A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark

He Ye
KTH Royal Institute of Technology
heye@kth.se

Matias Martinez
University of Valenciennes
matias.martinez@univ-valenciennes.fr

Martin Monperrus
KTH Royal Institute of Technology
martin.monperrus@csc.kth.se

***Abstract*—Automatic program repair papers tend to repeatedly use the same benchmarks. This poses a threat to the external validity of the findings of the program repair research community. In this paper, we perform an automatic repair experiment on a benchmark called QuixBugs that has been recently published. This benchmark has never been studied in the context of program repair. In this study, we report on the characteristics of QuixBugs, and we design and perform an experiment about the effectiveness of test-suite based program repair on QuixBugs. We study two repair systems, Astor and Nopol, which are representatives of generate-and-validate repair technique and synthesis repair technique respectively. We propose three patch correctness assessment techniques to comprehensively study overfitting and incorrect patches. Our key results are: 1) 13 / 40 buggy programs in the QuixBugs can be repaired with a test-suite adequate patch; 2) a total of 22 different plausible patches for those 13 buggy programs in the QuixBugs are present in the search space of the considered tools; 3) the three patch assessment techniques discard in total 12 / 22 patches that are overfitting. This sets a baseline for future research of automatic repair on QuixBugs. Our experiment also highlights the major properties and challenges of how to perform automated correctness assessment of program repair patches. All experimental results are publicly available on Github in order to facilitate future research on automatic program repair.**

I. Introduction

Automatic software repair aims to provide a fix to bugs in an automated way. Test-suite based repair, notably introduced by GenProg [1], is a widely studied family in program repair. In test-suite based repair, test suites are used as an executable specification of the program, with at least one failing test that reveals the bug. Test-suite based repair can be further divided into generate-and-validate techniques and synthesis-based techniques based on the employed patch generation strategy. Generate-and-validate techniques, such as GenProg [1], AE [2], Astor [3] and ACS [4], first generate as many patches as possible and then use the test suite to validate if the patch makes all tests pass. On the other hand, synthesis-based techniques such as AutoFix [5], SemFix [6], SearchRepair [7], Nopol [8], and Angelix [9], first extract constraints based on test suite execution and then synthesize a patch.

Recent automatic program repair papers tend to repeatedly use the same benchmarks. In program repair for C code, the ManyBugs benchmarks or its derivative is dominant [10]. In the context of program repair for Java, Defects4J is used in almost all evaluations of recent program repair approaches [11], [12], [13], [14], [15]. For technical research papers in ICSE'18, the majority of program repair papers on Java used Defects4J [16], [17], [18], [19]. However, only using the same benchmarks again and again pose a threat to the external validity of our research findings. The main threat is that the improvement that we now observe in the literature may only be valid for the benchmark under consideration but would not hold for other benchmarks. Even worse, those claimed improvements, if they only hold on the benchmark, may be decorrelated from for real usages by practitioners. Fortunately, the importance of external validity is acknowledged by many researchers [20], [21], [15], [14], [17], [22].

Problem: Research on program repair tends to repeatedly use the same benchmarks. This is a threat to the external validity for the results of our research community.

As building sound and conclusive empirical knowledge is key to science, reducing this major threat of external validity in the context of program repair is the main motivation of this paper. To reduce the threat, we aim at doing a novel empirical program repair experiment on a new and well-formed bug benchmark.

In this paper, we perform an automatic repair experiment on a benchmark called QuixBugs which was recently presented by Lin at al. [23]. QuixBugs is a program repair benchmark with 40 buggy algorithmic programs specified by tests cases. The buggy programs are both available in Python and Java. In our experiment, 1) We prepare Quixbugs for automated program repair in Java; 2) We select two representative approaches of test-suite based repair, Astor [3] and Nopol [8], and run them over all buggy programs. This results in 13 / 40 buggy programs repaired by 22 different plausible patches; 3) We assess the correctness of those 22 plausible patches by three different correctness assessment techniques. This identifies 12 overfitting patches. This novel experiment on a benchmark never used in a program repair context provides valuable findings that improves the external validity of program repair research. Our experiment sets a baseline for future research of automatic repair on QuixBugs.

To sum up, our contributions are:

- A new version of QuixBugs that is usable for automatic repair research for Java programs, extensive data about the content of QuixBugs (bug type, failure type, test profile).
- The confirmation of 2 empirical facts of program repair, improving their external validity: 1) the state-of-the-art of program repair tools produce overfitting patches. This confirms the results of [24], [25], [18]; 2) the state-of-the-

art of program repair tools also produce correct patches [26], [27]; 3) automatically generated tests can help to assess correctness of patches in scientific studies. This confirms the results of [28], [29], [30].

- Three new and important findings about program repair: 1) the state-of-the-art of program repair tools are able to repair programs with no passing tests at all (only failing test cases); 2) it is useful to design domain specific test generators to discard incorrect patches; and 3) a small number of automatically generated test cases is enough to identify incorrect patches in scientific studies.
- Experimental data that is made publicly available for facilitating this future research. Our results on QuixBugs set a baseline for further studies on QuixBugs. The data is anonymously available at [31].

The remainder of this paper is organized as follows. Section II presents how we prepare a new version of QuixBugs for the usage of automatic repair for Java programs. Section III presents four research questions (RQs) of our experiment, corresponding methodologies for these RQs, and reproducibility information of our experiment. Section IV presents our experiment results to answer the RQs. Section V discusses the threat of our study. Section VI discusses the related work of our experiment and Section VII concludes this paper.

II. Benchmark preparation

QuixBugs by Lin at al.[23] is a benchmark suite of 40 confirmed bugs from 40 classic algorithms. All bugs of QuixBugs were collected from Quixey Challenges [32], which consisted in giving human developers one minute to fix one program with a bug on a single line. The original QuixBugs benchmark contains: *a)* a set of 40 buggy programs available both in Python and in Java, *b)* for 31 out of 40 programs: JSON files with a set of inputs and expected outputs for each program, *c)* an engine that takes a program name, executes the program using the inputs from the corresponding JSON file, and prints the expected and obtained output, and *d)* for the remaining 9 out of 40 programs, a Java class that has encoded the inputs and outputs and prints the obtained output.

The initial version of QuixBugs was not usable for doing automatic repair in Java. Monperrus [33] states that, in the context of test-suite based repair, a "usable" benchmark must have [34]: 1) A clear, explicit, and not biased construction methodology (means the programs are well structured and public reviewed); 2) bugs and regression oracles. For test-suite adequate repair approach such as GenProg [1] and others [9], [8], [3], [2], [4], the oracles are the test suites; a failing test case means the presence of a bug while the absence of failing test means the correctness of the program w.r.t the inputs-outputs encoded in the test suite 3) Reality bugs (i.e., not seeded).

