Avoiding Overfitting in Automated Program Repair using Formal Methods

Amirfarhad Nilizadeh*, Gary T. Leavens*, Xuan-Bach D. Le†, Corina S. Păsăreanu‡, and David R. Cok§

***Abstract*—Automated Program Repair (APR) is used for detecting bug locations in a program and repairing them by creating patches. Most current APR tools are dynamic in the sense that they use a test suite for both detecting buggy statements and validating the correctness of the generated patch. A major problem with such dynamic APR tools is that the patches they generate are sometimes incorrect; that is, the patches are only good enough to pass the tests used for generating the repair but do not generalize beyond those tests. Because such "overfitting" is possible, current practice resorts to manual human judgment about the correctness of patches, which makes the process only semi-automatic, not reusable for future patch verification, and more importantly, subject to human biases. To alleviate these issues, we propose to use formal methods to describe the behavior of a program and automatically verify the correctness of patches. The novelty of this approach is twofold: (1) correctness of patches is automatically guaranteed, and (2) the effort is reusable, as specifications can be written once and reused in the future. We demonstrate this approach by experiments using JML-based specification and verification on patches generated by several open source and well-known APR tools. Our results show that by using formal methods it is possible to completely avoid overfitting with only a small number of false negatives. Our experiments also point out two new problems that can afflict APR tools: changes to the time complexity of programs and numeric problems.**

***Index Terms*—Automated Program Repair (APR), Overfitting, Verification, Formal Methods, JML, Kelinci**

I. Introduction

Bugs exist even in critical software; for example, bugs have been famously found in protocol and security applications, such as blockchain systems [1] and the well-known Heartbleed bug [2]. In addition to developer mistakes, changing requirements of a program can introduce bugs into a previously correct program. However, finding and fixing bugs is expensive. Studies show that about half of the cost of developing a software system is related to testing and debugging [3]–[5]. Also, billion of dollars are spent on debugging annually [6]. Software Fail Watch reported that $1.1 and $1.7 trillion were lost because of software failures in 2016 and 2017, respectively [7], [8]. Thus, detecting and fixing bugs is important for saving costs in software development.

Due to the importance of this problem, researchers have devoted much effort to speeding up the process of debugging by making it automatic and more reliable. Debugging a system has two general steps. The first step is to detect bugs and find their likely cause, which is known as *bug localization* [9], [10]. The second step is to repair the bugs, by patching the program statements that were found to be faulty [3]. Localizing bugs automatically has a longer history than repairing them and localization is already used in industry. However, in recent years research about repairing bugs automatically has received much attention.

Automated program repair (APR) is an area that attempts to make the whole process of debugging automatic. A variety of techniques have been published that are based on symbolic execution [11], genetic algorithms [12], [13], random mutations [14], formal methods [15]–[17], machine learning [18], and other special-purpose techniques. Current APR tools have shown great promise but they still have some limitations. In a 2017 study Martinez et al. [19] investigated how well three APR tools (jGenProg [20], jKali [20], and Nopol [21]) performed on a subset of the defects4J dataset [22]; these tools only generated patches for 21% of bugs in the study. Furthermore, only about 4% of the bugs in the defects4J dataset were patched correctly and the study's authors were unsure about the correctness of another 5% of the patches.

Current APR tools sometimes generate patches that pass all tests but are incorrect, a problem known as *overfitting* (see section III). Current practice follows either of the following methods to validate patch correctness: (1) patches are automatically validated by an independent test suite, or (2) humans manually examine the patches and label their correctness. While the former is automatic, it is incomplete because of the reliance on a test suite. The latter requires extensive manual effort and domain knowledge to make reliable correctness judgements, and more importantly is not reusable for future patch verification. To alleviate these issues, the main idea of our work is to avoid generating incorrect patches by using formal methods. We propose using formal specifications with existing APR tools; in particular our experiments focus on Java and use JML specifications and APR tools that target Java programs. Furthermore, we aim to prove the correctness of repaired programs automatically using static verification. By doing so, we can automatically and objectively verify the correctness of patches while making the patch verification process reusable, as specifications can be written once and reused in the future.

