2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST) | 978-1-7281-6836-4/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICST49551.2021.00033

Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods

Amirfarhad Nilizadeh*, Gary T. Leavens*, Xuan-Bach D. Le†, Corina S. Păsăreanu‡, and David R. Cok§
*University of Central Florida, Orlando, Florida, USA Email: af.nilizadeh@knights.ucf.edu, Leavens@ucf.edu
†University of Melbourne, Melbourne, Australia Email: bach.le@unimelb.edu.au
‡Carnegie Mellon University and NASA Ames Research Center, California, USA Email: Corina.S.Pasareanu@nasa.gov
§Safer Software Consulting, LLC, Rochester, New York, USA Email: david.r.cok@gmail.com

***Abstract*—Automated program repair (APR) techniques have shown a promising ability to generate patches that fix program bugs automatically. Typically such APR tools are *dynamic* in the sense that they find bugs by testing and they validate patches by running a program's test suite. Patches can also be validated manually. However, neither of these methods for validating patches can truly tell whether a patch is correct. Test suites are usually incomplete, and thus APR-generated patches may pass the tests but not be truly correct; in other words, the APR tools may be *overfitting* to the tests. The possibility of test overfitting leads to manual validation, which is costly, potentially biased, and can also be incomplete. Therefore, we must move past these methods to truly assess APR's overfitting problem.**

We aim to evaluate the test overfitting problem in dynamic APR tools using ground truth given by a set of programs equipped with formal behavioral specifications. Using these formal specifications and an automated verification tool, we found that there is definitely overfitting in the generated patches of seven well-studied APR tools, although many (about 59%) of the generated patches were indeed correct. Our study further points out two new problems that can affect APR tools: changes to the complexity of programs and numeric problems. An additional contribution is that we introduce the first publicly available data set of formally specified and verified Java programs, their test suites, and buggy variants, each of which has exactly one bug.

***Index Terms*—Automated Program Repair (APR); test overfitting; formal methods;**

I. Introduction

Bugs and reliability problems exist even in critical software; for example, bugs have been found in protocol and security applications, such as blockchain systems [1] and openssh [2].

The goal of automated program repair (APR) tools is to make debugging automatic. APR tools use a two-step process. The first step is to detect bugs and find their likely cause, which is known as *bug localization* [3]. The second step is to repair the bugs, which is known as *patch generation* [4].

However, current APR tools sometimes generate patches that pass all tests but are incorrect, a problem known as *test overfitting* [5]–[8] (see section III).

While APR tools can generate many patches, assessing the true correctness of patches, and thus the true extent of test overfitting, remains a challenge. Although one could validate the correctness of patched programs using a second independent test suite [5], [7] or by having humans examine the patched programs manually [8], [9], neither method can truly tell if a patch is correct. Test suites are usually incomplete and thus may not be able to identify all incorrect patches. On the other hand, manual patch validation may involve human biases and is also incomplete; for example, Martinez et al. [6] were unable to decide the correctness of about 14.3% of the patches in their study. Yet, without a reliable and unbiased way to assess patch correctness, it is impossible to know the true extent of the overfitting problem.

Our goal is to provide fresh insight into the test overfitting problem in dynamic APR tools by using formal methods (specification and verification) as an independent standard of correctness. The use of formal methods removes most of the incompleteness of testing and all of the potential for bias in human inspections of code. While it is possible that formal verification can be incomplete, we avoid much of this incompleteness in our experiments by creating a set of formally specified Java programs that are verified as correct [10].[1]

Formal methods have been found to be very effective in finding bugs [11], [12] and are the gold standard for proving program correctness.

Our study uses seven dynamic APR tools for Java, namely Cardumen [13], jGenProg, jKali, jMutRepair, which all are provided by the Astor framework [14], and ARJAE and Kali-A from ARJA framework [15] , and Nopol [16]. These 7 tools are representative and well-studied repair tools that target Java programs, and they are often used in recent studies [17].

We use a set of buggy programs equipped with test suites and full formal specifications. We run the dynamic APR tools on these buggy programs with their test suites to generate repairs and use automated formal verification to verify the correctness of the APR-generated patched programs against the corresponding full formal specifications. Our experimental evaluation confirms that overfitting is a problem with the dynamic APR tools that we tested, although all but about 59.17% of the validated repaired programs were indeed correct. Thus the formal methods tools that we used were effective in determining the extent of overfitting, although there were a few false negatives (about 4.15%) that we discuss in subsection V-B.

We created the first verified, publicly available dataset of Java programs with JML specifications for the experimental

[1] We believe that our data set is the first such formally specified and verified Java programs to be made public.

978-1-7281-6836-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ICST49551.2021.00033

evaluation [10].

We used the OpenJML [18] verifier to prove the correctness of each of these programs. We also created a dataset of buggy variants of these programs [19], each of which has exactly one bug (as required by most APR tools). The OpenJML tool is also able to identify the bug in each buggy Java program. Thus this dataset can be used to automatically and objectively evaluate test overfitting.

II. Background

This section introduces terminology related to APR systems. We also introduce the seven dynamic APR tools for Java used in our study. Finally, we give some background on the formal methods used in our experiments: the Java Modeling Language (JML) and the OpenJML tool.

A. APR Background

An APR tool has two inputs: a buggy program and a description of the program's expected behavior. The behavioral description is used for localizing bugs and for validating (or verifying) generated patches. Although some researchers have used formal specifications to describe the program's desired behavior [20]–[22], most APR tools are dynamic in that the behavioral description is a test suite [9], [23], [24]. Almost all open source APR tools [25] use a test suite as a behavioral description.

It is possible that an APR tool may not be able to repair a buggy program. This may happen for two reasons. First, the process of finding a patch may take longer than is expected or longer than the user is willing to wait. Second, an APR tool may exhaust the space of all possible patches it can generate, but none of these may be sufficient for a repair.

APR tools use either a static or dynamic approach. The focus of our experiments is the commonly used *dynamic APR* approach, which runs the buggy program and its test suite to localize bugs and validate patches. In dynamic APR, if a program with a patch passes the program's test suite, then that patch is considered to be a repair. By contrast, the *static APR* approach does not run the buggy program, but instead uses formal methods. For example, the work of Jobstmann [26] uses the static APR approach, with LTL specifications and model checking. Other researchers also have used the static approach [27]–[30].