We summarize the problems of the initial version of QuixBugs as: *1)* it did not provide any test case; *2)* programs contained compilation errors (for 5 programs); *3)* incorrect values to test buggy programs (for 3 programs); *4)* missing test assertions (for 9 programs); and *5)* missing a Java reference ground truth version (for all programs).

To overcome the mentioned limitations that hamper its use by test-suite based repair approaches, we introduce a new version of QuixBugs supplemented with test cases for reproducing buggy behaviors and a reference ground truth version for evaluating automatic repair patches. This new version of QuixBugs was already peer reviewed and accepted by the QuixBugs authors and integrated to their public repository at Github. The steps we carried out for creating the new version are:

1) Fix uncompilable Java programs: By compiling the initial version Java programs of QuixBugs, we noticed that there were compile errors in some programs (e.g., *breadth_first_search*). Some compile errors are designed as part of buggy programs. However, most automatic repair tools need dynamic analysis of buggy programs. Hence, we need them all to be compilable and able to run the original buggy programs.

2) Fix incorrect test data: to test 31 out of 40 buggy Java programs, QuixBugs provides pairs of inputs and expected outputs written in JSON files. We have noticed that some expected outputs were incorrect. For instance, the expected output of test 9 from the program *knapsack* is actually "1735" instead of "1652". Other Programs containing incorrect values are: *sqrt* and *pascal*. Once we detected all incorrect inputs and outputs, we corrected them by proposing a new set of correct values.

3) Creation of JUnit tests from JSON files: QuixBugs uses a specific test driver based on JSON test cases. It executes the program using the inputs, and prints the output. For example, "[127, 7]" is one line of the JSON test of Bitcount program, where 127 is the input parameter of the program and the expected output is 7. The test driver simply prints out both expected output and actual output instead of using oracles. However, automatic repair tools usually expect JUnit tests as oracle specification: each test executes the program passing the inputs via parameters and then compares the obtained output with that one expected via assertions. Thus, we implement an automatic JUnit test generator to generate 224 JUnit tests (test methods in JUnit) for 31 programs from the given JSON files.

4) Creation of JUnit tests from ad hoc assertion-less tests: there are 9 out of 40 Java programs from QuixBugs that are tested through a simple ad hoc main method that starts with encoded inputs, calls the program using them as arguments, and finally prints the obtained output. This method is not usable by a test suite based program repair tool. Thus, we have manually rewritten those methods to produce 35 JUnit tests for these 9 programs. We have contributed to the main QuixBugs repository with those JUnit tests. In total, our preparation has resulted in 259 JUnit test methods over 40 programs.

5) Creation of Java reference versions: By default, QuixBugs does not provide a reference ground truth for Java. For comparing the automatically generated patches with the correct version, we added a reference ground truth version based on the provided correct Python version from QuixBugs.

To summarize, QuixBugs was initially not usable for automatic repair tools in Java. In this section, we presented the tasks we carried out to built a new version of QuixBugs that can be used to evaluate the effectiveness of test-suite adequate repair tools. The new version of QuixBugs contains JUnit test oracles and reference programs, it was public peer reviewed by the QuixBugs authors, organized with Travis and Gradle component. All those changes have already been contributed to the research community on the QuixBugs repository.

III. Experiment

We now present our experiment on the effectiveness of test-suite based repair approaches on the QuixBugs benchmark. The experiment covers several dimensions of automatic repair: benchmark analysis, repair effectiveness, patch correctness assessment. First, we list the Research Questions (RQs) of our work, we then describe the research methodology for each RQ in this experiment. Finally, we present our experiment environment details.

A. Research Questions

For this experiment on program repair for QuixBugs, we pose the following research questions (RQs):

- **RQ1: What are the main characteristics of the QuixBugs benchmark?** To answer this question, we compute and discuss statistics of QuixBugs, including the type of bug, lines of code (LOC), JUnit tests, code coverage, etc.
- **RQ2: How many buggy programs of QuixBugs can be automatically repaired with test-suite adequate patches?** In this experiment, we consider one kind of automatic repair called test-suite based repair. In test-suite based repair, a bug is said to be repaired if a patch makes all tests pass. We call that patch *test-suite adequate patch*. To answer this question, we present and discuss the patches generated by two notable and publicly-available automatic repair tools.
- **RQ3: To what extent are the synthesized patches correct?** A patch can be test-suite adequate i.e., passes all test cases, but yet incorrect. It is due to, for instance, weaknesses of the test suite (i.e., missing inputs or outputs). In that case, the patch is said to overfit the test cases [25]. On the contrary, a "correct" patch means that it is not overfitting to the input data and to the considered test cases. To evaluate the correctness of patches, we use three different techniques based on: 1) a search-based test generation tool [30]; 2) a custom domain-specific test generation tool [35]; and 3) manual analysis [26], [27].
- **RQ4: What are the strengths and weaknesses of the considered automated correctness assessment techniques for discarding incorrect patches?** To answer this question, we analyze in detail the automated correctness assessment techniques used in RQ3.

B. Methodology

*1) **RQ1: Exploration of the benchmark**:* For each program of QuixBugs, we gather or compute the following information.

a) Types of bugs: First, we give the type of bug. Recall that QuixBugs contains various types of bugs such as incorrect comparison operator, missing precondition, incorrect array slice, etc. The types of bugs come from [23].

b) Numerical characteristics: Second, we compute numerical characteristics: the lines of code (LOC) of the program, the number of passing JUnit tests, the number of failing JUnit tests, the test execution time, and the branch coverage. When calculating the lines of code for each program, we discard the comment and authorship. We rely on Cobertura [1], a tool based on jcoverage5, to calculate the branch coverage for each program. Existing study [36], [37] shows branch coverage is a better measurement than statement coverage. Here, we present branch coverage of each program.

c) Input domain: Third, we extract the program preconditions and the input domain of each program. The program preconditions are constraints for input domain. Input domain of each program makes sure that the program is meaningful for exercised inputs.

d) Failures: Last, we extract the failure types for all programs of QuixBugs.

*2) **RQ2: Repairability**:* To apply automated program repair on QuixBugs, we first need to select appropriate program repair tools. For this, we consider four criteria: a) the tool must handle Java programs as QuixBugs uses this programming language; b) the tool must implement a test-suite based repair approach; c) the tool must be publicly available; and d) the tool can be considered as representative for a family of repair techniques.

According to these criteria, we evaluate a set of automatic repair tools that target Java programs, including ACS [4], JAID [14], ssFix [15], CapGen [17], SKETCHFIX [19], S3 [38], Astor [3], Nopol [8], and HDRepair [39]. Eventually, we decide to use Astor [3] and Nopol [8] for their open-source and continued maintenance by their authors.

