastically the time budget. However, new approaches and optimizations can minimize this problem. In this study, we detected 2,593 repair attempts that failed by timeout. It is not possible to predict the outcome of those attempts, but a previous study [28] showed that additional time budget might result in a higher number of generated patches by genetic programming approaches. However, in our experiment, the repair tools require 13.5 minutes on average to generate a patch, which is significantly lower than the allocated time budget (two hours).

5. [Incorrect configuration] We also observe that the RepairThemAll framework does not succeed to correctly compute parameters from some bugs to give as input to the repair tools, such as compliance level, source folders, and failing test cases. This results in failing repair attempts, which can be due to a bug in RepairThemAll or an impossibility to compile the bug. For instance, NPEFix fails 215 times because of issues related to classpath.[2]

6. [Other technical issues] The final cause of non-patch generation is related to other technical limitations that cause the non-execution of the repair tools. One of them is about too long command lines. The repair tools are executed from the command line, which means that all parameters must be provided in the command line. However, the size of the command line is limited, and in the case of projects that have a long classpath, the operating system denies the execution of the command line, which results in failing repair attempts. On Bugs.jar, for instance, 200 repair attempts finished with the error `[Errno 7] Argument list too long`.[3] Finally, there are also other diverse issues that cause the repair tools to crash. For instance, jGenProg finished its repair attempt on the bug Flink-6bc6dbec from Bugs.jar with a `NullPointerException`.[4]

Answer to RQ3. What are the causes that lead repair attempts to not generate patches? Through an analysis on logs of repair attempts, we identified six causes of non-patch generation, such as incorrect fault localization. Each cause should be investigated in detail in new studies. Moreover, repair tools' designers are also stakeholders on those causes, which inform them what are the weakness of their tools and help them to understand their previous evaluations' results.

[2] All the occurrences of InvalidClassPathException in our execution: https://git.io/fjRax
[3] Repair attempts that end with `Argument list too long`: https://git.io/fjRap
[4] Log file of the repair attempt: https://git.io/fjRab

5 DISCUSSION

Diversity of program repair benchmarks. In RQ2, we found that all 11 Java repair tools included in this study perform significantly better on the bugs from Defects4J than on the bugs from other benchmarks. Indeed, repair tool evaluations that only use Defects4J have a threat to the external validity since the repairability results cannot be generalize for other benchmarks. We then conclude that future tools should be evaluated on diverse benchmarks to mitigate that threat.

Impact of the repair tools' engineering on the repairability. During the conduction of this study, we observed that the implementations of the repair approaches play an important role in their ability to repair bugs. For instance, jKali and Kali-A share the same approach, but they neither have the same implementation nor the same results (see Table 6). Kali-A fixes 118 bugs while jKali fixes 52 with the same input. Note that this observation has also been correlated with the analysis of non-patch generation, where a significant number of causes is not related to the repair approaches themselves, but to their implementations. This observation highlights a potential bias in empirical studies on automatic program repair that compare the repairability of different repair approaches. Based on this observation, those studies only compare the effectiveness of repair tools, not the approaches themselves.

Challenges of creating RepairThemAll. The main challenges we faced to run repair tools are related to the creation of the RepairThemAll framework. First, we checked all test-suite-based repair tools for Java based on our criteria (e.g. availability) so that we could find the suitable tools for our study (see Table 2). Then, we had to understand the repair tools we finally gathered, where manual source code analysis was required so that we could compile them and find their inputs and requirements: the tools are diverse, sometimes not documented, and implemented by different researchers. Once we understood the repair tools, we could plug them in the RepairThemAll framework, which contains the abstraction around the tools. Those challenges are mainly due to lack of well-organized open-science repositories for all the repair tools. Good documentation, examples, and instructions on how to compile the tools can speed up the process of learning on how to execute repair tools.

The observed repairability compared to the previous evaluations. Table 1 shows the test-suite-based repair tools for Java and the repairability results from their previous evaluations. Those results are difficult to compare with the results of our study, because the previous evaluations on Defects4J did not consider all bugs from the benchmark. On Defects4J, only Cardumen fixes fewer bugs in this study compared to the previous evaluation. This can be explained by the difference of the setup (such as the number of random seed considered in the study), and potential bugs in the version of Cardumen we use. According to those results, RepairThemAll configures correctly the repair tools to generate patches since no major drawbacks have been observed.

Threats to validity. As with any implementation, the RepairThemAll framework is not free of bugs. A bug in the framework might impact the results we reported in Section 4. However, the framework and the raw data are publicly available for other researchers and potential users to check the validity of the results.