B. APR Tools

Our experiments use seven well-studied dynamic APR tools that can operate on arbitrary Java programs.[2] Our main goal is not comparing APR tools; instead, our goal is to study the problem of test overfitting and demonstrate that formal methods could truly be used to decide the correctness of patches and evaluate test overfitting.

Although a recent experimental study [31] shows that SimFix [32], ACS [33], AVATAR [34], and TBar [35] are the most promising APR tools, these tools are configured to only work on the Defects4J dataset. Since these tools could not be used for arbitrary Java programs, we were unable to run them on our dataset. In section IV we will explain that why a new dataset is created for this study instead of using Defects4J.

Therefore, we selected the 7 APR tools described below.

1) ARJAE: ARJAE is the improved version of ARJA tool [15] which is based on genetic programming. The ARJAE tool exploits several repair templates in addition to statement-level redundancy.

2) Cardumen: Cardumen [13] uses a spectrum-based fault localization (SBFL) tool for bug localization, GZoltar [36]. SBFL statistically analyzes the buggy program's behavior based on the failing and passing tests in the test suite. It then ranks statements of the program based on their likelihood of being faulty [37], [38]. Cardumen uses fine-grained program elements and automatically mined repair templates to generate patches.

3) jGenProg: GenProg [39] is one of the first and best-known APR algorithms. The Java version of this tool is jGenProg [14]. The jGenProg tool uses SBFL for bug localization. Then, it uses three operations (insert, remove, and replace) for repairing a program. Its genetic programming algorithm uses a fitness measure [14]. The genetic programming algorithm searches in the program's repair space and the best two candidates (those with the highest fitness) are used for a crossover, which generates new candidate patches.

4) jKali and Kali-A: The jKali [14] and Kali-A [15] tools are the Java versions of Kali [40]. They each use SBFL for localizing the faulty statements. Then the tool deletes faulty statements from the original program to create candidate programs.

5) jMutRepair: The jMutRepair tool [14] generates patches by mutating conditions in if statements. It uses three different kinds of mutations, but makes at most one change in each if-condition. The first kind is relational, which can change relational operators (such as changing < to <= or vice versa). The second kind is logical that can change "AND" to "OR" or vice versa. The third kind is unary that can apply arithmetic negation to expressions.

6) Nopol: Nopol [16] can fix bugs in if-conditions and synthesize code to prevent errors in such if-conditions. Nopol uses SBFL for fault localization, dynamically gathering information about if-conditions. Then, it transforms this information into an SMT problem. The solution to this SMT problem is then translated into a corrected if-condition.

C. Formal Behavioral Specification

A formal behavioral specification is a mathematical description of a system at an abstract level. It clarifies the assumptions about expected initial and final states of the system, but does not control the implementation's details. Design by Contract (DbC) [41], [42] is a form of executable specification based on Hoare's technique for program verification [43] that uses preconditions, postconditions, and invariants. JML [44]–[46] is a Hoare-style formal specification language with many features adapted from DbC and tailored to Java. JML is the best known formal behavior specification language for Java and has several

[2] We limit our study to Java due to our choice of JML as a formal method.

tools; in our experiments, we use the automated static checker found in the OpenJML tool [18].

In JML, preconditions for a method are defined by "requires" clauses, and postconditions are defined by "ensures" clauses. Also, JML specifications can specify behavior at the class level by using "invariant" clauses. Furthermore, the statement level uses "assert", "assume" and "maintaining" clauses (the latter specifying loop invariants).

JML specifications are written into Java programs as special annotation comments, either on lines following "//@" or between "/*@" and "@*/". The Java compiler considers these annotations to be comments.

For verifying that specifications are correctly implemented, our experiments used the static verifier of OpenJML [18]. In this static verifier, Java code and JML specifications are translated into first-order logic verification conditions. Then the tool calls an SMT solver to prove these verification conditions [47].

As a JML example, Fig. 1 shows a method `absolute` that returns the absolute value of its argument. Its JML specification has two specification cases, separated by **also**, which specify the method's behavior when the argument (`num`) is non-negative and when it is negative. The second case's precondition disallows the argument to be the largest negative integer since, in Java arithmetic that number has itself as its own negation. The return value (written **\result**) is constrained by the postcondition in each case.

```
/*@   requires 0 <= num;
  @   ensures \result == num;
  @ also
  @   requires Integer.MIN_VALUE < num
  @          && num < 0;
  @   ensures \result == -num; @*/
public /*@ pure @*/int absolute(int num) {
    if (0 <= num)
        return num;
    else
        return -num;
}
```

Figure 1. JML specification of the "absolute" method.

III. The Test Overfitting Problem in Dynamic APR

A test suite is usually not a complete specification of a program's behavior because most programs have an infinite or broad domain. Therefore, dynamic APR tools (which by definition use a test suite) may not always identify cases in which a patched program fails to exhibit the expected behavior (on the program's entire domain). Thus, the generated patches may not be reliable even if they pass the test suite. Using these patches automatically, without evaluating their correctness, is unwise, especially in safety-critical systems.

Test Overfitting means that the patch that an APR tool has generated for a program passes the program's test suite but is not a correct patch based on the program's specification. This specification may be implicit in a test suite or in the head of the person evaluating patches manually. However, in our study, we used an explicit, formal specification.

Evidence for the problem of test overfitting comes from recent studies [5]–[8]. For example, Martinez et al. [6] used a subset of Defects4J [48] with 224 real bugs to evaluate three APR tools in Java: jGenProg, jKali, and Nopol. Their results showed that these tools generated 84 patches for 47 individual bugs. They found overfitting in 61 patches (about 72.6%) and determined that 11 patches were correct (13.1%). However, Martinez et al. were unsure about the correctness of another 12 patches (about 14.3%), a problem we avoid by writing formal specifications for programs in our experiments.

Although a better test suite can describe a system's behavior more comprehensively, using more complete test suites can only partially help solve the test overfitting problem [49]. For example, the symbolic execution tool KLEE [50] is used by Smith et al. [5] to generate a white-box test suite. Smith et al. then compared GenProg [39] and RSRepair [51] by using both black-box and white-box test suites. Their work demonstrated that the tests' quality directly correlates with generating correct patches because some patches generated with the black-box tests did not pass the white-box tests. However, there is no guarantee that white-box tests will identify all instances of test overfitting due to fundamental limitations of testing (as programs have infinite domains).