Both Astor and Nopol repair Java programs, are test-suite based, and are publicly available on Github. Moreover, they are representatives of two widely known families of program repair techniques: generate-and-validate techniques and synthesis-based techniques. In this paper, Astor is considered as the representative tool for generate-and-validate approaches and Nopol is considered as representative tool for synthesis-based approaches. Both Astor and Nopol take as input the source code of a buggy program and the corresponding test suite which contains at least one failing test case, and both generate one or more patches that make all test cases pass if such patches exist in their search space.

We run the two repair systems Astor and Nopol separately on the whole QuixBugs benchmark. As our goal is to find as many different patches as possible, we do not stop the repair process after finding the first patch. For the same reason, we run all available repair modes of Astor [3] and Nopol [8].

*3) **RQ3: Patch correctness assessment**:* As shown in previous research [16], [29], [30], assessing correctness of

[1]http://cobertura.github.io/cobertura/

automatically generated patches is a hard research question. This is still a hot research area and, to our knowledge, no consensus on the best solution has yet been achieved. The major problem is that a patch that passes a test suite may still be incorrect, as it may overfit to the provided test cases [25] but fail to generalize to other test cases. In that case, we say that the patch overfits the input data encoded in the test cases [30].

In our experiment, we consider three techniques for patch correctness assessment: 1) using generated tests by a search-based approach based on a reference version [30]; 2) using generated tests by a domain-specific generator based on a reference version [20]; and 3) manual analysis [26], [27].

a) Search-based test generator: Using automated test generation is one way for assessing patch correctness [29], [30], [28], [16]. In our study, the search-based test generator technique takes input as a reference version of buggy programs. The reference version is used as oracle which means the output of the reference program is the expected output. In this paper, we consider the state-of-the-art automated test generation tool Evosuite [40] for generating those new correctness assessment tests. We have chosen Evosuite according to the results of Shamshiri et al. [21], which have shown that Evosuite is the most effective tool for this use case.

For each of the 40 buggy programs in QuixBugs dataset, we invoke Evosuite a fixed number n of times, with the same configuration, against the reference version. Eventually, we obtain n different independent JUnit test suites for each program. Since Evosuite is a randomized algorithm, we take $n = 30$ for the best practice [41]. We further remove those generated tests that fail on the reference version [30] (due to limitation of Evosuite).

We execute these independent tests over patched programs and we mark as *Incorrect* a patch if there is at least one failing test case. If no generated test fails, we consider that the correctness is *Unknown* (and not *Correct* because we do not assess the behavior over the full input domain.

b) Domain-specific test generator: In order to strengthen our patch correctness assessment, we consider a second random testing approach called InputSampling. InputSampling, as an implementation of random testing [35] for QuixBugs, samples the input space according to a uniform distribution, and that uses the reference version as oracle [20]. If the reference version throws an exception on a generated input, the input is considered as invalid because not satisfying the preconditions, and is not kept.

For implementing InputSampling, we manually identify the domain of each input variable for programs in QuixBugs. The test generator is configured to sample the input space so as to get a fixed number of valid test cases with no exception (e.g. 50 or 100 times). To assess the effectiveness of these two automated test assessment techniques, we generate the similar number of tests with Evosuite. We assess patches with InputSampling tests as the same way with Evosuite tests.

c) Manual analysis: The third considered patch assessment technique is manual analysis [26], [27]. In our experiment, we manually analyze all generated patches. The manual analysis means that the first author compares the patch against the reference version, the finding is discussed with another author so that a consensus is reached. We use four labels to assess the patches: *Identical*, *Correct*, *Unknown* and *Incorrect*. A generated patch is deemed as correct if it is either identical to or semantically equivalent to the reference program. In our study of manual analysis, the criteria of labeling an *Incorrect* patch is to provide at least one test case to show the output of reference program is different from the output of patched program. We note that manual assessment is time-consuming as each patch differ takes ranging from several minutes to several hours of work dependent on the difficulty of the program. In addition, author bias is unavoidable when considering semantic equivalence between machine patches and human patches, thus at least two authors are needed to be involved in our manual assessment process.

Finally, we compute and discuss a metric for each patch to identify how many techniques discard it. We summarize the assessment results with three cases: 1) a patch not discarded by any technique is considered as the correct patch; 2) a patch discarded by all these three techniques is considered as incorrect patches without further analysis; and 3) a patch is only discarded by one or two techniques needs further analysis for the reason.

*4) **RQ4: Analysis of automated correctness assessment**:* We analyze in details the two considered automated patch correctness assessment techniques used in RQ3, which respectively refer to Evosuite and InputSampling. This research supports three dimensions : *a)* sensitivity, *b)* performance, and *c)* limitations.

a) Sensitivity: First, we compute and discuss how each technique is good at discarding incorrect patches depending on the number of generated tests. Both techniques are driven by one key numerical parameters: the number of trials for Evosuite, and the number of generated inputs for InputSampling. For each of the two considered techniques, we vary the configuration parameter and analyze the corresponding correctness assessment results. For instance, we compare the results by invoking Evosuite only once against the results by invoking Evosuite 5 times.

b) Performance: Also, we analyze the time costs of the two considered techniques.

c) Limitations: Eventually, we discuss the advantages and limitations of each technique.

C. Reproducibility information

We use below version of source code in our experiment. *a)* Nopol (commit a8e4681), *b)* Astor (commit b38d9ec), *c)* QuixBugs (commit 5c2a1a5), *d)* Evosuite 1.0.6, *e)* JDK 1.8.0_151, and *f)* Cobertura 2.7.

IV. Experimental Results

We now present and discuss the experimental results of our four research questions.

