Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset

Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, Martin Monperrus

Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset

Matias Martinez · Thomas Durieux · Romain Sommerard · Jifeng Xuan · Martin Monperrus

Abstract Defects4J is a large, peer-reviewed, structured dataset of real-world Java bugs. Each bug in Defects4J comes with a test suite and at least one failing test case that triggers the bug. In this paper, we report on an experiment to explore the effectiveness of automatic test-suite based repair on Defects4J. The result of our experiment shows that the considered state-of-the-art repair methods can generate patches for 47 out of 224 bugs. However, those patches are only test-suite adequate, which means that they pass the test suite and may potentially be incorrect beyond the test-suite satisfaction correctness criterion. We have manually analyzed 84 different patches to assess their real correctness. In total, 9 real Java bugs can be correctly repaired with test-suite based repair. This analysis shows that test-suite based repair suffers from under-specified bugs, for which trivial or incorrect patches still pass the test suite. With respect to practical applicability, it takes on average 14.8 minutes to find a patch. The experiment was done on a scientific grid, totaling 17.6 days of computation time. All the repair systems and experimental results are publicly available on Github in order to facilitate future research on automatic repair.

Keywords software repair · bugs · defects · patches · fixes

M. Martinez
University of Lugano, Via Giuseppe Buffi 13, 6900 Lugano, Switzerland
Tel.: +41 58 666 40 00
E-mail: matias.sebastian.martinez@usi.ch

J. Xuan
State Key Lab of Software Engineering, Wuhan University, 299 Bayi Road, 430072 Wuhan, China
Tel.: +86 27 6877 6139
E-mail: jxuan@whu.edu.cn

T. Durieux · R.Sommerard · M. Monperrus
INRIA & University Lille 1, 40 Avenue du Halley, 59650 Villeneuve-d'Ascq, France
Tel.: +33 03 59 57 78 00
E-mail: thomas.durieux@inria.fr
E-mail: romain.sommerard@etudiant.univ-lille1.fr
E-mail: martin.monperrus@univ-lille1.fr

1 Introduction

Automatic software repair is the process of automatically fixing bugs. Test-suite based repair, notably introduced by GenProg (Le Goues et al (2012b)), consists in synthesizing a patch that passes a given test suite, which initially has at least one failing test case. In this paper, we say that this patch is "test-suite adequate". In this recent research field, few empirical evaluations have been made to evaluate the practical ability of current techniques to repair real bugs.

For bugs in the Java programming language, one of the largest evaluations is by Kim et al (2013), but as discussed in Monperrus (2014), this evaluation suffers from a number of biases and cannot be considered as a definitive answer to the repairability of Java bugs. For bugs in the C programming language, according to bibliometrics, the most visible one is by Le Goues et al (2012a). They reported on an experiment where they ran the GenProg repair system on 105 bugs in C code. However, Qi et al (2015) have shown at ISSTA 2015 that this evaluation suffers from a number of important issues, and call for more research on systematic evaluations of test-suite based repair.

Our motivation is to conduct a novel empirical evaluation in the realm of Java bugs, in order to better understand the fundamental strengths and weaknesses of current repair algorithms.

In this paper, we present the results of an experiment consisting of running repair systems for Java on the bugs of Defects4J. Defects4J is a dataset (Just et al (2014a)) by the University of Washington consisting of 357 real-world Java bugs. It has been peer-reviewed, is publicly available, and is structured in a way that eases systematic experiments. Each bug in Defects4J comes with a test suite including both passing and failing test cases.

For selecting repair systems, our inclusion criteria is that 1) it is publicly available and 2) it runs on modern Java versions and large software applications. This results in jGenProg, an implementation of GenProg (Le Goues et al (2012b) for Java; jKali, an implementation of Kali (Qi et al (2015)) for Java; and Nopol (DeMarco et al (2014); Xuan et al (2016)).

Our experiment aims to answer to the following Research Questions (RQs):

RQ1. *Can the considered repair systems synthesize patches for the bugs of the Defects4J dataset?* Beyond repairing toy programs or seeded bugs, for this research field to have an impact, one needs to know whether the current repair algorithms and their implementations work on real bugs of large scale applications.

RQ2. *In test-suite based repair, are the generated patches correct, beyond passing the test suite?* By "correct", we mean that the patch is meaningful, really repairing the bug, and is not a partial solution that only works for the input data encoded in the test cases. Indeed, a key concern behind test-suite based repair is whether test suites are good enough to drive the generation of correct patches, where correct means acceptable. Since the inception of the field, this question has been raised many times and is still a hot question: Qi et al (2015)'s recent results show that most of GenProg's patches on the now classical GenProg benchmark of 105 bugs are incorrect. We will answer RQ2 with a manual analysis of patches synthesized for Defects4J.

RQ3. *Which bugs in Defects4j are under-specified?* A bug is said to be under-specified if there exists a trivial patch that simply removes functionality. For those bugs, current repair approaches fail to synthesize a correct patch due to the lack of test cases. Those bugs are the most challenging bugs: to automatically repair them, one needs to reason on the expected functionality below what is encoded in the test

suite, to take into account a source of information other than the test suite execution. One outcome of our experiment is to identify those challenging bugs.

RQ4. *How long is the execution of each repair system?* The answer to this question also contributes to assess the practical applicability of automatic repair in the field.

Our experiment considers 224 bugs that are spread over 231K lines of code and 12K test cases in total. We ran the experiment for over 17.6 days of computational time on Grid'5000 (Bolze et al (2006)), a large-scale grid for scientific experiments.

Our contributions are as follows:

- **Answer to RQ1**. The Defects4J dataset contains bugs that can be automatically repaired with at least one of the considered systems. jGenProg, jKali, and Nopol together synthesize test-suite adequate patches for 47 out of 224 bugs with 84 different patches. Some bugs are repaired by all three considered repair approaches (12/47). **These results validate that automatic repair works on real bugs in large Java programs[1]. They can be viewed as a baseline for future usage of Defects4J in automatic repair research.**
- **Answer to RQ2**. Our manual analysis of all 84 generated patches shows that 11/84 are correct, 61/84 are incorrect, and 12/84 require a domain expertise, which we do not have. The incorrect patches tend to overfit the test cases. This is a novel piece of empirical evidence on real bugs that either the current test suites are too weak or the current automatic repair techniques are too dumb. **These results strengthen the findings in Qi et al (2015), on another benchmark, on another programming language. They show that the overfitting problem uncovered on the Genprog's benchmark by Qi et al (2015) is not specific to the benchmark and its weakness, but is more fundamental.**
- **Answer to RQ3**. Defects4J contains very weakly specified bugs. Correctly repairing those bugs can be considered as the next milestone for the field. **This result calls for radically new automatic repair approach that reasons beyond the test suite execution, using other sources of information, and for test suite generation techniques tailored for repair.**
- **Answers to RQ4**. The process of searching for a patch is a matter of minutes for a single bug. **This is an encouraging piece of evidence that this research will have an impact for practitioners.**

For the sake of open science and reproducible research, our code and experimental data are publicly available on Github (ExperimentalData (2016); AstorCode (2016); NopolCode (2016)).

The remainder of this paper is organized as follows. Section 2 provides the background of test-suite based repair and the dataset. Section 3 presents our experimental protocol. Section 4 details answers to our research questions. Section 5 studies three generated patches in details. Section 6 discusses our results and Section 7 presents the related work. Section 8 concludes this paper and proposes future work.

2 Background

In this paper, we consider one kind of automatic repair called test-suite based repair. We now give the corresponding background and present the dataset that is used in our experiment.

[1] The dataset and the repair system in Kim et al (2013) are not publicly available.

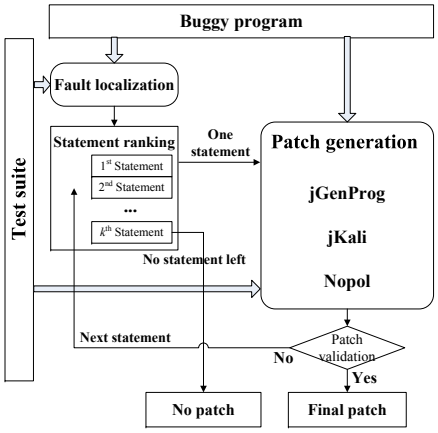

Fig. 1 Overview of test-suite based repair. Automatic repair takes a buggy program and its test suite as input; the output is the patch that passes the whole test suite if any.

