A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark

He Ye*, Matias Martinez†, Thomas Durieux‡, Martin Monperrus*
*KTH Royal Institute of Technology, Sweden {heye, martin.monperrus}@kth.se
†University of Valenciennes, France matias.martinez@univ-valenciennes.fr
‡INESC-ID, Portugal thomas@durieux.me

***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 never been studied in the context of program repair. In this study, we report on the characteristics of QuixBugs, and study five repair systems, Arja, Astor, Nopol, NPEfix and RSRepair, which are representatives of generate-and-validate repair techniques and synthesis repair techniques. We propose three patch correctness assessment techniques to comprehensively study overfitting and incorrect patches. Our key results are: 1) 15 / 40 buggy programs in the QuixBugs can be repaired with a test-suite adequate patch; 2) a total of 64 plausible patches for those 15 buggy programs in the QuixBugs are present in the search space of the considered tools; 3) the three patch assessment techniques discard in total 33 / 64 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 [17], 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, Astor, CapGen, 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, SemFix, and Nopol first extract constraints based on test suite execution and then synthesize a patch[26][9].

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 [16]. In the context of program repair for Java, Defects4J is used in almost all evaluations of recent program repair approaches. For technical research papers in ICSE'18, the majority of program repair papers on Java used Defects4J [10], [34], [37]. However, repeatedly using the same benchmarks 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.

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. [18]. 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 five representative approaches of test-suite based repair, Arja [42], Astor [21], Nopol [39], NPEFix [4] and the Java implementation of RSRepair [29], and run them over all buggy programs. This results in 15 / 40 buggy programs are repaired by 64 different plausible patches; 3) We assess the correctness of those 64 plausible patches by three different correctness assessment techniques. This identifies 33 overfitting patches. This novel experiment on a benchmark never used in a program repair context provides valuable findings that improve 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 on Java programs, together with extensive data about the characteristics of QuixBugs.
- The confirmation of 2 empirical facts of program repair, improving their external validity: 1) the state-of-the-art program repair tools produce overfitting patches, this confirms the results of [35], [31], [15]; 2) the state-of-the-art program repair tools also produce correct patches [29], [20]; 3) automatically generated tests can help to

assess the correctness of patches in scientific studies, this confirms the results of [40], [36], [41].

- Three new and important findings about program repair: 1) the state-of-the-art program repair tools are able to repair programs with only failing test cases and no passing tests at all; 2) it is useful to design program 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 [1]. Our results on QuixBugs set a baseline for further studies on QuixBugs.

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.[18] is a benchmark of 40 confirmed bugs from 40 classic algorithms. All bugs of QuixBugs were collected from Quixey Challenges [13], 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.

However, the initial version of QuixBugs was not usable for doing automatic repair in Java. Monperrus [26] states that, in the context of test-suite based repair, a "usable" benchmark must have [25]: 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 [17], 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 regression oracle; *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 were 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 in programs *knapsack* , *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. 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.

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?
- RQ2: How many buggy programs of QuixBugs can be automatically repaired with test-suite adequate patches?
- RQ3: To what extent are the generated patches correct?
- RQ4: What are the strengths and weaknesses of the considered automated correctness assessment techniques for discarding incorrect patches?

In RQ1, we are interested in the statistics of QuixBugs, including the type of bug, lines of code (LOC), JUnit tests, code coverage, etc. In RQ2, 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 focus on how many test-suite adequate patch could be generated by the state-of-the-art test suite based repair approaches. In RQ3 we are interested in how many patches generated by RQ2 are actually correct,i.e. not overfitting. A test-suite adequate patch i.e., passes all test cases, but yet incorrect. It is due to the weaknesses of the test suites, which are not able to completely specify the program behaviors. In that case, the patch passes all the given test cases but fail to be generalize is called overfitting patches [31]. 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 patch correctness techniques based on: 1) a search-based test generation tool [41]; 2) a custom program specific test generation tool [2]; and 3) manual analysis [20]. In RQ4, we analyze in detail the effectiveness of automated patch correctness assessment techniques used in RQ3.

B. Methodology

This section presents experiment methodology to answer the four research questions.