TABLE I: Descriptive statistics about the QuixBugs benchmark

Program name	Bug type	LOC	Passing tests	Failing tests	Code Coverage	Exe Sec.	Failure type
bitcount	A	10	0	9	100%	900	infinite loop
breadth_first_search	D	30	4	1	100%	<1	array index error
bucketsort	G	17	0	6	100%	<1	incorrect output
depth_first_search	N	23	4	1	100%	<1	stack overflow
detect_cycle	D	17	4	1	100%	<1	null pointer
find_first_in_sorted	C	22	4	3	90%	120	infinite loop(1) array index error(2)
find_in_sorted	E	19	5	2	100%	<1	stack overflow
flatten	J	18	1	6	83%	<1	stack overflow
gcd	F	10	0	5	100%	<1	stack overflow
get_factors	G	17	1	10	100%	<1	incorrect output
hanoi	B	53	0	7	100%	<1	incorrect output
is_valid_parenthesization	B	15	2	1	100%	<1	incorrect output
kheapsort	G	29	1	3	100%	<1	incorrect output
knapsac	C	30	4	6	100%	2	incorrect output
kth	B	25	3	4	100%	<1	array index error
lcs_length	G	48	1	8	95%	<1	incorrect output
levenshtein	E	15	1	6	100%	<1	incorrect output
lis	L	27	0	4	91%	<1	incorrect output
longest_common_subsequence	B	14	6	4	91%	<1	incorrect output
max_sublist_sum	B	13	2	4	100%	<1	incorrect output
mergesort	C	40	0	12	100%	<1	stack overflow
minimum_spanning_tree	B	67	0	3	72%	<1	concurrent modification
next_palindrome	G	28	4	1	87%	<1	incorrect output
next_permutation	C	32	0	8	83%	<1	incorrect output
pascal	B	29	1	4	100%	<1	array index error(3) incorrect output(1)
possible_change	D	23	0	9	100%	<1	array index error
powerset	B	24	1	4	100%	<1	incorrect output
quicksort	C	37	12	1	87%	<1	incorrect output
reverse_linked_list	M	12	1	2	100%	<1	null pointer
rpn_eval	F	28	3	3	100%	<1	incorrect output
shortest_path_length	K	49	2	2	92%	<1	incorrect output
shortest_path_lengths	F	31	0	4	100%	<1	incorrect output
shortest_paths	N	55	0	3	100%	<1	incorrect output
shunting_yard	N	31	0	4	100%	<1	incorrect output
sieve	J	35	1	5	75%	<1	incorrect output
sqrt	B	9	1	6	100%	360	infinite loop
subsequences	N	22	2	12	100%	<1	incorrect output
to_base	F	14	0	7	100%	<1	incorrect output
topological_ordering	B	25	0	3	100%	<1	incorrect output
wrap	N	22	0	5	75%	<1	incorrect output
Total	-	1034	70	189	1400	-	-

Legend about bug type [23]:
A:Incorrect Assignment operator, B:Incorrect variable, C:Incorrect comparison operator,
D:Missing condition, E:Missing/added+1, F:Variable swap G:Incorrect array slice H:Variable prepend
I:Incorrect data structure constant, J:Incorrect method called, K:Incorrect field dereference,
L:Missing arithmetic expression, M:Missing function call, N:Missing line

A. *RQ1: QuixBugs benchmark analysis*

Table I presents characteristics, including the numerical statistics and failure test symptoms of QuixBugs. All 40 buggy programs of QuixBugs are implementations of classic algorithms, such as different sorting algorithms *bucketsort, mergesort, quicksort* and etc. Program names are given in the first column in alphabetical order. The second column presents the type of bug of each program. There are 14 different bug types, what can be seen is that the bug types are diverse: some bugs could be repaired by replacing operators (A, C); other bugs could be repaired by adding conditional statements (D); repaired by inserting a new line of code (N), etc. Thus, if one repair approach can repair most of buggy programs in QuixBugs, it would mean that this approach is general in essence.

The third column gives the line of code (LOC) per program. We see that they range from 9 to 67 lines, which can be considered as small. However, we note that 14 are recursive programs and 13 programs contain nested loops. It means that, despite a small program size, the time complexity or space complexity of those programs is sometimes non-trivial. Table I also summarizes the statistics about JUnit tests: the fourth and fifth columns present the number of passing tests and failing tests. The six column gives the branch coverage information of JUnit tests. The seventh column presents the test execution time for each program. For instance, program bitcount has no passing test and 9 failing tests, that execute in 900 seconds (all failing tests are infinite loops, which are stopped with the JUnit timeout to 60s).

We see that all programs from the new version of the QuixBugs that we introduced have at least one failing JUnit test to expose the bug, which means that the prerequisite of test-suite repair is met. There are 15 programs with no passing tests. All benchmarks of the literature, to our knowledge, contain at least one passing test. Passing tests are important for repair approaches to model the expected behavior of the program, which means without these passing tests, synthesis based approaches, such as Nopol, have degenerated synthesis problems when repairing QuixBugs programs.

There are 37 / 40 programs whose tests run in less than 2 seconds, which suggests that program repair will be fast. For those 3 programs where the bug triggers an infinite loop, the tests timeout after 60 seconds, which explains the 3 large execution time values of programs (*bitcount, find_first_in_sorted and sqrt*).

The last column presents the failure types from 6 different types. They are 26 programs with incorrect output failures, 5 programs with stack overflow failures, 5 programs index out of bounds failures, 3 programs with infinite loop failures, 2 programs with null pointer failures, and 1 programs with current modification failure. For two programs, *find_first_in_sorted* and *pascal*, the corresponding test cases expose two different failures. This is interesting because it shows that beyond the bug type diversity previously discussed, QuixBugs also has a failure type diversity.

Besides the above characteristics of QuixBugs, preconditions of each program are important. The preconditions are the constraint for inputs. For example, the precondition of program *get_factor* is an integer value greater than 1 because the purpose of this program is to factor an integer value using trial division. Obviously, the program is meaningless when the input is a negative integer. All preconditions for all programs are given in the online appendix [31].

Comparing the benchmarks of literatures [10], [42], [43], [44], we found three unique characteristics in QuixBugs: *1)* There is a focus on algorithmic tasks: *sorting algorithms*, *search algorithms*, *Towers of Hanoi puzzle*, etc; whose time complexity or space complexity is non-trivial. We note that 14 / 40 programs contain recursion. *2)* There are 15 / 40 programs

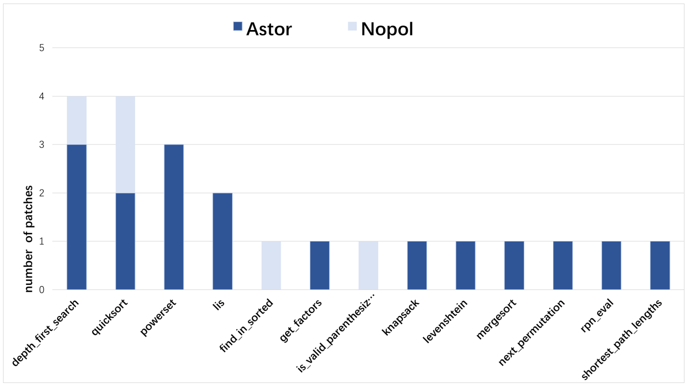

Fig. 1: QuixBugs repair results by Astor and Nopol

with only failing tests. To our knowledge, all other benchmarks in the literature always contain at least one passing test; *3)* The benchmark contains 3 infinite loops and 5 stack overflow failures, which is uncommon in benchmarks. Thus using QuixBugs for program repair will give new insights about the successes and limitations of current repair tools.

Answer to RQ1: QuixBugs is a valuable dataset for studying program repair. It has a large diversity of bug types, as well as a diversity of failure types. It contains buggy programs with unique characteristics compared to existing program repair benchmarks: 1) complex algorithmic tasks, 2) programs with no passing tests and 3) programs with infinite loops.