The contributions of this paper are:

1) Formalizing APR, including its comparison with program verification and program synthesis.
2) Preventing APR tools that use a test suite from creating patches that overfit to the test suite by using formal

*Dept. of Computer Science, University of Central Florida, Orlando, Florida 30816-2362, 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, NASA Ames Research Center, USA Email: Corina.S.Pasareanu@nasa.gov §Safer Software Consulting, LLC, Rochester, New York, Email: david.r.cok@gmail.com

methods to reject patches that are not correct.

3) Using a fuzzing tool for generating a test suite, instead of using a test suite written by the developer, and experimentally evaluating its effectiveness. [23].
4) Creating a dataset of Java programs with JML specifications for which OpenJML can prove their correctness [24]. This dataset is the largest open source dataset of Java+JML programs that are statically verified by OpenJML.
5) Creating a dataset of buggy Java programs with their JML specifications [25]. These buggy programs have exactly one bug and OpenJML correctly identifies their invalidity. This dataset can be used by researchers to automatically and objectively evaluate the effectiveness of their APR tools.
6) Experimentally evaluating the reliability of current APR tools for generating a correct patch.
7) Evaluating the effectiveness of OpenJML's extended static checker for classifying overfitting and correct patches.
8) Discovery of two new problems that arise in programs repaired with current APR tools: a) generated patches can increase the time complexity dramatically and b) a generated patch can create an integer overflow. (We discuss the latter and other more general numeric problems in subsection VII-B.)

II. Background

This section introduces terminology related to APR systems. After that, APR is formalized and compared with program verification and program synthesis. Then, five test-based APR tools for Java are introduced: Cardumen, jGenProg, jkali, jMutRepair and Nopol. These tools are used in section VI (Experimental Results) of this paper. Then, formal behavioral specification, the Java Modeling Language (JML) and OpenJML are presented.

A. Automated Program Repair

In general an APR system has two inputs: a buggy program and a description of the expected behavior of the program. This behavioral description is used for localizing bugs in a program and validating (or verifying) generated patches. In most current APR research, this behavioral description is a test suite [26], [27]. However, in some research, a formal specification is used [17], [28], [29]. All open source APR tools that are created by researchers are collected in a website [30] and almost all of them use a test suite as a behavioral description. An APR system, like a debugging system, has two main steps: (1) fault localization and (2) patch generation. Faulty locations are determined by using fault localization techniques, most notably spectrum-based fault localization such as Ochiai [31]. Given the localized faulty elements as input, patch generation involves two phases: (a) generating candidate patches and (b) validating or verifying the correctness of the candidate patches. The process of patch generation and validation (or verification) will continue until it reaches one of the following outputs: either a repaired program that satisfies the behavioral description or a "Not Successful" message indicating that the APR system cannot repair the buggy program. A repair system may be unsuccessful for two reasons. The first reason is a time out, which shows that the process of finding a patch has taken longer than is expected or that a user is willing to wait. The second reason is that all of the possible patches in the space of repairs of the APR technique have been tested and none of them could satisfy the behavioral description given.

Program 1. Buggy Program Absolute

```
public class Absolute {
    public int absolute(int num){
        if(0 <= num)
            return num;
        else
            return num;
    }
}
```

Program 2. Test Suite

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class JUnitAbsolute {
        @Test
        public void test() {
           Absolute a = new Absolute();
           assertEquals(10, a.absolute(10));
        }
        @Test
        public void test1() {
           Absolute a = new Absolute();
           assertEquals(10, a.absolute(-10));
        }
}
```

An example of inputs and output of an APR system for a small program is discussed in the following. In programs 1 and 2 a buggy `Absolute` Java program for integer numbers and its JUnit test suite are shown, respectively. This program and its test suite are the inputs of an APR system. Program 1 has a semantic error for computing the absolute value of an integer number; it does not return a positive value when the input argument of the "absolute" method is a negative integer number. In program 2, two inputs "10" and "-10" describe the behavior of the system; they represent tests with positive and negative integers in Java, respectively. A possible generated patch with an APR system is shown in program 3; it passes both tests successfully.[1] In the generated patch (Program 3) "- return num; %line6" means remove "return num;" (in line 6), and also "+ return -num; %line6" means add "return -num;" (also as line 6 of the "Absolute" class).

[1]None of the five APR tools discussed in this paper can generate a patch for program 1 using its test suite (program 2).

Program 3. Generated Patch