Another approach avoiding overfitting is found in the Overfitted PAtch Detection (Opad) tool of Yang et al. [52]. Opad increases the number of tests in a test suite by generating new test cases, using American Fuzzy Lop (AFL) [53]. Their results show that the test suite produced by Opad improved the detection of overfitting by 75%. Their result also shows that AFL can generate good tests. Thus, we used a fuzzer, also based on AFL, in our study.

While helpful, these results show that increasing the number of tests does not completely solve the test overfitting problem of dynamic APR. Thus, generated patches may not be reliable, and a developer has to evaluate each generated patch's correctness manually. However, evaluating a patch's correctness manually is potentially biased, difficult, time-consuming, and incomplete. Another problem with increasing the number of test cases in a program's test suite is that APR will take more time to execute. Running tests is the most expensive operation in dynamic APR tools [4], [51], [54]. Thus, there is a trade-off between the number of tests in a test suite and the run time of dynamic APR tools, and the generated patches' reliability.

IV. Creating the Datasets

Several well-known datasets of real-world buggy programs have been used for Java APR studies, the most well-known being Defects4J [48]. However, we were not able to use these datasets for our study. For many programs in Defects4J, it is not clear what their correct behavior should be. Furthermore, writing full behavioral specifications[3] for these datasets is not possible for three reasons. First, it is necessary to have a full specification of all used methods, libraries, and classes for verifying a program. However, such full specifications do not yet exist for these datasets. It is possible to infer some assertions and invariants by using a tool, like Daikon [55], which has been used in Yang and Yang's work [56] for adding specification into some programs of Defects4J dataset. However, these inferred specifications are not full as they cannot verify a program's correctness; instead, such (partial) specifications are suited for run time assertion checking (as in Yang's work). Second, for writing the specification, it is necessary to know each program's correct behavior, but that is not available for many of these programs. Finally, many of these datasets make extensive use of I/O. However, such programs are not currently specifiable in JML since they would require more detailed specifications of Java I/O than are currently available.

We tried to add specifications to datasets with smaller programs, like IntroClassJava [57] and the Java version of QuixBugs [58]. However, all the programs in IntroClassJava are using standard input and output. For QuixBugs, we verified two programs, which are included in our dataset. However, the other programs were using I/O or other libraries, which are not yet specified in JML.

To carry out our study, we created two datasets. The first is a set of verified Java programs with full JML specifications. The second is a buggy Java program dataset, which is derived from the first. These are discussed in the subsections below. Also, to avoid bias in the evaluation and study results, the process of generating each test suite and injecting bugs was automated by using a fuzzing tool and a mutation tool, respectively. Our open-source datasets are freely available [10], [19].

A. Creating a Java+JML Dataset

We curated or created 30 subject programs that implement various well-known algorithms and data structures, such as binary search, bubble sort, linear search, factorial, stack, queue, and bank account. These programs are small but well-known, and they are equipped with full formal specifications that describe their expected behavior, allowing OpenJML to automatically verify their correctness.

We were unable to use larger Java programs that have formal specifications written in JML, such as the KOA voting system [59] and some security protocols of Amazon Web Services (AWS) [60]. Like other publicly available Java datasets with JML, KOA was developed with partial specifications for run time assertion checking. Unfortunately, the static checker of OpenJML cannot verify it. The only practical systems annotated with full JML specifications are some security protocols in AWS. However, these are not publicly available.

The Java+JML dataset [10] is the only public dataset of Java programs with full formal behavioral specifications that are automatically verified. It could also be used for future research in other domains of formal methods.

B. Creating Test Suites

Dynamic APR tools require a test suite as an input. Since, as discussed above, the quality of a test suite is important for avoiding test overfitting, we used a fuzzing tool based on AFL to create test suites (following the work of Yang et al. [52]). Using a fuzzer to generate tests also helps avoid bias in the study, as we could automate the process of creating the test suite. For this purpose, we used the Kelinci fuzzing tool [61], [62], which is based on AFL. Kelinci automatically generates sequences of bytes in an attempt to cover all the branches of the program being tested. To create a test suite, we manually wrote drivers for each program that converted these sequences of bytes, generated by Kelinci, into the program's arguments. The drivers were written by considering each program's JML precondition (drawn from the dataset of Java+JML verified programs). To create our test suites, we ran Kelinci five times for each program in the verified programs dataset. We formed test suites from these tests by removing duplicates. This process resulted in one test suite for each verified program.

Kelinci attempts to cover distinct branches in the program; each run is based on a sequence of bytes that Kelinci discovered as covering some path in the program. Therefore, the test suites created using Kelinci have some of the power of white-box tests.

C. Creating a Buggy Program Dataset

Recall that adding full behavioral specifications to well-known datasets, such as Defects4J, is not possible yet. Thus, in this study, we created a new buggy dataset based on the Java+JML dataset. To avoid bias in the study, the process of creating the buggy program dataset was also automated. Since most of the dynamic APR tools can only fix a buggy program with one bug, we created a buggy program dataset in which each program has exactly one bug. PITest [63], a well-known mutation tool for Java programs, was used to create buggy variants of the verified Java programs by injecting single bugs into each. This process generated buggy variants from each of the Java+JML verified programs, which were added to our dataset as buggy Java programs. PITest generates bugs by changing control conditions, changing assignment expressions, removing a method call, and changing return values. Although bugs in this dataset are technically artificial, most of these changes are common real bug patterns. Based on the study of Pan at el. [64] change of if-conditions and change of assignment expressions are two out of the top three common bug patterns in Java. Many bugs in our dataset follow these two common patterns. Also, two classic APR tools are designed to fix bugs in if-conditions: jMutRepair [14] and Nopol [16]. Furthermore, PraPR [65] is an APR tool that uses PITest as a core and gives promising results for fixing real-world buggy programs. Finally, we used two real buggy programs from the QuixBug dataset. PITest generated the same bug after mutating

[3] By *full behavioral specification*, we mean specifications that are sufficient to verify the program's correctness statically.

the correct version of one of them; this shows that PITest does produce bugs that mimic real bugs.

In addition to this real bug (from the corrected QuixBug program), in total PITest created 596 unique buggy variants of our Java+JML dataset of verified programs; each of these 1+596 buggy programs has exactly one bug [19]. At first, PITest, created more mutants, but some were either semantically equivalent to other generated buggy programs or the corresponding verified program; these equivalent programs were discarded.