B. RQ2: Test-Suite Adequate Patches

In this RQ, we present the automatic repair results by employing two representative test-suite based approaches: Astor and Nopol. In total, we have performed 200 repair attempts with Astor (40 programs $\times$ 5 repair modes) and 80 repair attempts in Nopol (40 programs $\times$ 2 repair modes). All repair attempts have a timeout of 5 minutes. This means that the experiment time is bounded by $(200+80)\times 5 = 1400$ minutes, i.e., 23 hours.

We present our repair results in Figure 1. In total, we have obtained 22 patches generated by the two considered repair approaches, those patches repair 13 different buggy programs. Astor generates 17 patches for 11 programs while Nopol generates 5 patches for 4 programs. Two programs are repaired by both Astor and Nopol (*depth_first_search* and *quicksort*). We have more patches than repaired programs because: *1)* some bugs are repaired by both Astor and Nopol (e.g. *depth_first_search*), *2)* some bugs are repaired by different modes (e.g. *lis* is repaired in two different modes of Astor), and *3)* for a given repair mode, we configured the tools to output all patches from the search space (e.g. three patches for *powerset* are generated by one mode of Astor).

Listing 1 gives a code snippet of a *quicksort* patch generated by Astor. Both Astor and Nopol generate pretty-printing patches like this. The bug type of *quicksort* is C (*Incorrect comparison operator*). The "-" means the patch identifies this line of code is buggy and deletes it from sources code, instead, the patch adds a new line of statement where there is a "+" in the beginning. In this case, Astor generates patch by modifying conditions in if block, using "*if (true)*" to replace "*if (x>pivot)*". The commented code gives the human reference, in this case "*if (x>=pivot)*". In this context, we consider the patch correct since it is semantically the same with the reference version.

Listing 1: Quicksort patch generated by Astor

```
   if (x < pivot) {
           lesser.add(x);
     else
    //reference version: if(x>=pivot){
-    if (x > pivot) {
+    if (true) {
         greater.add(x);
     }
```

We discuss the bug type of programs in Section IV-A. From the repair result, there are 7 / 14 types of bugs that are fixed by the considered automatic repair approaches which is encouraging. They are B (*is_valid_parenthesization* and *powerset*), C (*knapsack*, *mergesort*, *next_permutation* and *quicksort*), E (*find_in_sorted* and *levenshtein*), F (*rpn_eval* and *shortest_path_lengths*), G (*get_factors*), L (*lis*), N (*depth_first_search*). Interestingly, the bug type C is the most repaired bug types (i.e. for 4 programs), the reason being that Nopol is specialized in repairing those bugs, and Astor has one specific mode for mutation repair of comparison operator [45].

Then we analyze how the tests impact the considered repair approaches. For the 13 repaired programs, 4 of them have 0 passing test. To our knowledge, all benchmarks of the literature contain at least one passing test case. Here, our experiment shows that program repair with only failing tests may be successful. All these 4 programs with no passing tests that are repaired by Astor. The reason is Astor is a generate-and-validate technique which not requires positive tests for synthesizing a patch. On the contrary, since Nopol is synthesis based, the absence of passing tests creates a degenerated synthesis problem. To this extent, Quixbugs is more appropriate for validating generate-and-validate techniques than for validating synthesis based ones.

Finally, we have aggregated the failure types of the patched programs: they are 10 of *incorrect output* and 3 of *stack overflow* errors. While this shows that the considered repair tools handle two kinds of failure types, it also shows that *infinite loops*, *null pointer exceptions*, c*concurrent modification exceptions* and *array index errors*, are not well handled by either Astor or Nopol.

In a summary, Astor is globally better at finding test-suite adequate patches. The reason is that it has a generic search space, while Nopol has a specific search space for conditional bugs. In addition, Astor is a generate and validate system which is able to generate patches for no passing tests program, while Nopol has limitation for generating patches in this case.

Answer to RQ2: 13 / 40 QuixBugs programs are repaired with test-suite adequate patches synthesized by the considered repair systems Astor and Nopol. Those test-suite adequate patches cover 7 bug types. Four programs can be repaired despite the absence of passing tests. Overall, this first ever program repair experiment on Quixbugs sets a baseline for future program repair research based on this benchmark.

C. RQ3: Patch Correctness Assessment

In this RQ, we answer how many patches obtained in RQ2 are correct beyond test-suite adequacy. For doing so, we use three different techniques (see Section III-B3).

We first present our assessment result in Table II, which gives 22 generated patches, together with the correctness assessment results. The first column gives the program name and corresponding patch number. The numbers in the second and third columns give the result assessed by the two considered techniques Evosuite and InputSampling, respectively. We note that those generated tests used to assess the correctness of patches all have high branch coverage, the average branch coverage is 90%. If the patch makes one or more test fail, we consider it as *Incorrect*, otherwise, *Unknown*. The form of *x / y* denotes the patch makes x generated tests fail out of total generated tests number y. The fourth column gives the manual assessment labels. Recall the manual assessment in Section III-B3, we give the four labels: Identical, Correct, Unknown and Incorrect. Figure 2 shows all three techniques discard 4 patches in common (in circle), 4 patches discarded by both Evosuite and manual analysis, 3 patches discarded by both InputSampling and manual analysis, and 1 additional patch only discarded by manual analysis.

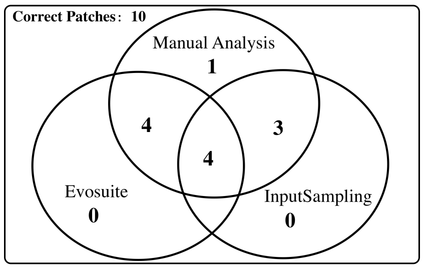

Fig. 2: Patches identified as incorrect by the three considered assessment techniques

The remaining 10 patches are classified as *Correct*. Three patches are identical to the reference version (*knapsack, levenshetein and rpn_eval*), the other 7 patches are semantically equivalent to the reference version (two patches for *lis*, four patches for *quicksort* and one patch for *mergesort*). Interestingly, two programs with no passing tests are correctly repaired (*lis* and *mergesort*). To our knowledge, the fact that program repair can succeed with only failing tests has never been studied or reported in the literature.