2.1 Overview of Automatic Repair Techniques

We briefly give an overview of automatic repair. For a survey on the field, we refer the reader to Monperrus (2015). There are two families of repair techniques: offline repair consists of generating source code patches, online repair, aka runtime repair, consists of modifying the system state at runtime to overcome failures. The latter is the direct descendant of classical fault tolerance. Automatic repair techniques use a variety of oracles to encode the expected behavior of the program under repair: the notable oracles are test-suites, pre- and post-conditions, formal behavioral models and runtime assertions. In the experiment presented in this paper, the considered benchmark makes us considering *offline patch generation* using *test suites* as oracles.

2.2 Test-Suite Based Repair

Test-suite based repair generates a patch according to failing test cases. Different kinds of techniques can be used, such as genetic programming search in GenProg (Le Goues et al (2012b)) and SMT based program synthesis in SemFix (Nguyen et al (2013)). Often, before patch generation, a fault localization method is applied to rank the statements according to their suspiciousness. The intuition is that the patch generation technique is more likely to be successful on suspicious statements.

Fig. 1 presents a general overview of test-suite based repair approaches. In a repair approach, the input is a buggy program as well as its test suite; the output is a patch that makes the test suite pass, if any. To generate a patch for the buggy program, the executed statements are ranked to identify the most suspicious statements. *Fault localization* is a family of techniques for ranking potential buggy statements (Jones et al (2002); Abreu et al (2007); Xuan and Monperrus (2014)). Based on the statement ranking, *patch generation* tries to modify a suspicious statement. For instance, GenProg (Le Goues et al (2012b)) adds, removes, and replaces Abstract Syntax Tree (AST) nodes. Once a patch is found, the whole test suite is executed to validate the patch;

Table 1 The Main Descriptive Statistics of Considered Bugs in Defects4J. The Number of Lines of Code and the Number of Test Cases are Extracted from the Most Recent Version of Each Project.

Project	#Bugs	Source KLoC	Test KLoC	#Test cases
Commons Lang	65	22	6	2,245
JFreeChart	26	96	50	2,205
Commons Math	106	85	19	3,602
Joda-Time	27	28	53	4,130
Total	224	231	128	12,182

if the patch is not validated by the test suite, the repair approach goes on with next statement and repeats the repair process.

2.3 Defects4J Dataset

Defects4J by Just et al (2014a) is a bug database that consists of 357 real-world bugs from five widely-used open-source Java projects. Bugs in Defects4J are organized in a unified structure that abstracts over programs, test cases, and patches.

Defects4J provides research with reproducible software bugs and enables controlled studies in software testing research (e.g., Noor and Hemmati (2015); Just et al (2014b)). To our knowledge, Defects4J is the largest open database of well-organized real-world Java bugs. In our work, we use four out of five projects, i.e., Commons Lang,[2] JFreeChart,[3] Commons Math,[4] and Joda-Time.[5] We do not use the Closure Compiler project[6] because the test cases in Closure Compiler are organized in a non-conventional way, using scripts rather than standard JUnit test cases. This prevents us from running with our platform and is left for future work. Table 1 presents the main descriptive statistics of bugs in Defects4J.

The advantages of using Defects4J for a automatic repair experiment are: (realism) it contains real bugs (as opposed to seeded bugs as in Nguyen et al (2013); Kong et al (2015)); (scale) bugs are in large software (as opposed to bugs in student programs as in Smith et al (2015)); (novelty) nobody has ever studied Defects4J for repair.

3 Experimental Protocol

We present an experimental protocol to assess the effectiveness of different automatic repair approaches on the real-world bugs of Defects4J. The protocol supports the analysis of several dimensions of automatic repair: test-suite adequacy, patch correctness, under-specified bugs, performance. We first list the Research Questions (RQs) of our work; then we describe the experiment design; finally, we present the implementation details.

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

3.1 Research Questions

3.1.1 ***RQ1****. Test-suite Adequate Repair*

For how many bugs can each system synthesize a patch that is test-suite adequate, i.e., it fulfills the test suite?

This is the basic evaluation criterion of automatic test-suite based repair research. In test-suite based repair, a bug is said to be automatically patchable if the synthesized patch makes the whole test suite pass. Test-suite adequacy does not convey a notion a correctness beyond the test suite. To answer this question, we run each repair approach on each bug and count the number of bugs that are patched according the test suite.

3.1.2 ***RQ2****. Patch Correctness*

Which generated patches are semantically correct beyond passing the test suite?

A patch that passes the whole test suite may not be exactly the same as the patch written by developers. It may be syntactically different yet correct. As shown by Qi et al (2015) it may also be incorrect. To answer this question, we manually examine all synthesized patches to identify the correctness.

3.1.3 ***RQ3****. Under-Specified Bugs*

Which bugs in Defects4j are not sufficiently specified by the test suite?

In test-suite based repair, the quality of a synthesized patch is highly dependent on the quality of the test suite. In this paper, we define an "under-specified bug" as a bug for which the test cases that specify the expected behavior have a low coverage and weak assertions. Among the incorrect patches identified in RQ2, we identify the bugs for which the primary reason for incorrectness is the test suite itself (and not the repair technique). To find such under-specified bugs, we use two pieces of evidence. First, we closely look at the results of jKali. Since this repair system only removes code or skips code execution, if it finds a patch, it hints that a functionality is not specified at all. Second, we complement this by a manual analysis of generated patches.

3.1.4 ***RQ4****. Performance (Execution Time)*

How long is the execution time of each repair system?

It is time-consuming to manually repair a bug. Test-suite based repair automates the process of patch generation. To assess to which extent automatic repair is usable in practice, we evaluate the execution time of each repair approach.

3.2 Experiment Design

According to the inclusion criteria presented in Section 3.2.1, we select three repair systems (Section 3.2.2) on the Defects4J dataset (Section 2.3). Since the experiment requires a large amount of computation, we run it on a grid (Section 3.2.3). We then manually analyze all the synthesized patches (Section 3.2.4).

3.2.1 Inclusion Criteria

For selecting repair systems, we have three inclusion criteria.

Criterion #1 In this experiment, we consider a dataset of bugs in software written in the Java programming language. We consider repair systems that are able to handle this programming language. This rules out many systems that repair C code, such as GenProg (Le Goues et al (2012b)) and Semfix (Nguyen et al (2013)).

Criterion #2 Defects4J contains bugs written in modern Java versions (Java 5, 7 and 8). We consider repair systems that support those versions. This rules out Arcuri's pioneering prototype Jaff (Arcuri and Yao (2008)).

Criterion #3 The third inclusion criterion is that the repair system is available, whether publicly or on demand. This rules out for instance PAR (Kim et al (2013)).

3.2.2 Repair Systems Under Study

As of submission time, there are three repair systems that meet all those criteria: jGenProg (AstorCode (2016); Martinez and Monperrus (2016)), jKali (AstorCode (2016)) and Nopol (NopolCode (2016)).

jGenProg jGenProg is an implementation of GenProg for Java. It implements the GenProg algorithm (Le Goues et al (2012b)) as follows. It randomly deletes, adds, and replaces AST nodes in the program. The modification point is steered by spectrum based fault localization. Pieces of code that are inserted through addition or replacement always come from the same program, based on the "redundancy hypothesis" (Martinez et al (2014)). There is a threat that this Java implementation does not reflect the actual performance of the original GenProg system for C, it is discussed in Section 6.1.2. We are the authors of jGenProg, and we make it publicly available at Github AstorCode (2016); Martinez and Monperrus (2016). Our rationale for implementing jGenProg is that GenProg (which works for C code) is arguably the baseline of this research area, being the most known and visible patch generation system. jGenProg has 9.5K lines of Java code.

jKali jKali is an implementation of Kali for Java. Kali (Qi et al (2015)) performs patch synthesis by only removing or skipping code. Kali is clearly not a "program repair" technique. However, it is a perfectly appropriate technique to identify weak test suites and under-specified bugs. We are the authors of jKali and we make it publicly available (AstorCode (2016)). Our rationale for implementing Kali is that is an appropriate technique to detect test suite weaknesses, yet the original Kali works for C.

Nopol Nopol is a repair system for Java (DeMarco et al (2014); Xuan et al (2016)) which targets a specific fault class: conditional bugs. It repairs programs by either modifying an existing if-condition or adding a precondition (aka. a guard) to any statement or block in the code. The modified or inserted condition is synthesized via input-output based code synthesis with SMT solvers (Jha et al (2010)). Nopol is publicly available (NopolCode (2016)) and has 25K lines of Java code.

Open Science We note that we are authors of those three systems. This is both good and bad. This is good because we know that they are of similar quality. This is bad because it shows that as of today, although a number of automatic repair contributions have been made for Java, the current practice is not to share the tools for sake of open and reproducible science.