1) RQ1: Exploration of the QuixBugs Benchmark: For each buggy program of QuixBugs, we gather or compute the following information:

a) Types of Bugs: We give the type of bug. Recall that QuixBugs contains various types of bugs such as incorrect comparison operator, incorrect array slice, etc. The types of bugs come from [18].

b) Numerical Characteristics: We compute numerical characteristics: the lines of code (LOC) of the program, the number of passing JUnit tests, failing JUnit tests, the test execution time and the branch coverage. We rely on Cobertura [1], a tool based on *jcoverage5*, to calculate the branch coverage for each program.

c) Input Domain: We extract the program preconditions and the input domain of each program. The program preconditions are constraints for input domain.

d) Failures: We summarize the failure types when executing test cases for each buggy program of QuixBugs dataset.

e) Unique Characteristics : Comparing with the benchmarks of literatures, we discuss the unique characteristics of QuixBugs dataset.

2) RQ2: Repairability: To apply automatic program repair on QuixBugs, we first 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 [9]. Eventually, we decide to choose 2018 version of Astor [2] with five repair modes [21], Arja [42], Nopol with two repair modes [39], NPEfix [4] and Java implementation of RSRepair [29] by [42] for their open-source and continued maintenance.

The five repair systems target 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, Arja, RSRepair are considered as the representative tools for generate-and-validate approaches and Nopol and NPEfix are considered as representative tools for synthesis-based approaches. All the five repair systems take as input the source code of a buggy program and the corresponding test suite which contains at least one failing test case, and generate one or more patches that make all test cases pass if such patches exist in their search spaces.

We run the five repair systems separately on the whole QuixBugs benchmark. In the experiment, we take at most five patches per bug from per repair system, thus, we do not stop the repair process after finding the first patch, until we obtain five test-suite adequate patches. For the same reason, we run all available repair modes of Astor [21], [22]: JGenProg, JKali, JMutRepair, Cardumen, Tibra and Nopol [39], [6]: SMT and Dynamoth.

3) RQ3: Patch Correctness Assessment.: As shown in previous research [37], [36], assessing correctness of 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

[1]http://cobertura.github.io/cobertura/
[2]https://github.com/SpoonLabs/astor

is that a patch that passes a test suite may still be incorrect, as it may overfit to the provided test cases [31] 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.

In our experiment, we consider three techniques for patch correctness assessment: a) using automatically generated tests by a search-based approach based on a reference version [41]; b) using automatically generated tests by a program specific generator based on a reference version [2]; and c) manual analysis of patch correctness [20].

a) Search-based Test Generation Technique: Using automated test generation is one way for assessing patch correctness [37], [36], [41], [40]. In our study, the search-based test generator technique takes as input a reference version of buggy program. The reference version is used as oracles which means the output of the reference program is the expected output. In this paper, we consider Evosuite [8], the state-of-the-art automated test generation tool, for generating those new correctness assessment tests. We have chosen Evosuite according to the results of Shamshiri et al. [30], which have shown that Evosuite is the most effective tool for this use case.

For each of the 40 buggy programs in the 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 (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) Program specific Test Generation Technique: In order to strengthen our patch correctness assessment, we consider a second random testing approach called InputSampling. InputSampling, as an implementation of random testing [2] for QuixBugs, samples the input space according to a uniform distribution, and that uses the reference version as oracle [23]. 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 of Patch Correctness: The third considered patch assessment technique is manual analysis [29], [20]. 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 authors 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. As 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 without further analysis; 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: *sensitivity, performance, and limitations.*

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. *Performance:* We analyze the time costs of the two considered automated patch correctness assessment techniques. *Limitations:* Eventually, we discuss the advantages and limitations of each technique.

IV. Experimental Results

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

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 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

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 [18]:
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

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 ranging 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 [1].

Comparing the benchmarks of literatures [16], [7], [11], [5], 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 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 experiment, we present the automatic repair results by employing five test-suite based repair systems. In total, we have performed 40 repair attempts with Arja, NPEFix and

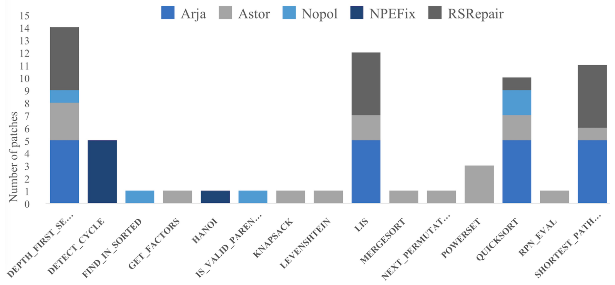

Fig. 1: QuixBugs repair results by 5 repair systems

RSRepair, respectively; 200 repair attempts with Astor (40 programs $\times$ 5 repair modes); 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 $(40+40+40+200+80)\times 5 = 2000$ minutes, i.e., 33 hours.