TABLE II: Patch correctness assessment results for the three considered techniques. x/y denotes that x generated tests identify a patch as incorrect out of y generated tests

Patches	Evosuite analysis	InputSampling analysis	Manual analysis
depth_first_search (Astor)−p0	Incorrect 30/120	Incorrect 154/300	Incorrect
depth_first_search (Astor)−p1	Incorrect 32/120	Unknown 0/300	Incorrect
depth_first_search (Astor)−p2	Incorrect 32/120	Unknown 0/300	Incorrect
depth_first_search (Nopol)	Incorrect 32/120	Unknown 0/300	Incorrect
find_in_sorted (Nopol)	Incorrect 142/316	Incorrect 168/300	Incorrect
get_factors (Astor)	Incorrect 27/119	Unknown 0/300	Incorrect
is_valid_parenthsization (Nopol)	Incorrect 32/150	Incorrect 61/300	Incorrect
knapsack(Astor)	Unknown 0/208	Unknown 0/300	Identical
levenshetein (Astor)	Unknown 0/120	Unknown 0/300	Identical
lis (Astor)−p0	Unknown 0/126	Unknown 0/300	Correct
lis (Astor)−p1	Unknown 0/126	Unknown 0/300	Correct
mergesort(Astor)	Unknown 0/241	Unknown 0/300	Correct
next_permutation(Astor)	Unknown 0/222	Unknown 0/300	Incorrect
powerset(Astor)−p0	Unknown 0/87	Incorrect 224/300	Incorrect
powerset(Astor)−p1	Unknown 0/87	Incorrect 300/300	Incorrect
powerset(Astor)−p2	Unknown 0/87	Incorrect 224/300	Incorrect
quicksort (Astor)−p0	Unknown 0/169	Unknown 0/300	Correct
quicksort (Astor)−p1	Unknown 0/169	Unknown 0/300	Correct
quicksort (Nopol)−p2	Unknown 0/169	Unknown 0/300	Correct
quicksort (Nopol)−p3	Unknown 0/169	Unknown 0/300	Correct
rpn_eval (Astor)	Unknown 0/218	Unknown 0/300	Identical
shortest_path_lengths (Astor)	Incorrect 16/167	Incorrect 297/300	Incorrect
# discarded patches	8	7	12

Evosuite We spent 48.7 hours to generate 30 different test suites with Evosuite for all 40 repaired QuixBugs programs, averaging 2.5 minutes for generating one test suite for each program. In different test suite, the number of test method is ranging from 3 to 14, i.e. approx 10 tests generated for each test suite. To address the noted problem that test cases generated by Evosuite could fail on the version used for generating them, we manually remove the failing tests. We remove tests from 6 out of 13 repaired programs, they are: *find_in_sorted*, *get_factors*, *knapsack*, *levenshtein*, *next_permutation*, *shortest_path_lengths*. After the preparation work, we ran the 22 patched programs over the corresponding 13 different test suites generated by Evosuite. As a result, 8 patches out of 22 patches are identified as *Incorrect*.

InputSampling Considering we invoked 30 runs of Evosuite, i.e. approx $(10 \times 30) = 300$ tests for each program, thus we generated 300 tests by InputSampling to compare similar number of tests with Evosuite. Each of the 22 patched pro-

TABLE III: Case studies of patch correctness assessment by the three considered techniques

Listing 2: Patch for find_in_sorted by Astor, discarded by Evosuite and InputSampling

```
   int mid = start + (end − start) / 2;
− if (x < arr[mid]) {
+ if (mid <= 2 || (x != arr[mid])
+ && (!arr.length < (arr[mid])))) {
     return binsearch(arr, x, start, mid);
  }else if (x > arr[mid]) {
     return binsearch(arr, x, mid, end);
```

Listing 3: Patch for get_factors by Astor, discarded by Evosuite but not InputSampling

```
   int max = (int) (Math.sqrt(n) + 1.0);
− for (int i = 2; i < max; i++) {
+ for (int i = 2; (n % n) == 0; i++) {
       if ((n % i) == 0) {
    ...}}
  return new ArrayList<Integer>();
```

Listing 4: Patch of powerset by Astor, discarded by InputSampling but not Evosuite

```
+ output.addAll(rest_subsets);
  for (ArrayList subset : rest_subsets) {
       ... }
  return output;
```

Listing 5: Patch for next_permutation by Astor, only discarded by manual analysis

```
   for (int j=perm.size()−1; j!=i; j−−) {
−    if ((perm.get(j)) < (perm.get(i))) {
+    if ((perm.get(j)) >= (perm.get(i))) {
```

grams was executed against the corresponding 300 tests to be assessed the correctness. As a result, 7 patches out of 22 are discarded. All these 7 patches assessment labels are consistent with manual analysis labels.

Manual analysis As a result, shown in the fourth column of Table II, 12 patches are labeled as *Incorrect*, and the other 10 patches are either identical or semantically correct. Together, Evosuite and InputSampling discard 11 different patches. But manual analysis identifies one additional patch for program *next_permutation* which not discarded by the two considered automated techniques.

Here, Table III presents the code snippets of four patches that representing difference assessment results given by the three considered assessment techniques. Now we discuss each of them.

a) A patch correctly discarded by both Evosuite and InputSampling: Listing 2 shows the Nopol patch of program *find_in_sorted*, which is discarded by 142 / 316 tests generated by Evosuite and 168 / 300 tests generated by InputSampling. The patch is incorrect because Nopol's synthesis exploits a spurious relation between variable *arr* and *mid* that only holds for the two original failing test cases. Evosuite and InputSampling correctly identify an input data that breaks this spurious relation between two variables, and thus appropriately discards this overfitting patch.

b) A patch discarded by Evosuite but not InputSampling: Listing 3 gives the Astor patch for *get_factors*, which is discarded by Evosuite but missed by InputSampling. The patch changes the stopping condition of the for loop by using (*n%n==0*). This expression always evaluates to true, which means the for loop is never stopped by the increment of variable i, however it is stopped with a return statement inside the loop.

This patch is discarded by Evosuite because of a division-by-zero exception, however, InputSampling fails to discard it. Program *get_factors* takes an integer as input. For this program, InputSampling randomly generates an Integer from range 0 to 1000 according to the precondition, but it fails to generate "0" in the 300 generated tests. On the contrary, Evosuite succeeds in using 0 as input and reproducing this arithmetic exception that discards the patch. We note that InputSampling can trivially be extended to always consider special values such as "0".

c) A patch discarded by InputSampling but not Evosuite: Listing 4 gives one of the patches of *powerset* that is discarded by InputSampling but not Evosuite. The reason is that Evosuite is not effective on this program, it only generates 90 test methods over all 30 runs. All generated tests for program *powerset* lack assertions, and the generated tests always pass in this case. On the contrary InputSampling samples the right inputs to discard the patch.

d) A patch discarded only by manual analysis: Listing 5 gives the patch of *next_permutation* by Astor. The patch uses the ">=" operator to fix the bug. The bug is present in an if condition which compares the position of two elements in a list *perm*. This patch is not identical to the reference version, where the used operator *">"*. Here, manual analysis reveals that this patch is incorrect: if the list *perm* contains the same values, the behavior of the patched program would be different from the one of the reference program. To prove our analysis, we have crafted such an input and indeed got two different output values between the reference program and the patched program, confirming our analysis (all counter-examples identified during manual analysis are available in our online appendix [31]). This special input was not generated by either Evosuite or InputSampling, and as such the patch was not discarded. This case shows that automated correctness assessment cannot fully replace manual analysis.