This study focuses on test-suite adequate patches, which means that the generated patches make the test suite pass; yet, there is no guarantee that they fix the bugs. Studying patch correctness [19, 44, 49] is out of the scope of this work. Our goal is to analyze the current state of the automatic program repair tools and identify potential flaws and improvements. The conclusions of our study do not require the knowledge on the correctness of the patches.

Our goal is to have a full picture of test-suite-based repair tools for Java. In our literature review, presented in Section 2, we found 24 repair tools that compose the full picture. Our study was conducted considering only 11 of them: note that this is the largest experiment in terms of number of repair tools (and benchmarks). However, we do not have the full picture we wanted to, which is a threat to the external validity of our results. Most of the repair tools that we did not include in our study are just not possible to be ran: for instance, PAR [17] is not even available. Open-source tools allow the community to generate knowledge in several directions. In our work, open-source tools allowed us to perform a novel evaluation on the state of the repair tools. Another direction is to help the development of new tools: for instance, DeepRepair [43] is built over Astor [27], which is a library for repairing.

6 RELATED WORKS

The works related to ours are empirical studies on the *repairability* of multiple automatic program repair tools. Repair tools for C programs were the subject of investigation in the first empirical studies on automatic program repair. Qi et al. [35] introduced the idea of *plausible* (i.e. *test-suite adequate*) versus *correct* patch. They studied the patch plausibility and correctness of four generate-and-validation repair tools on the bugs from the GenProg benchmark [20] (which later became a part of the ManyBugs benchmark [21]). They found that a small number of bugs are fixed by correct patches.

An *incorrect*, test-suite adequate patch is known as an *overfitting patch*, because it overfits the test suite. Such problem was named as *the overfitting problem* by Smith et al. [38], who studied it in the context of two generate-and-validate C repair tools on the bugs from the IntroClass benchmark [21]. They found that even using high-quality, high-coverage test suites results in overfitting patches. Later, Le et al. [19] analyzed the overfitting problem for semantics-based repair techniques for the C language. The study also investigates how test suite size and provenance, number of failing tests, and semantics-specific tool settings can affect overfitting.

For Java, Martinez et al. [26] reported on a remarkable large experiment, where three repair tools were executed on the bugs from Defects4J. The focus of their study was to measure the repairability of the repair tools and to find correct patches by manual analysis. They also found that a small number of bugs (9/47) could be repaired with a test-suite adequate patch that is also correct. Ye et al. [48] presented a study where nine repair tools were executed on the bugs from QuixBugs. They used automatically generated test cases based on the human-written patches to identify incorrect patches generated by the repair tools.

Motwani et al. [33] reported on an empirical study that included seven repair tools for both Java and C languages, where the Defects4J and ManyBugs benchmarks were used. They had a different focus: they investigated if the bugs repaired by repair tools are hard and important. To do so, they used the repairability data from previous works, and they performed a correlation analysis between the repaired bugs and measures of defect importance, the human patch complexity, and the quality of the test suite.

Table 9: Empirical studies on repair tools.

Work	Language	# Tools	# Bench	Main focus
[35]	C	4	1	plausible vs. correct patch
[38]	C	2	1	patch overfitting
[19]	C	4	2	patch overfitting
[26]	Java	3	1	patch overfitting
[48]	Java	9	1	patch overfitting
[33]	Java + C	7	2	repairability vs. bug-related measures
Ours	Java	11	5	benchmark overfitting+non-patch generation

Table 9 summarizes the mentioned studies. The main difference between our study and all the previous ones is the goal. Previous works focused on the *patch overfitting* problem and advanced/correlation analysis between the repairability of tools and bug-related characteristics. We introduce the *benchmark overfitting* problem, which is investigated in this paper as well as the causes of non-patch generation. Moreover, the scale of our study is much larger than previous studies, on repair tools and benchmarks.

7 CONCLUSIONS

In this paper, we presented an empirical study including 11 repair tools and 2,141 bugs from 5 benchmarks. In total, 23,551 repair attempts were performed: this is the largest experiment to our knowledge. The goal of our experiment is to obtain an overview of the current state of the repair tools for Java in practice. For that, we scaled up the previous experiments by considering more benchmarks of bugs, which combined have bugs from 130 projects, collected with different strategies.

We found that the repair tools are able to repair bugs from benchmarks that were not initially used for their evaluations. However, our results suggest that all repair tools overfit Defects4J. Finally, we analyzed why the repair tools do not succeed to generate patches: this study resulted in six different causes that can help future development of repair tools.