```
- return num; %line6
+ return -num; %line6
```

1) Formalizing APR: APR can be formalized as a function, $APR(P, S, \epsilon)$, where P denotes a program, S is an input/output specification, and ϵ is a maximum distance (number of changes allowed) in the program. The output is either "Yes" along with a program P' that satisfies S and is such that distance$(P, P') \leq \epsilon$, or "No" if there is no such program. In most APR research the distance(P, P') is defined as an edit distance on the abstract syntax trees of P and P'. A distance of one might mean, for example, that one token (e.g., an operator) was changed between P and P'.

In full generality, APR is an undecidable problem, because in general the specification S can have an infinite input domain and thus solving APR exactly is equivalent to solving program equivalence exactly. Note that $APR(P, S, 0)$ is equivalent to verifying the correctness of P with respect to the specification S, which is known as program verification. Also, note that $APR(P, S, \infty)$ is equivalent to program synthesis. Therefore, APR is a very hard problem.

However, in practice, APR systems do not try to solve the APR problem exactly; instead, of answering "No" when there is no program that satisfies the specification, they make no guarantee when they cannot find a patch (i.e., they can answer "Sorry" if they do not find a patch). This allows APR systems to work up to some time out.

A small edit distance corresponds to the *competent programmer hypothesis* [32], [33], which says that "Programmers, however, have one great advantage . . .: they create programs that are close to being correct!" [32]. If programs are nearly correct, then only small changes are needed to make them correct and these changes should correspond to small edit distances. Some research results have used datasets with hundreds of thousands of lines of code [3] [34] and demonstrated good rates of program repair, although the systems involved can only make small changes to programs.

2) Taxonomy of APR: Different classifications of APR tools are introduced in survey papers. In Monperrus's work [35], automated program repair techniques are classified into two main classes: behavioral and state repair. *Behavioral repair* changes the source or binary code of a selected program and *state repair* modifies the state of the program, a technique more like classical fault tolerance [35], [36]. In Gazzola's survey paper [3], APR tools are classified into two different groups: "generate and validate" and "semantics driven repair". The *Generate and Validate* approach has two steps: generating a candidate patch based on the repair space of the APR technique, and then validating the patch by using a test suite as an oracle. The *Semantics Driven* approach analyzes the behavior of the program semantically, by using a test suite or behavioral specification. Finally, Le Goues's work [37] classified APR research into two main classes including "heuristic repair" and "constraint based repair." *Heuristic Repair* uses a generate and test approach for repairing bugs. *Constraint Based Repair* uses symbolic execution, or other approaches, to extract properties of the program, and then generates a patch that satisfies those properties. Also, the paper by Le Goues et al. [37] introduces a new group namely, "learning based repair," which uses machine learning or neural networks for generating a patch. In fact, learning based repair can be used to improve the results of heuristic repair and constraint based repair.

To recognize more recent approaches, we classify APR systems in two main classes, "Static APR" and "Dynamic APR" approaches. The *Static APR* approach generates a patch and verifies the correctness of the repaired program statically by using formal methods. "Static APR" is divided in two classes: "Model Checking" and "Formal Specification." The work of Jobstmann [38] is a Model Checking approach; it uses LTL specifications to patch finite-state programs. The formal specification approach is found in the work of Gopinath, Malik, and Khurshid [15], the Maple system [39], and AllRepair [40].