> Answer to RQ3: By combining all patch assessment techniques together, 12 out of 22 patches are shown to be incorrect due to overfitting. By using the reference version as oracle, the search-based tool Evosuite discards 8 incorrect patches, the random testing tool InputSampling discards 7 patches, and the manual analysis discards 12 patches. This shows that: 1) test suite generation based on the reference version is useful for discarding incorrect patches in scientific studies but 2) it is not enough because it misses certain overfitting patches. As a result, we recommend to always complement automated correctness assessment with manual analysis.

D. *RQ4: Analysis of automated correctness assessment*

In this RQ, we study the sensitivity, performance and limitations of the two considered automated correctness assessment techniques based on Evosuite and InputSampling.

1) Sensitivity: Table IV compares the assessment results of 22 generated patches by different number of generated tests for both techniques. The ✗ denotes that the patch is discarded. The second, third and fourth column give the assessment results for all patches by invoking Evosuite once, 5 times and 30 times. The fifth, sixth and the last column give the assessment results for all patches based on respectively 10 tests, 50 tests and 300 tests generated by InputSampling.

We see with the second column that 7 patches are discarded when we invoke Evosuite only once, while 8 patches are discarded when running it 5 times, resulting in 50 tests, and 30 times with equivalent 300 tests. On the other hand, 6 patches are discarded by the first 10 InputSampling tests and 7 patches are discarded (one more) by generating 50 InputSampling tests and 300 InputSampling tests. All discarded patches are consistent with manual analysis labels (Incorrect) in RQ3.

This experiment shows that the more tests, the more incorrect patches are discarded, and this holds for both techniques. To that extent, we recommend that researchers use a budget based approach: generated as many tests within the time period they can afford. This table also clearly shows that both approaches are really complementary, they catch different incorrect patches.

2) Performance: Evosuite takes 975 minutes to generate 30 test suites for all 13 repaired programs, i.e. approx 15 seconds per test per program. InputSampling takes less than 15 minutes to generate 3900 test methods for them; which takes 0.23 second to generate a test per program, which is 65x faster. Considering again the budget-based approach mentioned in Section IV-D1, if one is short in time for assessing correctness, InputSampling is a more appropriate choice for small programs.

TABLE IV: Results of patch assessment by varying number of tests. A ✗ denotes the patch is identified as incorrect.

	Evosuit			**InputSampling**		
	10 tests ≈1 run	**50 tests ≈5 runs**	**300 tests ≈30 runs**	**10 tests**	**50 tests**	**300 tests**
depth_first_search(Astor)-p0	✗	✗	✗	✗	✗	✗
depth_first_search(Astor)-p1	✗	✗	✗			
depth_first_search(Astor)-p2	✗	✗	✗			
depth_first_search(Nopol)	✗	✗	✗			
find_in_sorted(Nopol)	✗	✗	✗	✗	✗	✗
get_factors(Astor)	✗	✗	✗			
is_valid_parenthsization(Nopol)	✗	✗	✗		✗	✗
knapsack(Astor)						
levenshetein(Astor)						
lis(Astor)-p0						
lis(Astor)-p1						
mergesort(Astor)						
next_permutation(Astor)						
powerset(Astor)-p0				✗	✗	✗
powerset(Astor)-p1				✗	✗	✗
powerset(Astor)-p2				✗	✗	✗
quicksort(Astor)-p0						
quicksort(Astor)-p1						
quicksort(Nopol)-p2						
quicksort(Nopol)-p3						
rpn_eval(Astor)						
shortest_path_lengths(Astor)		✗	✗	✗	✗	✗
# Total discarded patches	7	8	8	6	7	7

3) Limitations: According to our experiments, the limitations of Evosuite for automated correctness assessment are: *a)* since tests are generated to maximize coverage, some tests do not contain assertions, and as such are bad at catching behavioral changes (see the case of *powerset* discussed in IV-C0c); *b)* Evosuite does not support specifying the input domain, which means many Evosuite generated tests are useless if they trivially fail to meet the input precondition; *c)* Evosuite generates some test cases that fail, even on the version used for generating them, this requires additional work.

Our experiment gives interesting insights about InputSampling: *a)* as InputSampling samples the inputs randomly, it could miss some special values due to bad chance (see the case study of *get_factors* discussed in Section IV-C0b); *b)* InputSampling requires manual work to write the domain specific generators.

> Answer to RQ4: There is no silver-bullet for automated correctness assessment. Our experiments show that: 1) For both techniques, more automatically generated test cases lead to better assessment and to discarding more incorrect patches; 2) Search-based correctness assessment is much slower than input sampling; 3) The considered techniques do not discard the same patches, there is little overlap in the discarded patches. Consequently, we recommend that future research in program repair considers using several test generation techniques in conjunction, in order to maximize the validity of automated correctness assessment of their patches.

V. THREATS TO VALIDITY

The major threat in our study lies in the manual correctness assessment, which may result in misclassification due to lack of expertise or mistakes. This threat holds for all program repair papers based on manual assessment. The best mitigation to this threat is to make patches publicly-available for other researchers to further assess them: this is what we have done in our open-science repository [31].

The second threat is about the construct validity. For our experiment, the considered tools are not perfect and they surely contain bugs which prevent them from finding all possible patches. For this reason, the results we have reported are likely an under-estimation of the repairability of QuixBugs using automatic program repair. Future studies on QuixBugs will address this threat and identify new patches.

VI. RELATED WORK

1) Current Datasets of Bugs: The widely used benchmarks in automatic program repair research include Introclass [10], ManyBugs [10], SIR [44], Codeflaws [46], Defects4J [43], and IntroClassJava [42].

IntroClass Smith et al. [25] evaluate overfitting patches generated by GenProg and TrpAutoRepair on IntroClass. Le et al. [18] systematically characterize the nature of overfitting in semantics-based automatic program repair on the IntroClass and Codeflaws benchmarks. Ke et al. [7] evaluate SearchRepair on IntroClass. Jiang et al. [47] propose to use metamorphic relations as a repair oracle and evaluate their approach on the Introclass benchmark.

ManyBugs Qi et al. [26] evaluate three existing generate-and-validate repair techniques on ManyBugs. Mechtaev et al. [9] propose Angelix and evaluate it on ManyBugs. Long and Rinard [22] propose SPR and evaluate it on ManyBugs, and they also evaluate their another work Prophet [48] on ManyBugs.