We present our repair results in Figure 1. In total, we have obtained 64 patches generated by the five considered repair approaches, those patches repair 15 different buggy programs. Arja generates 20 patches for 4 programs; Astor generates 17 patches for 11 programs; Nopol generates 5 patches for 4 programs; NPEFix generates 6 patches for 2 programs; and RSRepair generates 16 patches for 4 repair systems.

There are 4 programs are repaired by more than three repair systems and the remaining 11 programs are repaired by only one repair system. We have more patches than repaired programs because: *1)* some bugs are repaired different repair systems. *2)* some bugs are repaired by different modes of one repair system (e.g. *lis* is repaired in two different modes of Astor), and *3)* for a given repair mode, we configured the tools to output at most five patches from the search space (e.g. five patches for *lis* are generated by Arja).

Listing 1 gives a code snippet of the fifth patch generated by Arja for the buggy program *quicksort*, i.e. the patch Id is *patch_quicksort_Arja_5.diff* The bug type of *quicksort* is C (*Incorrect comparison operator*). The commented code gives the human reference patch, in this case "*else if (x>=pivot)*". Arja generates patch by modifying conditions of *else* block, using "*else*" to replace "*else if (x>pivot)*". In this context, we consider the patch correct since it is semantically the same with the reference version.

```
for (Integer x : arr.subList(1, arr.size())) {
    if (x < pivot) {
        lesser.add(x);
//Reference patch: } else if (x >= pivot) {
- } else if (x > pivot) {
- greater.add(x);
- }
+ } else
+ greater.add(x);
}
```

Listing 1: Quicksort patch generated by patch_quicksort_Arja_5.diff

We discuss the bug type of programs in Section IV-A. From the repair result, there are 8 / 14 types of bugs that are fixed by the considered automatic repair approaches which is encouraging. They are B (*hanoi*, *is_valid_parenthesization* and *powerset*), C (*knapsack*, *mergesort*, *next_permutation* and *quicksort*), D(*detect_cycle*), 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 Arja, Astor and RSRepair has mutation repair strategy of comparison operator.

Then we analyze how the tests impact the considered repair approaches. For the 15 repaired programs, 5 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. There are four programs with no passing tests that are repaired by Astor, Arja and RSRepair and only one is repaired by NPEFix. The reason 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*, 4 of *stack overflow* errors and 1 of *null pointer exceptions*. While this shows that the considered repair tools handle three kinds of failure types, it also shows that *infinite loops*, *concurrent modification exceptions* and *array index errors*, are not well handled by the considered repair systems.

Answer to RQ2: 15 / 40 QuixBugs programs are repaired with test-suite adequate patches synthesized by five repair systems. Those test-suite adequate patches cover 8 bug types and 5 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 64 generated patches, together with the correctness assessment results that 33 out of 64 patches are discarded together by the three considerate assessment techniques. The first column gives the patch ID complying the name convention of *<buggyProgram_repairSystem_patchNo>*. 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, which 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 14 patches in common (in circle), 15 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.

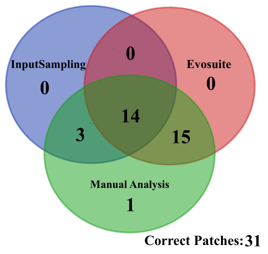

Fig. 2: Patches identified as incorrect by the three considered assessment techniques

The remaining 31 patches are classified as *Correct*. Three patches are identical to the reference version (*knapsack, levenshetein and rpn_eval*), the other 28 patches are semantically equivalent to the reference version. 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.

1) 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 15 repaired programs, they are: *find_in_sorted*, *get_factors*, *knapsack*, *levenshtein*, *next_permutation*, *shortest_path_lengths*. After the preparation work, we ran the 64 patched programs over the corresponding 15 different test suites generated by Evosuite. As a result, 29 patches out of 64 patches are identified as *Incorrect*.

2) 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 64 patched programs was executed against the corresponding 300 tests to be assessed the correctness. As a result, 17 patches out of 64 are discarded.