We also discarded buggy programs if they could not be identified as buggy by OpenJML's static checker or by the test suite derived from the corresponding correct program using Kelinci. For OpenJML, we investigated that OpenJML's static checker could find at least one warning for each of these 597 buggy programs (based on the full behavioral specification of the correct version of each program) in the buggy JAVA+JML dataset. There is an assumption in dynamic APR that at least one of the tests in the test suite must trigger the bug; we discarded 40 buggy programs out of 597 programs that were not triggered by the test suite for the corresponding correct program. Furthermore, the dynamic APR tools that we used do not generate patches when the buggy program has an infinite loop during testing (bug localization); we discarded 10 other such buggy programs. One of these 10 buggy programs was one of the real bugs.

In sum, 547 buggy programs are available in the buggy program dataset. Each buggy program has exactly one bug; furthermore, each bug can be detected with the supplied test suite, and the program does not go to an infinite loop during such tests. The bug in each buggy program can be identified by both OpenJML and by the corresponding test suite. Table I shows, for each verified program, the verified program's number of lines of code (LOC), that program's number of lines of specification (LOS), the number of tests generated (by the fuzzer) for that program's test suite, the number of buggy program variants of that program and branch coverage which is evaluated by JaCoCo [66]. In most programs, generated tests by Kelinci could cover all branches. However, the generated tests did not cover branches that are terminated by throwing an exception. Kelinci's drivers are written based on JML preconditions; preconditions are written based on their normal behavior. Thus, we omitted tests for the exceptional branches.

V. Study Results

This section discusses the results of our study evaluation of true test overfitting in the seven well-studied dynamic APR tools for Java (described in section II-B). We used OpenJML[4] to determine if the patched programs generated by these APR tools are truly correct or overfitted. We also discuss the false negatives that may result from the use of OpenJML to verify repaired programs.

Table I
Dataset statistics.

Name	LOC	LOS	No. Test	No. Bugs	Coverage
Absolute	36	36	18	12	100%
AddLoop	19	6	15	11	100%
Alphabet	126	106	59	28	100%
BankAccount	206	246	141	67	100%
BinarySearch	23	7	11	12	100%
BubbleSort	22	18	5	12	100%
Calculator	26	32	30	7	100%
CombinaPermu	32	42	19	9	100%
CopyArray	9	7	5	4	100%
Factorial	15	25	6	7	100%
Fibonacci	27	26	6	8	67%
FindFirstInSorted	17	12	17	9	100%
FindFirstZero	17	6	11	10	100%
FindInArray	54	29	14	16	100%
FindInSorted	18	13	10	15	100%
GCD	25	43	25	18	100%
Inverse	14	5	15	10	100%
LCM	26	24	29	14	100%
LeapYear	20	15	20	7	100%
LinearSearch	16	7	10	5	100%
OddEven	8	6	10	4	100%
Perimeter	60	45	30	17	100%
PrimeCheck	13	11	12	8	100%
PrimeNumbers	30	27	5	14	100%
Smallest	14	6	6	7	100%
StackQueue	324	386	155	100	85%
StrPalindrome	10	12	10	5	100%
StudentEnroll	148	184	33	46	67%
Time	163	125	74	58	86%
TransposeMatrix	15	16	10	7	100%

A. Test Overfitting in Dynamic APR Tools

Recall that seven open-source, well-studied, and dynamic APR tools for Java were used in this experiment: ARJAE, Cardumen, jGenProg, jKali, jMutRepair,[5] Kali-A, and Nopol. We applied each of these tools with their default setting to each of the 547 buggy programs in the buggy Java+JML dataset; Table II shows the results. Generated patches in this step are validated only by the corresponding test suites. Although ARJAE generates many possible candidate patches for each buggy program, we only evaluated the first generated patch.

Table II
Validation Results for APR Tools

APR Tools	Not Repaired	Validated Repairs
ARJAE	198	349
Cardumen	477	70
jGenProg	496	51
jKali	539	8
jMutRepair	505	42
Kali-A	457	90
Nopol	530	17

[4] The OpenJML version was 8.46.

[5] jMutRepair in the Astor framework was updated after our study.

In total, these tools generated 627 patches. In some cases, several APR tools generated a patch for one buggy program; for example, all seven tools generated a patch for program “BinarySearch bug13,” although their generated patches were different from each other.[6]

To truly detect test overfitting, we attempted to verify the validated repaired programs using OpenJML and the JML specifications (created for our Java+JML verified program dataset, see section IV-A). Table III shows the results using OpenJML’s static analyzer for each of the dynamic APR tools.

Table III
VERIFICATION OF VALIDATED PATCHES USING OPENJML

APR Tools	Validated	Verified	Not Verified
ARJAE	349	192	157
Cardumen	70	56	14
jGenProg	51	41	10
jKali	8	6	2
jMutRepair	42	40	2
Kali-A	90	21	69
Nopol	17	15	2

Based on the results in Table III, 371 validated repaired programs were truly correct and verified with respect to their full formal behavioral specification, which is about 59.17%; thus, 256 repaired programs, about 40.83%, were not verified and presumed to be overfitted.[7] These results show that test overfitting is still happening with these seven well-studied dynamic APR tools even when programs are small and tests are covering almost all branches.

B. Evaluating OpenJML’s Classifications

It is essential to evaluate the validity of OpenJML’s classification of patches as “Verified” and “Not Verified.” OpenJML’s verifier is sound, so it will not verify a program that is not correct. Thus there are no false positives; we manually checked the “Verified” repaired programs mentioned in Table III and found that all of them were, in fact, truly correct.

However, OpenJML’s verifier is not complete, so evaluating the “Not Verified” repaired programs is more complex and shows some interesting results. Table IV shows the result of manually evaluating the “Not Verified” repaired programs from Table III.

In Table IV, 256 repaired programs are in the “Not Verified” class. Of these repaired programs, 230 of them, about 36.68% of total generated patches, are absolutely overfitted: they do not work correctly for all input domain values based on their behavioral specification from the Java+JML dataset. However, the other 26 patches (about 4.15%) are correct for all inputs allowed by the program’s input domain and are thus false negatives. We classified these 26 patches into three main groups described below: structural, modularity, and overflow.