3.2.3 Large Scale Execution

We assess three repair approaches on 224 bugs. One repair attempt may take hours to be completed. Hence, we need a large amount of computation power. Consequently, we deploy our experiment in Grid'5000, a grid for high performance computing (Bolze et al (2006)). In our experiment, we manually set the computing nodes in Grid'5000 to the same hardware architecture. This avoids potential biases of the time cost measurement. All the experiments are deployed in the Nancy site of Grid'5000 (located in Nancy, France).

As we shall see later, this experiment is significantly CPU intensive: it takes in total 17.6 days of computation on Grid'5000. This means we had to make careful choices before coming to valid run of the experiment. In particular, we set the timeout to three hours per repair attempt, in order to have an upper bound.

For the same reason, although jGenProg is a randomized algorithm, for all but 5 bugs, we run it only once also to keep an acceptable experiment time, this important threat to validity is discussed in Section 6.1.4. This results in a conservative underestimation of GenProg's effectiveness but remains sound (found patches are actually test-suite adequate patches). Finally, although our and other's experience with repair has shown that multiple patches exist for the same bug, we stop the execution of a repair attempt after finding the first patch.

3.2.4 Manual Analysis

We manually examine the generated patches. For each patch, one of the authors, called thereafter an "analyst", analyzed the patch correctness, readability, and the difficulty of validating the correctness (as defined below). For sake of cross validation, the analyst then presents the results of the analysis to another co-author, called thereafter a "reviewer" in a live session. At the end of the live session, when an agreement has been reached between the analyst and the reviewer, a short explanatory text is written. Those texts are made publicly available for peer review (ExperimentalData (2016)). During agreement sessions, analysts and reviewers applied a conservative patch evaluation strategy: when there were a doubt or a disagreement about the correctness of a patch, is was labeled as "unknown".

Correctness Analysis The *correctness* of a patch can be correct, incorrect, or unknown. The term "correct" denotes that a patch is exactly the same or semantically equivalent to the patch that is written by developers. We assume that all patches included in the Defects4J dataset are correct. The equivalence is assessed according to the analyst's understanding of the patch. Analyzing one patch requires a period between a couple of minutes and several hours of work, depending on the complexity of the synthesized patch. On the one hand, a patch that is identical to the one written by developers is obviously true; on the other hand, several patches require a domain expertise that none of the authors has.

Note that in the whole history of automatic program repair, this is only the second time that such a large scale analysis of correctness is being performed (after Qi et al (2015)). While other published literature has studied aspects of the readability and usefulness of generated patches (e.g. Fry et al (2012); Tao et al (2014)), it was on a smaller scale with no clear focus on correctness.

Readability Analysis The *readability* of the patch can be "easy", "medium", or "hard"; and it results from the analyst opinion on the length and complexity of the patch. This subjective evaluation concerns: 1) The number of variables involved in the patch: a patch that refers to one or two variables is much easier to understand than a patch that refers to 5 variables. 2) The number of logical clauses: in a condition, a patch with a single conjunction (hence with two logical clauses) is considered easier a patch that with many sub-conjunctions. 3) The number of method calls: when a patch contains a method call, one must understand the meaning and the goal of this call to understand the patch. Hence, the more method calls, the harder the patch to understand.

Difficulty Analysis The *difficulty* analysis relates to the difficulty of understanding the problem domain and assessing the correctness. It indicates the effort that the analyst had to carry out for understanding the essence of the bug, the human patch and the generated patch. The analysts agree on a subjective difficulty on the scale "easy", "medium", "hard", or "expert" as follows. "Easy" means that it is enough to examine the source code of the patch for determining its correctness. "Medium" means that one has to fully understand the test case. "Hard" means that the analyst has to dynamically debug the buggy and/or the patched application. By "expert", we mean a patch for which it was impossible for us to validate due to the required expertise in domain knowledge. This happens when the analyst could not understand the bug, or the developer solution or the synthesized patch.

4 Empirical Results

We present and discuss our answers to the research questions that guide this work. The total execution of the experiment costs 17.6 days.

4.1 Test-suite Adequate Repair

RQ1. For how many bugs can each system synthesize a patch that fulfills the test suite?

The three automatic repair approaches in this experiment are able to together find test-suite adequate patches for 47 bugs of the Defects4J dataset. jGenProg finds a patch for 27 bugs; jKali identifies a patch for 22 bugs; and Nopol synthesizes a condition that makes the test suite passing for 35 bugs. Table 2 shows the bug identifiers, for which at least one patch is found. Each line corresponds to one bug in Defects4J and each column denotes the effectiveness of one repair approach. For instance, jGenProg and jKali are able to synthesize a test-suite adequate patch for Bug M2 from Commons Math.

Table 2 Results for 224 Bugs in Defects4J with Three Repair Approaches. In Total, the Three Repair Approaches can Generate Test-suite Adequate Patches for 47 Bugs (21%).

Project	Bug Id	jGenProg	jKali	Nopol
JFreeChart	C1	Patch	Patch	–
	C3	Patch	–	Patch
	C5	Patch	Patch	Patch
	C7	Patch	–	–
	C13	Patch	Patch	Patch
	C15	Patch	Patch	–
	C21	–	–	Patch
	C25	Patch	Patch	Patch
	C26	–	Patch	Patch
Commons Lang	L39	–	–	Patch
	L44	–	–	Patch
	L46	–	–	Patch
	L51	–	–	Patch
	L53	–	–	Patch
	L55	–	–	Patch
	L58	–	–	Patch
Time	T4	Patch	Patch	–
	T11	Patch	Patch	Patch
Commons Math	M2	Patch	Patch	–
	M5	Patch	–	–
	M8	Patch	Patch	–
	M28	Patch	Patch	–
	M32	–	Patch	Patch
	M33	–	–	Patch
	M40	Patch	Patch	Patch
	M42	–	–	Patch
	M49	Patch	Patch	Patch
	M50	Patch	Patch	Patch
	M53	Patch	–	–
	M57	–	–	Patch
	M58	–	–	Patch
	M69	–	–	Patch
	M70	Patch	–	–
	M71	Patch	–	Patch
	M73	Patch	–	Patch
	M78	Patch	Patch	Patch
	M80	Patch	Patch	Patch
	M81	Patch	Patch	Patch
	M82	Patch	Patch	Patch
	M84	Patch	Patch	–
	M85	Patch	Patch	Patch
	M87	–	–	Patch
	M88	–	–	Patch
	M95	Patch	Patch	–
	M97	–	–	Patch
	M104	–	–	Patch
	M105	–	–	Patch
Total	47 (21%)	27 (12%)	22 (9.8%)	35 (15.6%)

As shown in Table 2, some bugs such as T11 can be patched by all systems, others by only a single one. For instance, only Nopol synthesize a test-suite adequate patch for bug L39 and only jGenProg for bug M5.

Moreover, Table 2 shows that in project Commons Lang only Nopol generates test-suite adequate patches while jGenProg and jKali fail to synthesize a single patch. A possible reason is that the program of Commons Lang is more complex than that of Commons Math; both jGenProg and jKali cannot handle such a complex search space.

Fig. 2 shows the intersections among the three repair approaches as a Venn diagram, where each number is a number of patches for which one system is able to generate a test-suite adequate patch. Nopol can synthesize a patch for 18 bugs that neither jGenProg nor jKali could handle. All the bugs handled by jKali can also be handled by jGenProg and Nopol. For 12 bugs, all three repair systems can generate a patch to pass the test suite. The 2 discarded bugs will be discussed in Section 6.2.

To our knowledge, those results are the very first on automatic repair with the Defects4J benchmark. Recall that they are done with an open-science ethics, all the implementations, experimental code, and results are available on Github (ExperimentalData (2016)). Future research in automatic repair may try to synthesize more test-suite adequate patches than our work. Our experimental framework can be used to facilitate future comparisons by other researchers.

Among the incorrect patches identified in RQ2, we identify the bugs for which the primary reason for incorrectness is the test suite itself (and not the repair technique).

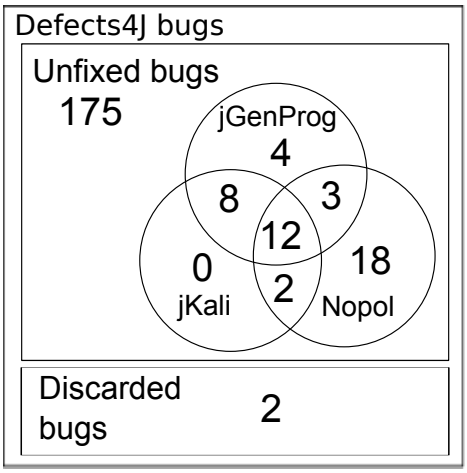

Fig. 2 Venn diagram of bugs for which test-suite adequate patches are found.