3) Manual Analysis: As a result, shown in the fourth column of Table II, 33 patches are labeled as *Incorrect*, and the other 31 patches are either identical or semantically correct. Together, Evosuite and InputSampling discard 32 different patches. But manual analysis identifies one additional patch for program *next_permutation* which not discarded by the two considered automated techniques.

Here, Listing 2 to Listing 5 presents the code snippets of four patches that representing difference assessment results given by the three considered assessment techniques.

TABLE II: Patch Correctness Assessment Results

Patch ID	Evosuite analysis	InputSampling analysis	Manual analysis
depth_first_search_Astor_1	Incorrect 20/120	Incorrect 154/300	Incorrect
depth_first_search_Astor_2	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Astor_3	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Arja_1	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Arja_2	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Arja_3	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Arja_4	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Arja_5	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_Nopol_1	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_RSRepair_1	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_RSRepair_2	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_RSRepair_3	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_RSRepair_4	Incorrect 21/120	Unknown 0/300	Incorrect
depth_first_search_RSRepair_5	Incorrect 21/120	Unknown 0/300	Incorrect
detect_cycle_NPEFix_1	Unknown 0/120	Unknown 0/300	Correct
detect_cycle_NPEFix_2	Unknown 0/120	Unknown 0/300	Correct
detect_cycle_NPEFix_3	Unknown 0/120	Unknown 0/300	Correct
detect_cycle_NPEFix_4	Unknown 0/120	Unknown 0/300	Correct
detect_cycle_NPEFix_5	Unknown 0/120	Unknown 0/300	Correct
find_in_sorted_Nopol_1	Incorrect 142/316	Incorrect 168/300	Incorrect
get_factors_Astor_1	Incorrect 27/119	Unknown 0/300	Incorrect
hanoi_NPEFix_1	Incorrect 29/29	Unknown 300/300	Incorrect
is_valid_parenthsization_Nopol_1	Incorrect 32/150	Incorrect 61/300	Incorrect
knapsack_Astor_1	Unknown 0/208	Unknown 0/300	Identical
levenshetein_Astor_1	Unknown 0/120	Unknown 0/300	Identical
lis_Astor_1	Unknown 0/126	Unknown 0/300	Correct
lis_Astor_2	Unknown 0/126	Unknown 0/300	Correct
lis_Arja_1	Unknown 0/126	Unknown 0/300	Correct
lis_Arja_2	Unknown 0/126	Unknown 0/300	Correct
lis_Arja_3	Unknown 0/126	Unknown 0/300	Correct
lis_Arja_4	Unknown 0/126	Unknown 0/300	Correct
lis_Arja_5	Unknown 0/126	Unknown 0/300	Correct
lis_RSRepair_1	Unknown 0/126	Unknown 0/300	Correct
lis_RSRepair_2	Unknown 0/126	Unknown 0/300	Correct
lis_RSRepair_3	Unknown 0/126	Unknown 0/300	Correct
lis_RSRepair_4	Unknown 0/126	Unknown 0/300	Correct
lis_RSRepair_5	Unknown 0/126	Unknown 0/300	Correct
mergesort_Astor_1	Unknown 0/241	Unknown 0/300	Correct
next_permutation_Astor_1	Unknown 0/222	Unknown 0/300	Incorrect
powerset_Astor_1	Unknown 0/87	Incorrect 224/300	Incorrect
powerset_Astor_2	Unknown 0/87	Incorrect 300/300	Incorrect
powerset_Astor_3	Unknown 0/87	Incorrect 224/300	Incorrect
quicksort_Astor_1	Unknown 0/169	Unknown 0/300	Correct
quicksort_Astor_2	Unknown 0/169	Unknown 0/300	Correct
quicksort_Arja_1	Unknown 0/169	Unknown 0/300	Correct
quicksort_Arja_2	Unknown 0/169	Unknown 0/300	Correct
quicksort_Arja_3	Unknown 0/169	Unknown 0/300	Correct
quicksort_Arja_4	Unknown 0/169	Unknown 0/300	Correct
quicksort_Arja_5	Unknown 0/169	Unknown 0/300	Correct
quicksort_Nopol_1	Unknown 0/169	Unknown 0/300	Correct
quicksort_Nopol_2	Unknown 0/169	Unknown 0/300	Correct
quicksort_RSRepair_1	Unknown 0/169	Unknown 0/300	Correct
rpn_eval_Astor_1	Unknown 0/218	Unknown 0/300	Identical
shortest_path_lengths_Astor_1	Incorrect 16/167	Incorrect 297/300	Incorrect
shortest_path_lengths_Arja_1	Incorrect 51/167	Incorrect 281/300	Incorrect
shortest_path_lengths_Arja_2	Incorrect 51/167	Incorrect 281/300	Incorrect
shortest_path_lengths_Arja_3	Incorrect 51/167	Incorrect 281/300	Incorrect
shortest_path_lengths_Arja_4	Incorrect 51/167	Incorrect 281/300	Incorrect
shortest_path_lengths_Arja_5	Incorrect 46/167	Incorrect 281/300	Incorrect
shortest_path_lengths_RSRepair_1	Incorrect 51/167	Incorrect 281/300	Incorrect
shortest_path_lengths_RSRepair_2	Incorrect 46/167	Incorrect 281/300	Incorrect
shortest_path_lengths_RSRepair_3	Incorrect 46/167	Incorrect 281/300	Incorrect
shortest_path_lengths_RSRepair_4	Incorrect 46/167	Incorrect 281/300	Incorrect
shortest_path_lengths_RSRepair_5	Incorrect 51/167	Incorrect 281/300	Incorrect
# discarded patches	29	17	33

```
-  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);
//Reference patch: return binsearch(arr,x,mid+1,end);
```