Table IV
RESULTS FOR OPENJML’S VERIFICATION ATTEMPTS

APR Tools	Not Verified	Overfitting (True Negatives)	Not Overfitting (False Negatives)
ARJAE	157	138	19
Cardumen	14	11	3
jGenProg	10	6	4
jKali	2	2	0
jMutRepair	2	2	0
Kali-A	69	69	0
Nopol	2	2	0

1) Structural: In this class, APR tools repair a bug by changing a Java method’s structure, like adding a new loop, which OpenJML needs a new loop invariant. 14 out of 26 programs in this group were generated by ARJAE, Cardumen, and jGenprog (these generated 10, 3, and 1 programs, respectively). For example, Fig. 2 is the buggy program “Factorial bug2.” Modifying the condition of the “for loop” from “<” to “<=” is the simplest patch. However, the jGenProg tool repaired this buggy program by adding a new “for loop” as a patch, as shown in Fig. 3. The repaired program looks strange[8], but it is correct for its entire domain: the integers between 0 and 20 (to avoid overflowing).

```
public long factorial(int n) {
   int i;
   long fact = 1;
   if ( n == 0) {
       return fact;
   }
   for (i = 1; i < n; i++) {
         fact = fact * i;
   }
   return fact;
}
```

Figure 2. The “factorial” method of the “Factorial bug2” program.

Sometimes the time complexity of the repaired program in this group increased, and in some cases, the new structure changed the program’s order of time complexity. Changing the order of time complexity is a new problem in dynamic APR (see section VI-A).

2) Modularity: JML verification is modular because each method must satisfy its own specification, regardless of how it is used. By contrast, black-box testing evaluates an entire program. Thus programs can be correct in a whole-program sense (as in testing) but not modularly correct. Our study found 6 cases repaired with ARJAE where OpenJML could not verify a program because it was not modularly correct, but the program was correct in a whole program sense.

[6] In total, these seven APR tools together generated a validated patch for 373 individual buggy programs in our dataset, which is about 68.19%.

[7] Also, 235 individual repaired programs have at least one correct patch verified, which is about 42.96% in a total of 547 buggy programs.

[8] Previously, Wang et al. [67] noted that the patches generated by APR tools sometimes are syntactically different from developer-provided patches.

```
public long factorial(int n) {
    int i;
    long fact = 1;
    if ( n == 0) {
        return fact;
    }
    for (i = 1; i < n; i++) {
        for (i = 1; i < n; i++) {
            fact = fact * i;
        }
        fact = fact * i;
    }
    return fact;
}
```

Figure 3. The “Factorial bug2” repaired by jGenProg.

```
//@ requires 0 <= index && index < fib.length;
//@ ensures \result == fib[index];
public /*@ pure @*/ long getFib(int index) {
    fib[1] = 1;
    return fib[index];
}
```

Figure 4. The “getFib” method of the “Fibonacci bug8” modified by ARJAE.

For example, in “PrimeNumbers bug9” there is a buggy public method, but ARJAE instead modified a private method called by the buggy method so that both methods work correctly together. However, the patched program was not modularly correct; that is, the private method in the patched program does not satisfy its JML specification. The public method has correct behavior using the patched private method that it calls, although OpenJML will not verify it. On the other hand, for testing, in a whole program sense, the combination of these two methods passes all the test cases, as the public method with the incorrect private method has the specified behavior. This shows that a method’s visibility is an important issue for correctness, as testing does not directly call private methods.

Also, in our study, sometimes the generated patch with ARJAE could repair the bug, but also ARJAE made unnecessary changes in the buggy method or other methods. In most cases, these changes make a new bug in the program (overfitting) or do not change the program’s behavior (validate and verified). However, sometimes OpenJML did not verify the program because the insertion caused a side effect (an unnecessary assignment statement) in a method specified to be pure by JML.[9] In these cases, from the point of view of modular verification, the program is incorrect, but it has the specified behavior. For example, “Fibonacci bug8” was fixed correctly with ARJAE, but also it inserted “`fib[1]=1;`” in a pure method, as shown in Fig. 4. However, this change does not affect the value of the `fib[1]` field that has already value one, and so the program’s behavior is not changed, although OpenJML considers the program to be incorrect.

3) Overflow: An Integer overflow can also happen in 6 of the repaired programs (3 by ARJAE and 3 with jGenProg), which prevents OpenJML from verifying these programs. We discuss this problem further in subsection VI-B.

In summary, using OpenJML to check for test overfitting is sound but not complete. There are no false positives, but some false negatives that result from using a verifier to determine the truth about patch correctness. For the reasons mentioned earlier, it is difficult to eliminate these false negatives. [10]

VI. New Problems for Dynamic APR

We are not aware of any research on dynamic APR about the possible negative effects resulting from an APR tool changing repaired program’s complexity or introducing potential numerical problems. Some of these issues occurred in our experiments, but they were not detected by the programs’ test suites or by JML’s verifier. We discuss them in detail in the following subsections.

A. Changing a Program’s Complexity

A program’s complexity is typically abstracted away (or ignored) by both testing and formal specification and verification, which tend to focus only on a program’s functional behavior. However, in many practical situations, a program’s time and space complexity (in the worst case or the average case) may matter a great deal, especially in critical systems.

1) Time Complexity: Our results show that the order of time complexity can change and increase dramatically by using current dynamic APR tools (at least by using ARJAE and Cardumen). Increasing the time complexity is not a test overfitting problem in theory because the expected behavior is still correct. However, it could be a significant issue in practice.

For example, the order of time complexity changed from $O(log(n))$ to $O(n)$ in seven generated repaired programs by using ARJAE and Cardumen on binary search programs (such as “BinarySearch bug9”). These seven repaired programs pass all of the tests, and they are correct for all possible inputs. However, in reality, changing the order of time complexity from $O(log(n))$ to $O(n)$ changes the character of the program; in this case, the character of binary search is changed to linear search. For example, Fig. 5 shows the “Binary” method of the “BinarySearch bug9” program. Here the expected patch is “`mid = low + (high - low) / 2;`” that would replace the buggy “`mid = low + (high - low) * 2;`”. However, the repair generated by Cardumen is “`mid = low + (low - low) * 2`”, as shown in

[9] A pure method is a method that must terminate and does not have side effects, such as changing class field values using the assignment.