Answer to RQ1. In Defects4J, at least one of the automatic repair systems under consideration can generate a test-suite adequate patch for 47 out of 224 bugs. Nopol can is the most effective (35 bugs). If jKali finds a test-suite adequate patch, so can jGenProg or Nopol. This experiment sets a record to be beaten by future yet-to-be-invented repair systems.

4.2 Patch Correctness

RQ2. Which patches are semantically correct beyond passing the test suite?

Table 3 Manual Assessment of 84 Patches that are Generated by Three Repair Approaches.

Project	Bug id	Patch id	Approach	Correctness	Readability	Difficulty
JFreeChart	C1	1	jGenProg	Incorrect	Easy	Easy
	C1	2	jKali	Incorrect	Easy	Easy
	C3	3	jGenProg	Unknown	Medium	Medium
	C3	4	Nopol	Incorrect	Easy	Medium
	C5	5	jGenProg	Incorrect	Easy	Medium
	C5	6	jKali	Incorrect	Easy	Medium
	C5	7	Nopol	**Correct**	Easy	Medium
	C7	8	jGenProg	Incorrect	Easy	Easy
	C13	9	jGenProg	Incorrect	Easy	Easy
	C13	10	jKali	Incorrect	Easy	Easy
	C13	11	Nopol	Incorrect	Easy	Easy
	C15	12	jGenProg	Incorrect	Easy	Medium
	C15	13	jKali	Incorrect	Medium	Medium
	C21	14	Nopol	Incorrect	Hard	Expert
	C25	15	jGenProg	Incorrect	Medium	Medium
	C25	16	jKali	Incorrect	Medium	Medium
	C25	17	Nopol	Incorrect	Easy	Easy
	C26	18	jKali	Incorrect	Easy	Medium
	C26	19	Nopol	Incorrect	Easy	Medium
Commons Lang	L39	20	Nopol	Incorrect	Easy	Medium
	L44	21	Nopol	**Correct**	Easy	Medium
	L46	22	Nopol	Incorrect	Easy	Medium
	L51	23	Nopol	Incorrect	Easy	Easy
	L53	24	Nopol	Incorrect	Hard	Expert
	L55	25	Nopol	**Correct**	Easy	Medium
	L58	26	Nopol	**Correct**	Easy	Medium
Time	T4	80	jGenProg	Incorrect	Easy	Medium
	T4	81	jKali	Incorrect	Easy	Medium
	T11	82	jGenProg	Incorrect	Easy	Easy
	T11	83	jKali	Incorrect	Easy	Easy
	T11	84	Nopol	Incorrect	Medium	Medium
Commons Math	M2	27	jGenProg	Incorrect	Easy	Hard
	M2	28	jKali	Incorrect	Easy	Hard
	M5	29	jGenProg	**Correct**	Easy	Easy
	M8	30	jGenProg	Incorrect	Easy	Easy
	M8	31	jKali	Incorrect	Easy	Easy
	M28	32	jGenProg	Incorrect	Medium	Hard
	M28	33	jKali	Incorrect	Easy	Hard
	M32	34	jKali	Incorrect	Easy	Easy
	M32	35	Nopol	Unknown	Hard	Expert
	M33	36	Nopol	Incorrect	Medium	Medium
	M40	37	jGenProg	Incorrect	Hard	Hard
	M40	38	jKali	Incorrect	Easy	Medium
	M40	39	Nopol	Unknown	Hard	Expert
	M42	40	Nopol	Unknown	Medium	Expert
	M49	41	jGenProg	Incorrect	Easy	Medium
	M49	42	jKali	Incorrect	Easy	Medium
	M49	43	Nopol	Incorrect	Easy	Medium
	M50	44	jGenProg	**Correct**	Easy	Easy
	M50	45	jKali	**Correct**	Easy	Easy
	M50	46	Nopol	**Correct**	Easy	Medium
	M53	47	jGenProg	**Correct**	Easy	Easy
	M57	48	Nopol	Incorrect	Medium	Medium
	M58	49	Nopol	Incorrect	Medium	Hard
	M69	50	Nopol	Unknown	Medium	Expert
	M70	51	jGenProg	**Correct**	Easy	Easy
	M71	52	jGenProg	Unknown	Medium	Hard
	M71	53	Nopol	Incorrect	Medium	Hard
	M73	54	jGenProg	**Correct**	Easy	Easy
	M73	55	Nopol	Incorrect	Easy	Easy
	M78	56	jGenProg	Unknown	Easy	Hard
	M78	57	jKali	Unknown	Easy	Hard
	M78	58	Nopol	Incorrect	Medium	Hard
	M80	59	jGenProg	Incorrect	Hard	Medium
	M80	60	jKali	Unknown	Easy	Medium
	M80	61	Nopol	Unknown	Easy	Medium
	M81	62	jGenProg	Incorrect	Easy	Medium
	M81	63	jKali	Incorrect	Easy	Medium
	M81	64	Nopol	Incorrect	Easy	Medium
	M82	65	jGenProg	Incorrect	Easy	Medium
	M82	66	jKali	Incorrect	Easy	Medium
	M82	67	Nopol	Incorrect	Easy	Medium
	M84	68	jGenProg	Incorrect	Easy	Easy
	M84	69	jKali	Incorrect	Easy	Easy
	M85	70	jGenProg	Unknown	Easy	Easy
	M85	71	jKali	Unknown	Easy	Easy
	M85	72	Nopol	Incorrect	Easy	Easy
	M87	73	Nopol	Incorrect	Medium	Expert
	M88	74	Nopol	Incorrect	Easy	Medium
	M95	75	jGenProg	Incorrect	Easy	Hard
	M95	76	jKali	Incorrect	Easy	Hard
	M97	77	Nopol	Incorrect	Easy	Medium
	M104	78	Nopol	Incorrect	Hard	Expert
	M105	79	Nopol	Incorrect	Medium	Medium
84 Patches for 47 bugs				11 Correct	61 Easy	21 Hard/Expert
5 patches correct from jGenProg, 1 from jKali and 5 from Nopol						

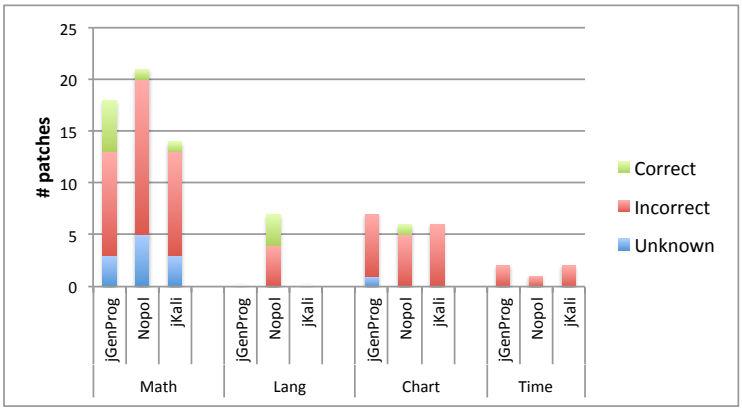

Fig. 3 Bar chart that summarizes the results from the correctness analysis.

We manually evaluate the correctness of generated patches by the three repair approaches under study. A generated patch is *correct* if this patch is the same as the manually-written patch by developers or the patch is semantically equivalent to the manual patch. A generated patch is *incorrect* if it actually does not repair the bug (beyond test-suite adequacy, beyond making the failing test case pass – a kind of incomplete bug oracle) or if it breaks an expected behavior (beyond keeping the rest of the test suite passing).

Recall the history of automatic repair research. It has been hypothesized that a major pitfall of test-suite based repair is that a test suite cannot completely express the program specifications, so it is hazardous to drive the synthesis of a correct patch with a test suite (Qi et al (2015); Fry et al (2012)). Previous works have studied the maintainability of automatic generated patches (Fry et al (2012)) or their aids for debugging task (Tao et al (2014)). However, only recent work by Qi et al (2015) has invested resources to manually analyze the correctness previously-generated patches by test-suite based repair. They found that the vast majority of patches by GenProg in the GenProg benchmark of C bugs are incorrect.

To answer the question of patch correctness, we have manually analyzed all the patches generated by Nopol, jGenProg and jKali in our experiment, 84 patches in total. This represents more than ten full days of work. To our knowledge, only Qi et al (2015) have performed a similar manual assessment of patches synthesized with automatic repair.

Table 3 shows the results of this manual analysis. The "bug id" column refers to the Defects4J identifier, while "Patch id" is an univocal identifier of each patch, for easily identifying the patch on our empirical result page (ExperimentalData (2016)). The three main columns give the correctness, readability and difficulty as explained in Section 3.2.4.