SIR Stratis and Rajan [49] propose and evaluate a novel approach to improve instruction locality across test case runs on SIR. Nguyen et al. [6] propose SemFix and evaluate it on SIR. An empirical study by Kong et al. [50] compares different repair systems: GenProg, RSRepair, and AE in an experiment based on the SIR benchmark.

Codeflaws Papadakis et al. [51] collect and analyze mutant quality indicators based on Codeflaws. Chekam et al.[52] propose a new perspective to tackle the fault revelation mutant selection and evaluate their work on Codeflaws.

Defects4J Martinez et al. [27] and Yu et al. [30] report their experiments using Defects4J for evaluating the effectiveness of automatic repair techniques. Xin and Reisse [29] propose DiffTGen to identify incorrect Patches generated for Defects4J bugs. In addition, they also use Defects4J to evaluate their proposed automated program repair technique ssFix [13]. Xiong et al. [4] propose the ACS (Accurate Condition Synthesis) repair system and evaluate it on four projects of the Defects4J benchmark. Wen et al. [17] propose CapGen, a context-aware patch generation technique and evaluate this technique on the Defects4J benchmark. Chen et al. [14] propose JAID, a novel APR technique for Java programs, which is capable of constructing detailed state abstractions and it is evaluated on the Defects4J benchmark. Saha et al. [15] evaluate their proposed repair technique ELIXIR on two benchmarks, one of them is Defects4J.

IntroClassJava Wen at al. [17] propose CapGen, a context-aware patch generation technique and evaluate their technique on IntroClassJava. Le et al. propose and evaluate S3 on IntroClassJava [38], and they also evaluate another approach called JFIX [53] on IntroClassJava. Zemín et al. [54] check generated patches produced by automated repair tools and evaluate their work on IntroClassJava. To the best of our knowledge, QuixBugs has never been used in a program repair experiment until our study.

2) Patch correctness assessment: Synthesizing new inputs for patch correctness assessment has been studied a couple of times. Xin and Risse [29] propose DiffTGen that adds new generated tests by Evosuite to the original test suite to prevent the repair technique from generating a similar overfitting patch again. Yang et al. [28] propose Opad to filter out incorrect patches by augmenting existing test cases in C programs with memory-safety oracles and by creating new test cases with fuzz testing. Recent work by Xiong et. al. [16] determine the patch correctness by comparing the execution similarity of the original and new generated tests before and after the patch. These three techniques aim to prevent the generation of overfitting patches. On the contrary, our work focuses on assessing patch correctness for scientific studies. The most related work is with the study by Yu et al. [30], showing that test case generation can help to identify overfitting patches. Our study builds on their methodology, but on a new benchmark.

VII. CONCLUSION

We have presented a novel program repair experiment over QuixBugs, a new benchmark of bugs by MIT. Our experiment reduces the threat to external validity to major empirical findings about program repair: the state-of-art program repair tools produce many overfitting patches, and automatically generated test cases can help to assess patch correctness in scientific studies. Our experiment also enables us to deepen our understanding of program repair: 1) it is possible to repair programs with no passing tests at all (only failing test cases); 2) it is useful to design domain specific test generators to discard incorrect patches; and 3) a small number of automatically generated test cases is enough to identify incorrect patches in scientific studies. Our future work concerns the creation of a novel benchmark that would contain valuable bugs for program repair research.

VIII. ACKNOWLEDGMENTS

We would like to thank James Koppel for his insight and feedback on this paper. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. volume 38, Piscataway, NJ, USA, January 2012. IEEE Press.

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

[3] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA, Demonstration Track. 441444.*, 2016.

[4] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of ICSE*, 2017.

[5] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, 2010.

[6] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *ICSE*, 2013.

[7] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[8] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clment, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. 2016.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

[10] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. 2015.

[11] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer ANSYMO. Fine-tuning spectrum based fault localisation with frequent method item sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[12] Xuan-Bach D. Le. Towards efficient and effective automatic program repair. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[13] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[14] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[15] Ripon K. Saha, Yingjun Lyu, and Hiroaki Yoshida. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[16] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. In *Proceedings of ICSE*, 2018.

[17] Ming Wen, Junjie Chen, rongxin wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of ICSE*, 2018.

[18] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues.

[19] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of ICSE*, 2018.

[20] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of ICSE*, 2018.

[21] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[22] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015.

[23] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity Pages 55-56.*, 2017.

[24] Qi Xin. Towards addressing the patch overfitting problem. In *Software Engineering Companion (ICSE), 2017 IEEE/ACM 39th International Conference on*, 2017.

[25] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015.

[26] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 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 2015, 2015.

[27] Matias Martinez, Thomas Durieux, Jifeng Xuan, Romain Sommerard, and Martin Monperrus. Automatic repair of real bugs: An experience report on the defects4j dataset.

[28] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 48, 2017 (ESEC/FSE17), 11 pages*, 2017.

[29] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, 2017.

[30] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *CoRR*, abs/1703.00198, 2017.

[31] Github repository of our study is available online. *https://github.com/KTH/quixbugs-experiment*, 2018.

[32] Ryan Lawler. How do you hire great engineers? give them a challenge. In *https://gigaom.com/2012/01/19/quixey-challenge/*, 2012.

[33] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[34] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[35] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 2012.

[36] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[37] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering*, pages 597–608. IEEE Press, 2017.

[38] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[39] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 213–224. IEEE, 2016.

[40] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE '11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011.

[41] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[42] Thomas Durieux and Martin Monperrus. IntroClassJava: A Benchmark

of 297 Small and Buggy Java Programs. Research Report hal-01272126, Universite Lille 1, 2016.

[43] Ren Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of ISSTA*, 2014.

[44] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*

[45] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[46] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182, 2017.

[47] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. A metamorphic testing approach for supporting program repair without the need for a test oracle. 2016.

[48] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, 2016.

[49] Panagiotis Stratis and Ajitha Rajan. Test case permutation to improve execution time. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[50] X. Kong, L. Zhang, W. E. Wong, and B. Li. Experience report: How do techniques, programs, and tests impact automated program repair? In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 194–204, 2015.

[51] Mike Papadakis, Titcheu Thierry Chekam, and Yves Le Traon. Mutant quality indicators. *http://hdl.handle.net/10993/34352*, 2018.

[52] Titcheu Thierry Chekam, Mike Papadakis, Tegawend Franois D Assise Bissyande, and Yves Le Traon. Reference : Predicting the fault revelation utility of mutants. 2018.

[53] Xuan-Bach D. Le, Duc-Hiep Chu, and Lo. Jfix: Semantics-based repair of java programs via symbolic pathfinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[54] Luciano Zemín, Simón Gutiérrez Brida, Ariel Godio, César Cornejo, Renzo Degiovanni, Germán Regis, Nazareno Aguirre, and Marcelo Frias. An analysis of the suitability of test-based patch acceptance criteria. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, SBST '17, 2017.