[10] In theory, since Hoare logic is inherently incomplete [68], [69], it will be impossible to eliminate all potential false negatives.

```
public static int Binary(int[] arr, int key){
    if (arr.length == 0) {
        return -1;
    } else {
        int low = 0;
        int high = arr.length;
        int mid =  high / 2;
        while(low < high && arr[mid] != key){
            if (arr[mid] < key) {
                low = mid + 1;
            } else {
                high = mid;
            }
            mid = low + (high - low) * 2;
        }
        if (low >= high) {
            return -1;
        }
        return mid;
    }
}
```

Figure 5. The “Binary” method of the “BinarySearch bug9” program.

```
public static int Binary(int[] arr, int key){
    if (arr.length == 0) {
        return -1;
    } else {
        int low = 0;
        int high = arr.length;
        int mid =  high / 2;
        while(low < high && arr[mid] != key){
            if (arr[mid] < key) {
                low = mid + 1;
            } else {
                high = mid;
            }
            mid = low + (low - low) * 2;
        }
        if (low >= high) {
            return -1;
        }
        return mid;
    }
}
```

Figure 6. The “BinarySearch bug9” program repaired by Cardumen.

the “Binary” method of Fig. 6; its time complexity is $O(n)$.[11] The static analyzer of OpenJML does not prove repaired program’s correctness, because the change in the loop invalidates the program’s loop invariant. This shows that time complexity is sometimes tied to the specification’s structure.

Also, in some cases, time complexity did not change dramatically, but the generated patches are not efficient because they do not respect the program’s intended cache behavior; thus, the computation time was increased.

2) Space Complexity: A similar issue involving the space complexity of programs could also occur in APR repaired programs. However, we did not notice any changes in space complexity in our study.

The false negatives that result in changing the program’s complexity could conceivably be addressed by writing specifications that specify the program’s time and/or space complexity. However, such a specification is not easy to write in JML or similar specification languages, and is likely to remain a source of incompleteness.

B. Numeric Problems

There are several potential numeric problems with programs that are hard to test because they may only occur in a minimal number of cases in huge input space. These include integer wrap-around and floating-point underflow, overflow, and loss of precision.

1) Integer Overflow: OpenJML considers an integer overflow as a potential bug, even if it is actually harmless. Thus if APR tools generate repaired programs with integer overflows, then these repairs cannot be verified as correct by OpenJML unless the code is specified to allow for an intended overflow.

Therefore, if a programming language (unlike Java) considered such overflows to be errors, then patches that generate such overflows that the test suite did not identify might be mistakenly validated (but would be overfitted).

In six programs repaired by ARJAE and jGenProg, an integer overflow can happen for some inputs. However, in Java, these integer overflow problems are not errors and do not affect the final result, so the patches are indeed correct. As an example, Fig. 7 and Fig. 8 show the “AddLoop” method of “AddLoop bug2” and its repaired version with jGenProg, respectively. The simplest patch for Fig. 7 is using “**while** (n > 0)” instead of “**while** (n >= 0)” for the first loop. However, it inserts “sum = sum - 1;” at the end of the loop. Integer overflow in Fig. 8 happens for “sum”, when the sum of “x” and “y” is equal to the maximum **int**.

The false negatives that result from integer overflows are due to the verification logic that JML uses, which considers such overflows to be bugs. However, in these six programs, the code was actually correct. This incompleteness in JML is likely to remain because JML designers believe that overflow is more likely to be a sign of a bug than to be used correctly.[12]

2) Floating Point Problems: Floating-point arithmetic could cause similar problems if the verification logic is conservative. There may be some programs in which there is the possibility of floating-point problems (such as underflow or overflow), but those problems do not actually occur. Thus the verification logic would generate a false negative.

[11] Also, the patch generated by the ARJAE tool was similar: “mid = low + (high - high) * 2;”.

[12] JML specifications can indicate that integer overflow is permitted and can reason correctly about such program logic, but this does require explicit specification by the programmer that such behavior is intended.

VII. Related work

We know of no other works that show the definite existence of test overfitting in dynamic APR because no other study uses formal verification.

The work of Le et al. [8] examines other ways to measure the extent of test overfitting, using independent tests and human judgment. It concludes that neither human judgment nor independent testing can truly determine overfitting.

Various other works have evaluated the extent of test overfitting in APR-generated patches. Qi et al. [40] evaluated four APR tools for C on the GenProg benchmark [70]: GenProg [39], RSRepair [51], AE [71] and Kali. They found a significant amount of overfitting: 16 out of 18 patches (about 89%) for GenProg, 8 out of 10 patches (80%) for RSRepair, 24 out of 27 (about 89%) for AE, and 25 out of 27 (about 93%) for Kali.

The other related work is Martinez et al. [6]. They used the Defects4J dataset [48] and evaluated three Java APR tools: jGenProg, jKali, and Nopol. Their results showed that about 72.6% of the generated patches were overfitted patches. Martinez et al. were unsure about the correctness of another 12 patches (about 14.3%), a problem we avoid by writing formal specifications for programs in our experiments, which eliminated the category of unsure results.

Empirical evaluations by Le et al. [7], [72], [73] showed that even APR tools that use the program's behavior (when running the test suite) for generating the patches suffer from the overfitting problem. Le et al. used two C program datasets: IntroClass [74] and Codeflaws [75]. The first dataset has two kinds of test suites; one of them is created by a human, and a second is generated by KLEE [50]. In their work they evaluated four APR tools for C programs: Angelix [24], CVC4 [76], Enumerative [77], and SemFix [78]. Their results for the IntroClass dataset showed that 75%, 80%, 81% and 90% of generated patches for Angelix, CVC4, Enumerative, and SemFix were overfitted, respectively. Also, in the Codeflaws dataset, 54%, 83.5%, 87%, and 68% of generated patches for Angelix, CVC4, Enumerative, and SemFix were overfitted, respectively.

```
/*@ requires Integer.MIN_VALUE <= x + y &&
    x + y <= Integer.MAX_VALUE &&
    y != Integer.MIN_VALUE; @*/
//@ ensures \result == x + y;
public static int AddLoop(int x, int y) {
    int sum = x;
    if (y > 0) {
        int n = y;
        //@ decreases n;
        /*@ maintaining sum == x + y - n &&
            0 <= n; @*/
        while (n >= 0) {
            sum = sum + 1;
            n = n - 1;
        }
    } else {
        int n = -y;
        /*@ maintaining sum == x + y + n &&
            0 <= n; @*/
        //@ decreases n;
        while (n > 0) {
            sum = sum - 1;
            n = n - 1;
        }
    }
    return sum;
}
```

Figure 7. AddLoop method of the "AddLoop bug2" program.

```
/*@ requires Integer.MIN_VALUE <= x + y &&
    x + y <= Integer.MAX_VALUE &&
    y != Integer.MIN_VALUE; @*/
//@ ensures \result == x + y;
public static int AddLoop(int x, int y) {
    int sum = x;
    if (y > 0) {
        int n = y;
        //@ decreases n;
        /*@ maintaining sum == x + y - n &&
            0 <= n; @*/
        while (n >= 0) {
            sum = sum + 1;
            n = n - 1;
        }
        sum = sum - 1;
    } else {
        int n = -y;
        /*@ maintaining sum == x + y + n &&
            0 <= n; @*/
        //@ decreases n;
        while (n > 0) {
            sum = sum - 1;
            n = n - 1;
        }
    }
    return sum;
}
```

Figure 8. The "AddLoop bug2" program repaired by jGenProg.