The results in Table 3 may be fallible due to the subjective nature of the assessment. For mitigating this risk, all patches as well as detailed case studies are publicly available on Github (ExperimentalData (2016)).

In total, we have analyzed 84 patches. Among these patches, 27, 22, and 35 patches are synthesized by jGenProg, jKali, and Nopol, respectively.

As shown in Table 3, 11 out of 84 analyzed patches are correct and 61 are incorrect. Meanwhile, for the other 12 patches, it is not possible to clearly validate the correctness, due to the lack of domain expertise (labeled as *unknown*). Figure 3 graphically summarizes this analysis. This figure clearly convey the fact that most test-suite adequate patches are actually incorrect because of overfitting. Section 5 will present three case studies of generated patches via manual analysis.

Among the 11 correct patches, jGenProg, jKali, and Nopol contribute to 5, 1, and 5 patches, respectively. All the correct patches by jGenProg and jKali come from Commons Math; 3 correct patches by Nopol come from Commons lang, one comes from JFreeChart and the other from Commons Math. After the controversy about the effectiveness of GenProg (Qi et al (2015)), it is notable to see that there are bugs for which only jGenProg works for real.

Among 84 analyzed patches, 61 patches are identified as easy to read and understand. For the difficulty of patch validation, 21 patches are labeled as hard or expert. This result shows that it is hard and time consuming to conduct a manual assessment of patches.

For the incorrect patches, the main reasons are as follows. First, all three approaches are able to remove some code (pure removal for jKali, replacement for jGenProg, precondition addition for Nopol). The corresponding patches simply exploit some under-specification and remove the faulty but otherwise not tested behavior. When the expected behavior seems to be well-specified (according to our understanding of the domain), the incorrect patches tend to overfit the test data. For instance, if a failing test case handles a 2×2 matrix, the patch may use such test data to incorrectly force the patch to be suitable only for matrices of size of 2×2.

This goes along the line of Qi et al (2015)'s results and Smith et al (2015)' findings. Compared to Smith et al (2015), we study real bugs in large programs and not small ones in student programs. Compared to Qi et al (2015), we study a different benchmark in a different programming language. To this extent, the key novelty of our experimental results is that: 1) overfitting also happens for Java software tested with a state-of-the-art testing infrastructure (as opposed to old-school C programs in Qi et al (2015)); 1) overfitting also happens with Nopol, a synthesis based technique (and not only for Genprog and similar techniques). Our results clearly identify overfitting as the next big challenge of test-suite based automatic repair.

Answer to RQ2. Based on the manual assessment of patch correctness, we find out that only 11 out of 84 generated patches are semantically correct (beyond passing the provided test suite). The reason is that the current test suite based repair algorithms tend to overfit the test case data or to exploit the holes left by insufficiently specified functionality.

4.3 Under-specified bugs

RQ3. Which bugs in Defects4j are not sufficiently specified by the test suite?

As shown in Section 4.1, the repair system jKali can find test-suite adequate patches for 22 bugs. Among these generated patches, from our manual evaluation, we find out that 18 patches are incorrect (other 3 patches are unknown). In each of those generated patches by jKali, one statement is removed or skipped to eliminate the failing program behavior, instead of making it correct. This kind of patches shows

Table 4 The Most Challenging Bugs of Defects4J Because of Under-specification.

Project	Bug ID
Commons Math	M2, M8,M28,M32,M40, M49, M78, M80, M81, M82,M84, M85,M95
JFreeChart	C1,C5, C13, C15, C25,C26
Time	T4,T11

that the corresponding test suite is too weak with respect to the buggy functionality. The assertions that specify the expected behavior of the removed statement and the surrounding code are inexistent or too weak.

One exception among 22 patches by jKali is the patch of Bug M50. As shown in Section 4.2, the patch of Bug M50 is correct. That is, the statement removal is the correct patch. Another special case is Bug C5 which is patched by jKali (incorrect) and by Nopol (correct). The latter approach produces a patch similar to that done by the developer. A patch (written by developers or automatically generated) that is test-suite adequate for an under-specified bug could introduce new bugs (studied previously by Gu et al (2010)) or it could not be completely correct due to a weak test suite used as the bug oracle (Qi et al (2015)). Table 4 summarizes this finding and list the under-specified bugs.

This result is important for future research on automatic repair with Defects4J. First, any repair system that claims to correctly repair one of those bugs should be validated with a detailed manual analysis of patch correctness, to check whether the patch is not a variation on the trivial removal solution. Second, those bugs can be considered as the most challenging ones of Defects4J. To repair them, a repair system must somehow reason about the expected functionality beyond what is encoded in the test suite. This is what was actually been done by the human developer. A repair system that is able to produce a correct patch for those bugs would be a great advance for the field.

Answer to RQ3. Among the incorrect patches identified in RQ2, there are 21 bugs for which the primary reason for incorrectness is the test suite itself (and not the repair technique), they are under-specified bugs. For them, the test suite does not accurately specify the expected behavior and can be trivially repaired by removing code. To us, they are the most challenging bugs: to automatically repair them, one needs to reason on the expected functionality below what is encoded in the test suite for instance by taking into account a source of information other than test suite execution. Repairing those bugs can be seen as the next grand challenge for automatic repair.

4.4 Performance

RQ4. How long is the execution time for each repair approach on one bug?

For real applicability in industry, automatic repair approaches must execute fast enough. By “fast enough”, we mean an acceptable time period, which depends on the usage scenario of automatic repair and on the hardware. For instance, if automatic repair is meant to be done in the IDE, repair time should last at most some minutes on a standard desktop machine. On the contrary, if automatic repair is meant to be done on a continuous integration server, it is acceptable to last hours on a more powerful server hardware configuration.

Table 5 Time Cost of Patch Generation

Time cost	jGenProg	jKali	Nopol
Min	40 sec	36 sec	31sec
Median	1h 01m	18m 45sec	22m 30sec
Max	1h 16m	1h 27m	1h 54m
Average	55m 50sec	23m 33sec	30m 53sec
Total	8 days 12h	3 days 6h	7 days 3h

The experiments in this paper are run on a grid where most of nodes have comparable characteristics. Typically, we use machines with Intel Xeon X3440 Quad-core processor and 15GB RAM. Table 5 shows the time cost of patch generation in hours for bugs without timeout. As shown in Table 5, the median time for one bug by jGenProg is around one hour. The fastest repair attempt yields a patch in 31 seconds (for Nopol). The median time to synthesize a patch is 6.7 minutes. This means that the execution time of automatic repair approaches is comparable to the time of manual repair by developers. It may be even faster, but we don't know the actual repair time by real developers for the bug of the dataset.

When a test-adequate patch can be synthesized, it is found within minutes. This means that most of the time of the 17.6 days of computation for the experiment is spent on unfixed bugs, which reach the timeout. For jGenProg, it is always the case, because the search space is extremely large. For jKali, we often completely explore the search space, and we only reach the timeout in 20 cases. For Nopol, the timeout is reached in 26 cases, either due to the search space of covered statements or the SMT synthesis that becomes slow in certain cases. One question is whether a larger timeout would improve the effectiveness. According to this experiment, the answer is no. The repairability is quite binary: either a patch is found fast, or the patch cannot be found at all. This preliminary observation calls for future research.

Answer to RQ4. Both the median value and the average value of repair execution time are about one hour on machines that represent server machines used nowadays for continuous integration. Performance is not a problem with respect to practical usage of automatic repair.

4.5 Other Findings in Defects4J

Our manual analysis of results enables us to uncover two problems in Defects4J. First, we found that bug #8 from project JFreeChart (C8) is flaky, which depends on the machine configuration. Second, bug #99 from Commons Math (M99) is identical to bug M97. Both issues were reported to the authors of Defects4J and will be solved in future releases of Defects4J.

5 Case Studies

In this section, we present three case studies of generated patches by jGenProg, jKali, and Nopol, respectively. These case studies are pieces of evidence that: State-of-the-art automatic repair is able to find correct patches (Sections 5.1 and 5.3), but also fails with incorrect patches (Section 5.2). It is possible to automatically generate the same patch as the one written by the developer (Section 5.1).

```
1  double solve(UnivariateRealFunction f,
2          double min, double max, double initial)
3      throws MaxIterationsExceededException,
4          FunctionEvaluationException {
5  // PATCH: return solve(f, min, max);
6    return solve(min, max);
7  }
```

Fig. 4 Code snippet of Bug M70. The manually-written patch and the patch by jGenProg are the same, which is shown in the `PATCH` comment at Line 5, which adds a parameter to the method call.