Listing 2: Patch for find_in_sorted_Nopol_1, discarded by Evosuite and InputSampling

```
    int max = (int) (Math.sqrt(n) + 1.0);
-   for (int i = 2; i < max; i++) {
+   for (int i = 2; (n % n) == 0; i++) {
     ... }
    return new ArrayList<Integer>();
 //Reference patch:
 //return new ArrayList<Integer>(Arrays.asList(n));
```

Listing 3: Patch for get_factors_Astor_1, discarded by Evosuite but not InputSampling

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

Listing 4: Patch of powerset_Astor_1, discarded by InputSampling but not Evosuite

```
  for (int j=perm.size()-1; j!=i; j--) {
-   if ((perm.get(j)) < (perm.get(i))) {
+   if ((perm.get(j)) >= (perm.get(i))) {
    //Reference patch:
    //if (perm.get(j) > perm.get(i)) {
```

Listing 5: Patch for next_permutation_Astor_1, only discarded by manual analysis

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 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 a patch 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, thus the generated tests always pass. On the contrary, InputSampling samples the right inputs to discard it.

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 [1]). 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, 33 out of 64 patches are shown to be incorrect due to overfitting. By using the reference version as oracle, the search-based tool Evosuite discards 29 incorrect patches, the random testing tool InputSampling discards 17 patches, and the manual analysis discards 33 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: We compare the automated assessment results of all 64 patches by invoking Evosuite once, 5 times and 30 times, and the result based on respectively 10 tests, 50 tests and 300 tests generated by InputSampling. The data is available on our online appendix [1].

There are 28 patches are discarded when we invoke Evosuite only once, while 29 patches (one more) are discarded when running it 5 times and 30 times. On the other hand, 16 patches are discarded by the first 10 InputSampling tests and 17 patches (one more) are discarded 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 a small number of automatically generated test cases is enough to identify incorrect patches in scientific studies. On the other hand, the more tests, the more incorrect patches are discarded, and this holds for both techniques. Our experiment shows that both approaches are really complementary, they catch different incorrect patches. To that extent, we recommend that researchers use a budget based approach: generated as many tests within the time period they can afford.

2) Performance: Evosuite takes 985 minutes to generate 30 test suites for all 15 repaired programs, i.e. approx 15 seconds per test per program. InputSampling takes less than 15 minutes to generate $(15 \times 30) = 4500$ test methods for them; which takes 0.2 second to generate a test per program, which is 75X 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.

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-C3c); *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 corner values due to bad chance (see the case study of *get_factors* discussed in Section IV-C3b); *b)* and InputSampling requires manual work to write the program 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 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 [1].

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. Our future studies on QuixBugs will address this threat and identify new patches.

VI. Related Work

Current Datasets of Bugs: The widely used benchmarks in automatic program repair research include Introclass [16], ManyBugs [16], SIR [5], Codeflaws [33], Defects4J [11], and IntroClassJava [7].