Ye et al. [79] evaluated five repair systems based on QuixBugs benchmark [58] consisting of 40 small-sized Java buggy programs. Their results show that 64 patches were generated for 15 individual programs. Then, they evaluated the correctness of patches by generating more tests using EvoSuite [80] and by manual analysis. Their results based on manual analysis show that 33 out of 64 generated patches were overfitting, while the test suite found 29 overfitting patches.

Durieux et al. [17] evaluated 11 APR tools for Java programs on 5 benchmarks with 2141 bugs. However, Durieux et al. did not analyze patch overfitting. On the other hand, they introduced the problem of tools overfitting to a benchmark. That is, they showed that these Java APR tools could generate a higher percentage of patches for the Defects4J benchmark (46.84%) than for other benchmarks.

Although somewhat less related to our study evaluation, several related works try to solve the test overfitting problem by creating a better test suite. Smith et al. [5] were among the first to study the test overfitting problem on the GenProg and RSRepair tools. They used a C dataset with 998 bugs. Then they used two test suites: 1) black-box tests created by a developer 2) white-box tests generated by the symbolic execution tool KLEE [50]. Also, by sampling, they selected 25%, 50%, 75%, and 100% of the black-box tests to evaluate different code coverage levels. They showed that black-box tests with lower code coverage generated lower quality patches. Also, all of the generated patches with black-box tests could not pass the white-box tests.

Yang et al. [52] propose to increase the number of tests in a test suite by generating more tests using the AFL fuzzer. In their work, they could filter 321/427 (or about 75%) of the overfitted patches that were generated by GenProg/AE [71], Kali [40], and SPR [81]. Yu et al. [82] describe related techniques targeting Java programs, using the Defects4J dataset; they show that generating new tests can lead to fewer overfitted repairs. Our experiments also used a test suite generated with a Java version of AFL (Kelinci) and still found test overfitting.

Xin and Reiss [83] presented a patch testing technique (DiffTGen) for generating new tests by comparing buggy and repair programs. They used 89 patches generated by jGenProg, jKali, Nopol, and HDRepair [84] and found that 79 of the patches were overfitting. Their technique could detect 39 of these as overfitting.

Xiong et al. [49] propose a technique to increase the size of the test suite without having a test oracle. Their approach was applied to a dataset with 139 patches that were generated with jGenProg, Nopol, jKali, ACS, and HDRepair. The larger test suite leads to 56.3% fewer overfitted repaired programs, but their approach still has some overfitting.

In sum, previous studies of the test overfitting problem only used incomplete test suites or manual analysis to assess patch correctness. However, neither of these patch assessment methods can tell if a patch is truly correct. Our work is the first to study the true extent to which several dynamic APR tools overfit. We do this by employing automated verification on programs equipped with full behavioral specifications. By using OpenJML, we can assess the extent of overfitting in an objective and unbiased way.

VIII. Threats to Validity

In our work, we first tried to add full behavioral specifications to well-known datasets with real bugs, such as Defects4J. However, adding full specifications to these datasets was not possible. We were only able to add full specification into two programs from QuixBugs [58] that are in our dataset. The three main problems for adding full behavioral specification are discussed in section IV. Note that inference tools do not currently provide a full behavioral specification that can automatically verify the correctness of a system. For example, Daikon [55] only provides partial specifications suitable for run time assertion checking. Thus, we manually created the first publicly available Java dataset that can be automatically verified with OpenJML. To avoid bias in the buggy versions of these verified programs, we used PITest and Kelinci to generate a dataset of buggy variants of these verified programs, ensuring that each buggy variant has exactly one bug that is detected by the corresponding test suite and OpenJML. Although the bugs injected by PITest are artificial, they are drawn from common classes of bugs and sometimes match actual bugs (as was the case for one of the buggy programs in QuixBugs). See section IV-C for more discussion about this.

Another threat to our experiment's validity is that we used seven well-studied dynamic APR tools that work on Java programs. A recent experimental study [31] shows that SimFix, ACS, AVATAR, and TBar are the most promising APR tools. However, they are configured to only work on the Defects4J dataset. Also, this paper's main idea is not comparing APR tools; instead, our main goal is to explore true test overfitting by using formal methods. Thus, we selected some tools which are well-known and well-studied. See section II-B for more discussion about this.

IX. Conclusion

Our results truly prove the existence of test overfitting. Instead of showing test overfitting by evaluating candidate patches manually or adding more tests, we used formal methods to decide patch correctness truly. We created the first publicly available Java dataset with JML annotations (30 programs) that OpenJML can verify. A test suite was then created for each correct program, and 547 buggy program versions were created. Next, we used seven well-studied APR tools for generating patches.

Our results confirm the existence of test overfitting on at least the studied APR tools, even when programs are not large, and the tests cover almost all branches. Since OpenJML is sound, using OpenJML eliminates false positives. Furthermore, OpenJML rarely has false negatives; almost always, when a candidate patch is correct, OpenJML verifies it.

We also pointed out two new problems that can affect APR tools: changes to a program's complexity and numeric problems. These are future work for APR tools.

Acknowledgment

Thanks to Matias Martinez, Martin Monperrus, Jifeng Xuan and Yuan Yuan for help with APR tools. Thanks to Shirin Nilizadeh and Yannic Noller for help with Kelinci. Thanks to John Singleton for contributions to the Java+JML dataset. Dr. Păsăreanu's work was partially funded by NSF Grant 1901136 (the HUGS project).

References

[1] C. Decker and R. Wattenhofer, "Bitcoin transaction malleability and MtGox," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 313–326.