5.1 Case Study of M70, Bug that is Only Repaired by jGenProg

In this section, we study Bug M70, which is repaired by jGenProg, but cannot be repaired by jKali and Nopol.

Bug M70 in Commons Math is about univariate real function analysis. Fig. 4 presents the buggy method of Bug M70. This buggy method contains only one statement, a method call to an overloaded method. In order to perform the correct calculation, the call has to be done with an additional parameter `UnivariateRealFunction f` (at Line 1) to the method call. Both the manually-written patch and the patch by jGenProg add the parameter `f` to the method call (at Line 5). This patch generated by jGenProg is considered correct since the it is the same as that by developers.

To repair Bug M70, jGenProg generates a patch by replacing the method call by another one, which is picked elsewhere in the same class. This bug cannot be repaired by either jKali or Nopol. jKali removes and skips statements; Nopol only handles bugs that are related to IF conditions. Indeed, the fact that certain bugs are only repaired by one tool confirms that the fault classes addressed by each approach are not identical. **To sum up, Bug M7 shows that the GenProg algorithm, as implemented in jGenProg, is capable of uniquely repairing real Java bugs (only jGenProg succeeds).**

5.2 Case Study of M8, Bug that is Incorrectly Repaired by jKali and jGenProg

In this section, we present a case study of Bug M8, for which jKali as well as jGenProg find a test-suite adequate patch, but not Nopol.

Bug M8[7] in Commons Math, is about the failure to create an array of a random sample from a discrete distribution. Fig. 5 shows an excerpt of the buggy code and the corresponding manual and synthesized patches (from `class DiscreteDistribution<T>`). The method `sample` receives the expected number `sampleSize` of random values and returns an array of the type `T[]`.

The bug is due to an exception thrown at Line 11 during the assignment to `out[i]`. The method `Array.newInstance(class, int)` requires a class of a data type as the first parameter. The bug occurs when a) the first parameter is of type `T1`, which is a subclass of `T` and b) one of the samples is an object which is of type `T2`, which is a

[7] Bug ID in the bug tracking system of Commons Math is Math-942, `http://issues.apache.org/jira/browse/MATH-942`.

```
1  T[] sample(int sampleSize) {
2    if (sampleSize <= 0) {
3      throw new NotStrictlyPositiveException([...]);
4    }
5  // MANUAL PATCH:
6  // Object[] out = new Object[sampleSize];
7    T[] out = (T[]) Array.newInstance(
8      singletons.get(0).getClass(), sampleSize);
9    for (int i = 0; i < sampleSize; i++) {
10 // PATCH: removing the following line
11     out[i] = sample();
12   }
13   return out;
14 }
```

Fig. 5 Code snippet of Bug M8. The manually-written patch is shown in the `MANUAL PATCH` comment at Lines 5 and 6 (changing a variable type). The patch by jKali in the `PATCH` comment removes the loop body at Line 11.

sub-class of `T`, but not of type `T1`. Due to the incompatibility of types T1 and T2, an `ArrayStoreException` is thrown when this object is assigned to the array.

In the manual patch, the developers change the array type in its declaration (from `T[]` to `Object[]`) and the way the array is instantiated. The patch generated by jKali simply removes the statement, which assigns `sample()` to the array. As consequence, method `sample` never throws an exception but returns an empty array (only containing null values). This patch passes the failing test case and the full test suite as well. The reason of this is that the test case has only one assertion: it asserts that the array size is equal to 1. There is no assertion on the content of the returned array. However, despite passing the test suite, the patch is clearly incorrect. This is an example of a bug that is not well specified by the test suite. For this bug, jGenProg can also generate a patch by replacing the assignment by a side-effect free statement, which is semantically equivalent to removing the code. **To sum up, Bug M8 is an archetypal example of under-specified bugs as detected by the jKali system.**

5.3 Case Study of L55, Bug that is Repaired by Nopol, Equivalent to the Manual Patch

Nopol (DeMarco et al (2014)) focuses on condition-related bugs. Nopol collects runtime data to synthesize a condition patch. In this section, we present a case study of Bug L55, which is only repaired by Nopol, but cannot be repaired by jGenProg or jKali.

Bug L55 in Commons Lang relates to a utility class for timing. The bug appears when the user stops a suspended timer: the stop time saved by the suspend action is overwritten by the stop action. Fig. 6 presents the buggy method of Bug L55. In order to solve this problem, the assignment at Line 10 has to be done only if the timer state is running.

As shown in Fig. 6, the manually-written patch by the developer adds a precondition before the assignment at Line 10 and it checks that the current timer state is running (at Line 7). The patch by Nopol is different from the manually-written one. The Nopol patch compares the stop time variable to a integer constant (at Line 9), which is pre-defined in the program class and equals to 1. In fact, when the timer

```
1  void stop() {
2    if (this.runningState != STATE_RUNNING
3        && this.runningState != STATE_SUSPENDED) {
4      throw new IllegalStateException(...);
5    }
6  // MANUAL PATCH:
7  // if (this.runningState == STATE_RUNNING)
8  // NOPOL PATCH:
9  // if (stopTime < StopWatch.STATE_RUNNING)
10   stopTime = System.currentTimeMillis();
11   this.runningState = STATE_STOPPED;
12 }
```

Fig. 6 Code snippet of Bug L55. The manually-written patch is shown in the `MANUAL PATCH` comment at Lines 6 and 7 while the patch by Nopol is shown in the `NOPOL PATCH` at Lines 8 and 9. The patch by Nopol is equivalent to the manually-written patch by developers.

is running, the stop time variable is equals to -1; when it is suspended, the stop time variable contains the stop time in millisecond. Consequently, both preconditions by developers and by Nopol are equivalent and correct. Despite being equivalent, the manual patch remains more understandable. This bug is neither repaired by jGenProg nor jKali. To our knowledge, Nopol is the only approach that contains a strategy of adding preconditions to original statements, which does not exist in jGenProg or jKali. **To sum up, Bug L55 shows an example of a repaired bug, 1) that is in a hard-to-repair project (only Nopol succeeds) and 2) whose specification by the test suite is good enough to drive the synthesis of a correct patch.**

Summary. In this section, we have presented detailed case studies of three patches that are automatically generated for three real-world bugs of Defects4J. Our case studies show that automatic repair approaches are able to repair real bugs. However, different factors, in particular the weakness of some test cases, yield clearly incorrect patches.

6 Discussion

6.1 Threats to Validity

6.1.1 Benchmark

Our results are obtained on four subject programs, from a benchmark in which there was no attempt to provide a representative set of fault classes. Future experiments on other benchmarks are required mitigate this threat to the external validity.

We have taken the bugs of the Defects4J benchmark as they are and we did not modify the test cases. This is very important to decrease the following threats. First, we measure the effectiveness of the repair tool as it would have be at the time when the bug was reported. We do not use other tests of information "from the future". It would introduce a potential experimental bias if we modify the benchmark, which has been set up independently of our experiment.

6.1.2 Implementations of GenProg and Kali

jGenProg and jKali are the re-implementations of the GenProg and Kali algorithms in Java. There exists a threat that the implementations do not produce results that are as good as the original systems would (there is also a risk that the re-implementation produces better results). The authors of jGenProg and jKali carefully and faithfully reimplemented the original systems. Note this threat is absolute. Either one takes the risk or one give up any comparative evaluations between programming languages. To find potential re-implementation issues, the systems are publicly available on Github for peer-review.

6.1.3 Bias of assessing the correctness, readability, and difficulty

In our work, each patch in Table 3 is validated by an analyst, which is one of the authors. An analyst manually identifies the correctness of a patch and labels the related readability and difficulty. Then, he discusses and validates the result of the patch analysis with another author, called a reviewer. Since patch correctness analysis on an unknown code base is a new kind of qualitative analysis, only explored by Qi et al (2015), we do not know its inherent problems and difficulty. This is why the methodological setup is a pilot one, where no independent validation has been attempted to measure inter-annotator agreement. This results in a threat to the internal validity of the manual assessment.

We share our results online on the experiment Github repository to let readers have a clear idea of the difficulty of our analysis work and the correctness of generated patches (see Section 4.1). For assessing the equivalence, one solution would be to use automatic technique, as done in mutation testing. However, whether the current equivalence detection techniques scale on large real Java code is an open question.

6.1.4 Random nature of jGenProg

jGenProg, as the original GenProg implementation, has a random aspect: statements and mutations are randomly chosen during the search. Consequently, it may happen that a new run of jGenProg yields different results. There are two different threats. Our experiment may have over-estimated the repair effectiveness of GenProg (too many patches found) or underestimated the repair effectiveness (too few).