IntroClass: Smith et al. [31] evaluate overfitting patches generated by GenProg and TrpAutoRepair on IntroClass. Le et al. [15] systematically characterize the nature of overfitting in semantics-based automatic program repair on the IntroClass and Codeflaws benchmarks. Ke et al. [12] evaluate SearchRepair on IntroClass.

ManyBugs: Qi et al. [29],Mechtaev et al. [24], Long and Rinard [19] evaluate repair approaches on ManyBugs.

SIR: Stratis and Rajan [32] evaluate their approach to improve instruction locality across test case runs on SIR. Nguyen et al. [27] propose SemFix and evaluate it on SIR.

Codeflaws: Papadakis et al. [28] collect and analyze mutant quality indicators based on Codeflaws. Chekam et al.[3] propose a new perspective to tackle the fault revelation mutant selection and evaluate their work on Codeflaws.

Defects4J: Martinez et al. [20] and Yu et al. [41] report their experiments using Defects4J for evaluating the effectiveness of automatic repair techniques. Xiong et al. [38] propose the ACS repair system and evaluate it on four projects of the Defects4J benchmark. Wen et al. [34] propose CapGen, a context-aware patch generation technique and evaluate this technique on the Defects4J. Hua et al. [10] propose and evaluate SketchFix on Defects4J.

IntroClassJava: Wen at al. [34] also evaluate CapGen on IntroClassJava. Le et al. propose and evaluate S3 on IntroClassJava [14], To the best of our knowledge, QuixBugs has never been used in a program repair experiment until our study.

Patch correctness assessment: Synthesizing new inputs for patch correctness assessment has been studied a couple of times. Xin and Risse [36] 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. [40] 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. [37] 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. [41], 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 the QuixBugs benchmark [18]. 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 program 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.

Acknowledgments

We would like to thank James Koppel for his insight and feedback on this paper. This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by Fundao para a Ciłncia e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017.

References

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

[2] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *TSE*, 2012.

[3] T. T. Chekam, M. Papadakis, T. F. D. A. Bissyande, and Y. Le Traon. Reference : Predicting the fault revelation utility of mutants. 2018.

[4] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. Npefix: Automatic runtime repair of null pointer exceptions in java. *CoRR 2015*.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*

[6] T. Durieux and M. Monperrus. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *AST*, 2016.

[7] T. Durieux and M. Monperrus. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Technical report, 2016.

[8] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE '11*, 2011.

[9] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *TSE*, 2017.

[10] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *ICSE*, 2018.

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

[12] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search. In *Proceedings of ASE*, 2015.

[13] R. Lawler. How do you hire great engineers? give them a challenge. 2012.

[14] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *ESEC/FSE*, 2017.

[15] X. B. D. Le, F. Thung, D. Lo, and C. L. Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 2018.

[16] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. 2015.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. In *IEEE Trans. Softw. Eng.*, 2012.

[18] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings of SPLASH.*, 2017.

[19] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of ESEC/FSE*, 2015.

[20] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus. Automatic repair of real bugs: An experience report on the defects4j dataset. 2017.

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

[22] M. Martinez and M. Monperrus. Open-ended exploration of the program repair search space with mined templates: the next 8935 patches for defects4j. *CoRR*, 2017.

[23] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of ICSE*, 2018.

[24] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, 2016.

[25] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of ICSE*, 2014.

[26] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 2018.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *ICSE*, 2013.

[28] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. *http://hdl.handle.net/10993/34352*, 2018.

[29] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of ISSTA*, 2015.

[30] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of ASE*, 2015.

[31] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of ESEC/FSE*, 2015.

[32] P. Stratis and A. Rajan. Test case permutation to improve execution time. In *Proceedings of ASE*, 2016.

[33] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *Proceedings of ICSE-C*, 2017.

[34] M. Wen, J. Chen, rongxin wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *ICSE*, 2018.

[35] Q. Xin. Towards addressing the patch overfitting problem. In *Software Engineering Companion (ICSE)*, 2017.

[36] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, 2017.

[37] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based automatic program repair. In *ICSE*, 2018.

[38] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, 2017.

[39] J. Xuan, M. Martinez, F. Demarco, M. Clment, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE*, 2016.

[40] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *ESEC/FSE17*, 2017.

[41] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the nopol repair system. 2018.

[42] Y. Yuan and W. Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *CoRR*, abs/1712.07804, 2017.