[2] L. Zhang, D. Choffnes, D. Levin, T. Dumitraş, A. Mislove, A. Schulman, and C. Wilson, “Analysis of SSL certificate reissues and revocations in the wake of Heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 489–502.

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[4] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[5] 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 the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 532–543.

[6] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[7] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, “Overfitting in semantics-based automated program repair,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

[8] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, “On reliability of patch correctness assessment,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[9] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[10] A. Nilizadeh, “Java-JML,” https://github.com/Amirfarhad-Nilizadeh/Java-JML, accessed: 2021-1-20.

[11] D. L. Dill and J. Rushby, “Acceptance of formal methods: Lessons from hardware design,” *IEEE Computer*, vol. 29, no. 4, pp. 23–24, 1996.

[12] S. Barner, Z. Glazberg, and I. Rabinovitz, “Wolf–bug hunter for concurrent software using formal methods,” in *International Conference on Computer Aided Verification*. Springer, 2005, pp. 153–157.

[13] M. Martinez and M. Monperrus, “Ultra-large repair search space with automatically mined templates: the Cardumen mode of Astor,” in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.

[14] ——, “Astor: A program repair library for Java,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[15] Y. Yuan and W. Banzhaf, “Arja: Automated repair of java programs via multi-objective genetic programming,” *IEEE Transactions on Software Engineering*, 2018.

[16] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in Java programs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[17] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, “Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 302–313.

[18] D. R. Cok, “OpenJML: JML for Java 7 by extending OpenJDK,” in *NASA Formal Methods Symposium*. Springer, 2011, pp. 472–479.

[19] A. Nilizadeh, “BuggyJavaJML,” https://github.com/Amirfarhad-Nilizadeh/BuggyJavaJML, accessed: 2021-1-20.

[20] T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin, “Automatic program repair using formal verification and expression templates,” in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2019, pp. 70–91.

[21] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[22] R. Könighofer and R. Bloem, “Automated error localization and correction for imperative programs,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 2011, pp. 91–100.

[23] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, “Automatic program repair with evolutionary computation,” *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[24] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[25] “Program repair,” http://program-repair.org/index.html, accessed: 2020-10-14.

[26] B. Jobstmann, A. Griesmayer, and R. Bloem, “Program repair as a game,” in *International conference on computer aided verification*. Springer, 2005, pp. 226–238.

[27] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, “Enhancing automated program repair with deductive verification,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 428–432.

[28] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using SAT,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 173–188.

[29] T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin, “Automatic program repair using formal verification and expression templates,” in *Verification, Model Checking, and Abstract Interpretation*, C. Enea and R. Piskac, Eds. Cham: Springer International Publishing, 2019, pp. 70–91.

[30] B.-C. Rothenberg and O. Grumberg, “Sound and complete mutation-based program repair,” in *International Symposium on Formal Methods*. Springer, 2016, pp. 593–611.

[31] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, “On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs,” in *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.

[32] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[33] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[34] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[35] ——, “Tbar: revisiting template-based automated program repair,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[36] J. Campos, A. Riboira, A. Perez, and R. Abreu, “GZoltar: an Eclipse plug-in for testing and debugging,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[37] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[38] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, “An empirical study of boosting spectrum-based fault localization via PageRank,” *IEEE Transactions on Software Engineering*, 2019.

[39] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.

[40] 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 the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[41] B. Meyer, “Applying ’design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[42] G. T. Leavens and Y. Cheon, “Design by contract with JML,” 2006, available from https://www.cs.ucf.edu/ leavens/JML//jmldbc.pdf.

[43] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580,583, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[44] G. T. Leavens, A. L. Baker, and C. Ruby, “JML: A notation for detailed design,” in *Behavioral specifications of Businesses and Systems*. Springer, 1999, pp. 175–188.

[45] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, “An overview of JML tools

and applications," *International journal on software tools for technology transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[46] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[47] D. R. Cok, "Openjml: software verification for java 7 using jml, openjdk, and eclipse," *arXiv preprint arXiv:1404.6608*, 2014.

[48] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from Defects4J," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.

[49] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 789–799.

[50] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[51] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 254–265.

[52] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 831–841.

[53] M. Zalewski, "Technical" whitepaper" for afl-fuzz," *URl: http://lcamtuf. coredump. cx/afl/technical_details. txt*, 2014.

[54] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[55] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[56] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2020, pp. 1–8.

[57] T. Durieux and M. Monperrus, "Introclassjava: A benchmark of 297 small and buggy java programs," 2016.

[58] 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 Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017, pp. 55–56.

[59] J. R. Kiniry, A. E. Morkan, D. Cochran, F. Fairmichael, P. Chalin, M. Oostdijk, and E. Hubbers, "The KOA remote voting system: A summary of work to date," in *International Symposium on Trustworthy Global Computing*. Springer, 2006, pp. 244–262.

[60] B. Cook, "Formal reasoning about the security of Amazon Web Services," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 38–47.

[61] R. Kersten, K. Luckow, and C. S. Păsăreanu, "POSTER: AFL-based fuzzing for Java with Kelinci," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2511–2513.

[62] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "DifFuzz: differential fuzzing for side-channel analysis," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 176–187.

[63] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 449–452.

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

[65] A. Ghanbari and L. Zhang, "PraPR: Practical program repair via bytecode mutation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1118–1121.

[66] M. R. Hoffmann, E. Mandrikov, and M. Friedenhagen, "Java Code Coverage for Eclipse," https://www.eclemma.org/jacoco/, accessed: 2020-12-14.

[67] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[68] S. A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM Journal on Computing*, vol. 7, pp. 70–90, 1978.

[69] P. Cousot, "Methods and logics for proving programs," in *Handbook of Theoretical Computer Science*, J. van Leewen, Ed. New York: MIT Press, 1990, vol. B: Formal Models and Semantics, ch. 15, pp. 841–993.

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

[71] 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)*. IEEE, 2013, pp. 356–366.

[72] X.-B. D. Le, D. Lo, and C. Le Goues, "Empirical study on synthesis engines for semantics-based program repair," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 423–427.

[73] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "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, pp. 376–379.

[74] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[75] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[76] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 198–216.

[77] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.

[78] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[79] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2019, pp. 1–10.

[80] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[81] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Citeseer, 2015, pp. 166–178.

[82] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Test case generation for program repair: A study of feasibility and effectiveness," *arXiv preprint arXiv:1703.00198*, 2017.

[83] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 226–236.

[84] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.