Our study opens several opportunities for future investigations. First, our hypotheses on why the repair tools perform better on Defects4J can be further confirmed. For instance, one of the hypotheses is the fact that the buggy program versions were changed in Defects4J due to the bug fix isolation. A study to confirm this hypothesis is a full contribution itself. Second, other repair tools can also be executed to aggregate and scale up our study. ssFix, for instance, is possible to run, despite the fact we had issues for it, which lead to its exclusion in this work. Moreover, the tools that are hardcoded to be ran on Defects4J could also be adapted to work for other benchmarks of bugs. Finally, an investigation on the bug type distribution in the benchmarks should also be conducted. This would provide the information on how many bugs a repair tool is actually able to fix, i.e. by finding the bugs that meet the repair tools' bug class target.

ACKNOWLEDGMENTS

We acknowledge CAPES for partially funding this research, and Marcelo Maia for discussions. This material is based upon work supported by Fundação para a Ciência e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017 and UID/CEC/50021/2019.

REFERENCES

[1] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer International Publishing, Cham, 3–20.

[2] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-Based Program Repair without the Contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Piscataway, NJ, USA, 637–647.

[3] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 433–436.

[4] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, Washington, DC, USA, 65–74.

[5] Defects4J. 2011. Defects4J patch for Closure-51 bug. http://program-repair.org/defects4j-dissection/#!/bug/Closure/51.

[6] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. 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)*. IEEE, Klagenfurt, Austria, 349–358.

[7] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. The RepairThemAll framework repository. https://github.com/program-repair/RepairThemAll.

[8] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. The repair attempts' results. https://github.com/program-repair/RepairThemAll_experiment.

[9] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Website for browsing the generated patches. http://program-repair.org/RepairThemAll_experiment.

[10] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. ACM, New York, NY, USA, 85–91.

[11] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* Technical Report #hal-01272126. University of Lille, University of Lille.

[12] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: A Benchmark of JavaScript Bugs. In *Proceedings of the 12th International Conference on Software Testing, Verification, and Validation (ICST '19)*. IEEE Computer Society, Washington, DC, USA, 1–12.

[13] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-Demand Candidate Generation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 12–23.

[14] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM, New York, NY, USA, 298–309.

[15] johnlenz. 2011. Human patch for Defects4J Closure-51 bug. https://github.com/google/closure-compiler/commit/a02241e5df48e44e23dc0e66dbef3fdc3c91eb3e.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, New York, NY, USA, 437–440.

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811.

[18] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '16)*. IEEE, Suita, Japan, 213–224.

[19] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 163–163.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13.

[21] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256.

[22] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. ACM, New York, NY, USA, 55–56.

[23] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC '18)*. IEEE Computer Society, Washington, DC, USA, 1–5.

[24] Xuliang Liu and Hao Zhong. 2018. Mining StackOverflow for Program Repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, Campobasso, Italy, 118–129.

[25] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. IEEE, Hangzhou, China, 468–478.

[26] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic Repair of Real Bugs in Java: A Large-scale Experiment on the Defects4J Dataset. *Empirical Software Engineering* 22, 4 (Aug. 2017), 1936–1964.

[27] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16), Demonstration Track*. ACM, New York, NY, USA, 441–444.

[28] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In *Proceedings of the 10th International Symposium on Search-Based Software Engineering (SSBSE '18). Lecture Notes in Computer Science, vol 11036*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer International Publishing, Cham, 65–86.

[29] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 492–495.

[30] Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 234–242.

[31] Martin Monperrus. 2018. Automatic Software Repair: a Bibliography. *Comput. Surveys* 51, 1, Article 17 (Jan. 2018), 24 pages.

[32] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report hal-01956501. HAL/archives-ouvertes.fr, HAL/archives-ouvertes.fr.

[33] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (Oct. 2018), 2901–2947.

[34] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 254–265.

[35] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 24–36.

[36] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 10–13.

[37] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object-Oriented Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Piscataway, NJ, USA, 648–659.

[38] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 532–543.

[39] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, Campobasso, Italy, 130–140.

[40] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. 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)*. IEEE Press, Piscataway, NJ, USA, 180–182.

[41] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374.

[42] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1–11.

[43] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. IEEE, Hangzhou, China, 479–490.

[44] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches throughTest Case Generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, New York, NY, USA, 226–236.

[45] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Piscataway, NJ, USA, 660–670.

[46] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426.

[47] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (April 2016), 34–55.

[48] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *International Workshop on Intelligent Bug Fixing (IBF '19, co-located with SANER)*. IEEE, Hangzhou, China, 1–10.

[49] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System. *Empirical Software Engineering* 24, 1 (Feb. 2019), 33–67.

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