To analyze the overestimation threat, for each of the 27 repaired bug, we have run jGenProg ten times with different seeds. For $18/27$ cases, a test-adequate patch was found in all 10 runs, in 6 cases, a test-adequate patch was found in at least 7 runs. In the remaining 3 cases, a patch was found in a minority of runs, and we can consider that our reference run was lucky to find at least one test-adequate patch. Note that this experiment is quite fast because the runs are mostly successful and hence terminate fast, much before the 3-hour time budget limit.

Analyze the underestimation threat is much more challenging, because if a bug is not repaired, it means that the trial lasted at least 3 hours (the time budget used in this experiment). Consequently, if the bug is really unrepairable, trying with n different seeds would require $n \times 3$ hours of computation. For $224 - 27 = 197$ unrepaired bugs, this sets up a worst case bound of $197 \times 10 \times 3 = 5910$ hours of computation, i.e. 224 days, which is not reasonable, even split into parallel tasks. Consequently, we decided to sample 5 unrepaired bugs and run with 10 different seeds (representing an experimental

budget of $5 \times 10 \times 3 = 150$ hours). The result was clear cut: no additional patches could be found. This suggests that the "unrepairedness" comes from the essential complexity of the problem and not from the accidental randomness of jGenProg.

6.1.5 Presence of multiple patches

In this experiment, we stop the execution of a repair system once the first patch is found. This is the patch that is manually analyzed. However, as also experienced by others, there are often several if not dozens of different patches per bug. It might happen that a correct patch lies somewhere in this set of generated patches. We did not manually analyze all generated patches because it would require months of manual work. This finding shows that there is a need for research on approaches that order the generated patches so as to reveal the most likely to be correct, this has been preliminarily explored by Long and Rinard (2016b).

6.2 Impact of Flaky Tests on Repair

Our experiment uncovered one flaky test in Defects4J (C8). We realized that flaky tests have a direct impact on automatic repair. If the failing test case is flaky, the repair system might conclude that a correct patch has been found while it is actually correct. If one of the passing test cases is flaky, the repair system might conclude that a patch has introduced a regression while it is not the case, this results in an underestimation of the effectiveness of the repair technique.

6.3 Reflections on GenProg

The largest evaluations of GenProg are by Le Goues et al (2012a) and Qi et al (2015). The former reports that 55/105 bugs are repaired (under the definition that the patch passes the test suite), while the latter argued that only 2/105 bugs are correctly repaired (under the definition that the patch passes the test suite and that the patch is correct and acceptable from the viewpoint of the developer). The difference is due to an experimental threat, the presence of under-specified bugs and a potential subjectiveness in the assessment.

In this paper, we find that our re-implementation of GenProg, jGenProg correctly repairs 5/224 (2.2%) bugs. In addition, it uniquely repairs 4 bugs, such as M70 discussed in Section 5.1. We think that the difference in repair rate is probably due to the inclusion criteria of both benchmarks (GenProg and Defects4J). To our opinion, none of them reflect the actual distribution of all bug kinds and difficulty in the field.

However, the key finding is that having correctly and uniquely repaired bugs indicates that the core intuition of GenProg is valid, and that GenProg has to be a component of an integrated repair tool that would mix different repair techniques.

6.4 On the Choice of Repair Techniques

Given a new bug, can one choose the most appropriate repair technique? In practice, when a new bug is reported, one does not know in advance the type of its root cause.

The root cause may be an incorrect condition, a missing statement, or something else. It means that given a failure (or a failing test case) there is no reason to choose one or another repair technique (say jGenProg or Nopol). The pragmatic choice is to run all available techniques, whether sequentially or in parallel. However, there may be some information in the failure itself about the type of the root cause. Future results on the automatic identification of the type of root cause given a failure would be useful for repair, in order to guide the choice of repair technique.

7 Related Work

7.1 Real-World Datasets of Bugs

The academic community has already developed many methods for finding and analyzing bugs. Researchers employ real-world bug data to evaluate their methods and to analyze their performance in practice. For instance, Do et al (2005) propose a controlled experimentation platform for testing techniques. Their dataset is included in SIR database, which provides a widely-used testbed in debugging and test suite optimization.

Dallmeier and Zimmermann (2007) propose iBugs, a benchmark for bug localization obtained by extracting historical bug data. BugBench by Lu et al (2005) and BegBunch by Cifuentes et al (2009) are two benchmarks that have been built to evaluate bug detection tools. The PROMISE repository (Menzies et al (2015)) is a collection of datasets in various fields of software engineering. Le Goues et al (2015) have designed a benchmark of C bugs which is an extension of the GenProg benchmark.

In this experience report, we employ Defects4J by Just et al (2014a) to evaluate software repair. This database includes well-organized programs, bugs, and their test suites. The bug data in Defects4J has been extracted from the recent history of five widely-used Java projects. To us, Defects4J is the best dataset of real world Java bugs, both in terms of size and quality. To our knowledge, our experiment is the first that evaluates automatic repair techniques via Defects4J.

7.2 Test-Suite Based Repair Approaches

The idea of applying evolutionary optimization to repair derives from Arcuri and Yao (2008). Their work applies co-evolutionary computation to automatically generate bug patches. GenProg by Le Goues et al (2012b) applies genetic programming to the AST of a buggy program and generates patches by adding, deleting, or replacing AST nodes. PAR by Kim et al (2013) leverages patch patterns learned from human-written patches to find readable patches. RSRepair by Qi et al (2014) uses random search instead of genetic programming. This work shows that random search is more efficient in finding patches than genetic programming. Their follow-up work (Qi et al (2013)) uses test case prioritization to reduce the cost of patch generation.

Debroy and Wong (2010) propose a mutation-based repair method inspired from mutation testing. This work combines fault localization with program mutation to exhaustively explore a space of possible patches. Kali by Qi et al (2015) has recently been proposed to examine the repair power of simple actions, such as statement removal.

SemFix by Nguyen et al (2013) is a notable constraint based repair approach. This approach provides patches for assignments and conditions by combining symbolic execution and code synthesis. Nopol by DeMarco et al (2014) is also a constraint based method, which focuses on repairing bugs in IF conditions and missing preconditions. DirectFix by Mechtaev et al (2015) achieves the simplicity of patch generation with a Maximum Satisfiability (MaxSAT) solver to find the most concise patches. Relifix (Tan and Roychoudhury (2015)) focuses on regression bugs. SPR (Long and Rinard (2015)) defines a set of staged repair operators so as to early discard many candidate repairs that cannot pass the supplied test suite and eventually to exhaustively explore a small and valuable search space.

Besides test-suite based repair, other repair setups have been proposed. Pei et al (2014) proposed a contract based method for automatic repair. Other related repair methods include atomicity-violation fixing (e.g. Jin et al (2011)), runtime error repair (e.g. Long et al (2014)), and domain-specific repair (e.g. Samimi et al (2012); Gopinath et al (2014)).

7.3 Empirical Investigation of Automatic Repair

Beyond proposing new repair techniques, there is a thread of research on empirically investigating the foundations, impact and applicability of automatic repair.

On the goodness of synthesized patches, Fry et al (2012) conducted a study of machine-generated patches based on 150 participants and 32 real-world defects. Their work shows that machine-generated patches are slightly less maintainable than human-written ones. Tao et al (2014) performed a similar study to study whether machine-generated patches assist human debugging. Monperrus (2014) discussed in depth the acceptability criteria of synthesized patches.

Martinez and Monperrus (2015) studied thousands of commits to mine repair models from manually-written patches. They later investigated (Martinez et al (2014)) the redundancy assumption in automatic repair (whether you can repair bugs by rearranging existing code). Zhong and Su (2015) conducted a case study on over 9,000 real-world patches and found two important facts for automatic repair: for instance, their analysis outlines that some bugs are repaired with changing the configuration files.

A recent study by Kong et al (2015) compares four repair systems: including GenProg (Le Goues et al (2012b)), RSRepair (Qi et al (2014)), Brute-force-based repair (Le Goues et al (2012a)), and AE (Weimer et al (2013)). This work reported repair results on 119 seeded bugs and 34 real bugs from the Siemens benchmark and in the SIR repository. Our experiment is on a bigger set of real bugs coming from larger and more complex software applications.

Long and Rinard (2016a) have performed an analysis of the search space of their two systems SPR and Prophet on the benchmark of C bugs. They show that not all repair operators are equal with respect to finding the correct patch first. This is clearly different from what we do in this paper: 1) we consider a different benchmark (Defects4J), in a different programming language (Java), while they consider the Manybugs benchmark of C bugs. 2) they do not perform any manual analysis of their results.

8 Conclusion

We have presented a large scale evaluation of automatic repair on a benchmark of real bugs in Java programs. Our experiment was conducted with three automatic repair systems on 224 bugs in the Defects4J dataset. We find out that the systems under consideration can synthesize a patch for 47 out of 224. *Since the dataset only contains real bugs from large-scale Java software, this is a piece of evidence about the applicability of automatic repair in practice.*

However, our manual analysis of all generated patches show that most of them are incorrect (beyond passing the test suite), because the overfit on the test data. Our experiment shows that overfitting also happens for Java software tested with a state-of-the-art testing infrastructure (as opposed to old-school C programs in Qi et al (2015)); and also happens with Nopol, a synthesis based technique (and not only for Genprog and similar techniques).

Our results indicate two grand challenges for this research field: 1) producing systems that do not overfit (RQ2) and 2) producing systems that reason on the expected behavior beyond what is directly encoded in the test suite (RQ3). There is also a third challenge, although not the focus on this experiment: during the exploratory phase, we have observed the presence of multiple patches for the same bug. This indicates a need for research on approaches that order the generated patches so as to reveal the most likely to be correct. Preliminary work is being done on this topic by Long and Rinard (2016b).

Lastly, there is also a need for research on test suites. For instance, any approach that automatically enriches test suites with stronger assertions would have direct impact on repair, by preventing the synthesis of incorrect patches.

References

Abreu R, Zoeteweij P, Van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, IEEE, pp 89–98

Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: Proceedings of the IEEE Congress on Evolutionary Computation, pp 162–168, DOI 10.1109/CEC.2008.4630793, URL http://dx.doi.org/10.1109/CEC.2008.4630793

AstorCode (2016) The github repository of jgenprog and jkali. http://bit.ly/1OLZSAu

Bolze R, Cappello F, Caron E, Daydé M, Desprez F, Jeannot E, Jégou Y, Lanteri S, Leduc J, Melab N, et al (2006) Grid'5000: a large scale and highly reconfigurable experimental grid testbed. vol 20, SAGE Publications, pp 481–494

Cifuentes C, Hoermann C, Keynes N, Li L, Long S, Mealy E, Mounteney M, Scholz B (2009) Begbunch: Benchmarking for c bug detection tools. In: Proceedings of ISSTA, ACM, New York, USA, pp 16–20, DOI 10.1145/1555860.1555866, URL http://doi.acm.org/10.1145/1555860.1555866

Dallmeier V, Zimmermann T (2007) Extraction of Bug Localization Benchmarks from History. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, pp 433–436, DOI 10.1145/1321631.1321702, URL http://doi.acm.org/10.1145/1321631.1321702

Debroy V, Wong WE (2010) Using mutation to automatically suggest fixes for faulty programs. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, pp 65–74, DOI 10.1109/ICST.2010.66, URL http://dx.doi.org/10.1109/ICST.2010.66

DeMarco F, Xuan J, Le Berre D, Monperrus M (2014) Automatic repair of buggy if conditions and missing preconditions with smt. In: Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, ACM, pp 30–39

Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4):405–435

ExperimentalData (2016) The github repository of the experimental data. http://bit.ly/1ON6Vmf

Fry ZP, Landau B, Weimer W (2012) A human study of patch maintainability. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp 177–187

Gopinath D, Khurshid S, Saha D, Chandra S (2014) Data-guided repair of selection statements. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 243–253

Gu Z, Barr E, Hamilton D, Su Z (2010) Has the bug really been fixed? In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol 1, pp 55–64, DOI 10.1145/1806799.1806812

Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Proceedings of the International Conference on Software Engineering, IEEE, vol 1, pp 215–224

Jin G, Song L, Zhang W, Lu S, Liblit B (2011) Automated atomicity-violation fixing. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 389–400, DOI 10.1145/1993498.1993544, URL http://doi.acm.org/10.1145/1993498.1993544

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on Software engineering, ACM, pp 467–477

Just R, Jalali D, Ernst MD (2014a) Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp 437–440

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014b) Are mutants a valid substitute for real faults in software testing. In: 22nd International Symposium on the Foundations of Software Engineering (FSE 2014)

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, pp 802–811

Kong X, Zhang L, Wong WE, Li B (2015) Experience report: How do techniques, programs, and tests impact automated program repair? In: Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on, IEEE, pp 194–204

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012a) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Software Engineering (ICSE), 2012 34th International Conference on, IEEE, pp 3–13

Le Goues C, Nguyen T, Forrest S, Weimer W (2012b) Genprog: A generic method for automatic software repair. Software Engineering, IEEE Transactions on 38(1):54–72

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The manybugs and introclass benchmarks for automated repair of c programs. IEEE Transactions on Software Engineering (TSE), in press

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 166–178, DOI 10.1145/2786805.2786811, URL http://doi.acm.org/10.1145/2786805.2786811

Long F, Rinard M (2016a) An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In: Proceedings of the 38th International Conference on Software Engineering, pp 702–713

Long F, Rinard M (2016b) Automatic patch generation by learning correct code. SIGPLAN Not 51(1):298–312, DOI 10.1145/2914770.2837617, URL http://doi.acm.org/10.1145/2914770.2837617

Long F, Sidiroglou-Douskos S, Rinard MC (2014) Automatic runtime error repair and containment via recovery shepherding. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, p 26, DOI 10.1145/2594291.2594337, URL http://doi.acm.org/10.1145/2594291.2594337

Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y (2005) Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools

Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering 20(1):176–205, DOI 10.1007/s10664-013-9282-8, URL http://dx.doi.org/10.1007/s10664-013-9282-8

Martinez M, Monperrus M (2016) Astor: A program repair library for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp 441–444, DOI 10.1145/2931037.2948705, URL http://doi.acm.org/10.1145/2931037.2948705

Martinez M, Weimer W, Monperrus M (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, pp 492–495, DOI 10.1145/2591062.2591114, URL http://doi.acm.org/10.1145/2591062.2591114

Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering, IEEE

Menzies T, Krishna R, Pryor D (2015) The promise repository of empirical software engineering data. URL http://openscience.us/repo

Monperrus M (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, ACM, pp 234–242

Monperrus M (2015) Automatic software repair: a bibliography. Tech. Rep. hal-01206501, University of Lille, URL http://www.monperrus.net/martin/survey-automatic-repair.pdf

Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 772–781

Noor T, Hemmati H (2015) Test case analytics: Mining test case traces to improve risk-driven testing. In: Proceedings of the IEEE 1st International Workshop on Software Analytics, IEEE, pp 13–16

NopolCode (2016) The github repository of nopol. http://bit.ly/1mBlOlx

Pei Y, Furia CA, Nordio M, Wei Y, Meyer B, Zeller A (2014) Automated fixing of programs with contracts. IEEE Trans Software Eng 40(5):427–449, DOI 10.1109/TSE.2014.2312918, URL http://doi.ieeecomputersociety.org/10.1109/TSE.2014.2312918

Qi Y, Mao X, Lei Y (2013) Efficient automated program repair through fault-recorded testing prioritization. In: 2013 IEEE International Conference on Software Maintenance, pp 180–189, DOI 10.1109/ICSM.2013.29, URL http://dx.doi.org/10.1109/ICSM.2013.29

Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 254–265

Qi Z, Long F, Achour S, Rinard M (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, ACM, New York, NY, USA, ISSTA 2015, pp 24–36, DOI 10.1145/2771783.2771791, URL http://doi.acm.org/10.1145/2771783.2771791

Samimi H, Schäfer M, Artzi S, Millstein TD, Tip F, Hendren LJ (2012) Automated repair of HTML generation errors in PHP applications using string constraint solving. In: Proceedings of the 34th International Conference on Software Engineering, pp 277–287, DOI 10.1109/ICSE.2012.6227186, URL http://dx.doi.org/10.1109/ICSE.2012.6227186

Smith EK, Barr E, Le Goues C, Brun Y (2015) Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Bergamo, Italy, DOI 10.1145/2786805.2786825

Tan SH, Roychoudhury A (2015) Relifix: Automated repair of software regressions. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 471–482, URL http://dl.acm.org/citation.cfm?id=2818754.2818813

Tao Y, Kim J, Kim S, Xu C (2014) Automatically generated patches as debugging aids: a human study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 64–74

Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: Models and first results. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, pp 356–366

Xuan J, Monperrus M (2014) Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), ACM

Xuan J, Martinez M, Demarco F, Clément M, Lamelas S, Durieux T, Le Berre D, Monperrus M (2016) Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Transactions on Software Engineering URL https://hal.archives-ouvertes.fr/hal-01285008

Zhong H, Su Z (2015) An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, pp 913–923