The *Dynamic APR* approach generates a patch and validates it by using a test suite; typically patches are generated either based on the repair space of an APR technique like GenProg [13] and RSRepair [14], or a semantic analysis of the behavior of a program, as is done in Angelix [41] and Nopol [21]. These two kinds of techniques for generating patches are noted in Gazzola's survey paper [3], which distinguishes between "generate and validate" and "semantics driven" techniques. Figure 1 shows these classifications.

B. APR Tools

In this work five APR tools that operate on Java programs are studied. Four of these tools are selected from Astor [20], [42], which is an automatic software repair framework. These four tools are Cardumen, jGenProg, jKali, and jMutRepair. Also, Nopol is another Java APR tool that is used in our evaluation. We next introduce these tools.

All of the tools we study (and also some datasets) are collected in the program repair website [30]. All of the tools we study use a test suite as a behavioral description, as is the case for almost all of the tools that are available on the program repair website.[2]

1) Cardumen: Cardumen [42], [45] is one of the APR tools recently introduced by the Astor repair framework for Java. It repairs 77 bugs of defects4J dataset [22]; eight of these 77 are not repaired with the other APR tools.

Cardumen uses a spectrum-based fault localization (SBFL) tool for bug localization, GZoltar [46]. SBFL techniques statistically analyze the behavior of the software system based on the failing and passing tests. Then, statements of the program are ranked based on their likelihood of being faulty [47], [48]. It then uses fine-grained program elements and automatically mined repair templates to repair a program.

[2]Only AllRepair [40] uses assertions as a formal specification and AutoFix [43], [44] uses a formal specification as it is based on the Eiffel language. Even AutoFix uses a test suite; however, that test suite is created from the program's formal specification.

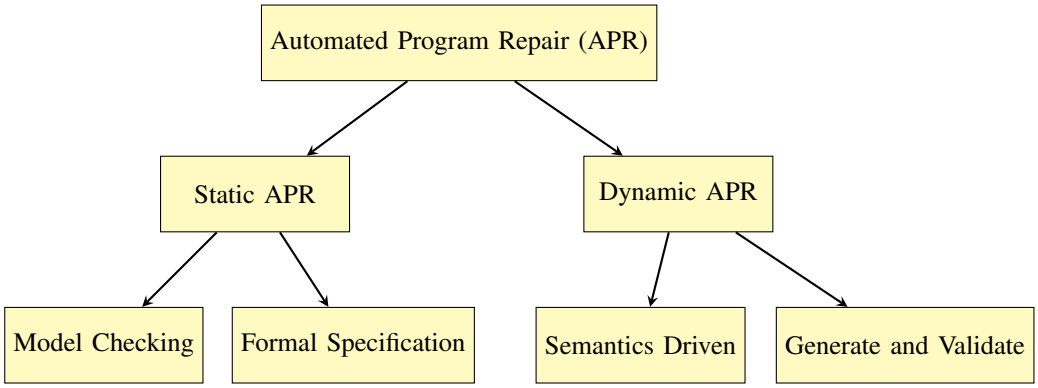

Figure 1. Automated Program Repair Classification

2) jGenProg: GenProg [12] is one of the first and best-known APR algorithms. The Java version of this tool, introduced by the Astor repair framework, is jGenProg [20].

The jGenProg algorithm uses SBFL for bug localization. Then, it uses three operations (insert, remove and replace) for repairing a program. The algorithm uses a measure called "fitness," which evaluates the generated candidate patches. The *fitness of a patch* is the number of tests that pass the repaired program using that patch, minus twice the number of tests that fail. The algorithm uses genetic programming for searching in the repair space of the program; the best two candidates (those with the highest fitness) are used for crossover, which generates new candidate patches. If a program with a candidate patch passes all of the tests, then that patch is considered to be the repair.

3) jkali: This tool is the Java version of kali [49], which is for C. The tool just removes the faulty statement [50]. It uses the SBFL technique [31] for localizing the faulty statements. For each potentially faulty statement, jkali considers removing it, and if the program can pass its test suite, then it considers this change as a repair.

4) jMutRepair: The idea of jMutRepair is to repair a buggy Java program by mutating if conditions. jMutRepair uses three different kinds of mutations for if statements and each time it only makes one change in the if condition. The first is the relational group that can evaluate the six different relational operations (<, <=, >, >=, ==, !=). The second is a logical group that can change "AND" to "OR" or vice versa. The third is a unary group that can apply arithmetic negation to expressions. Again, if a program with a candidate patch passes all the tests in the test suite, then the patch will be considered to be a repair.

5) Nopol: Nopol [21], [51] is an open source APR tool for Java that uses a test suite. Nopol can fix bugs in if conditions and synthesize code to prevent errors in 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

Some enterprises that need to provide program correctness and security are using formal behavioral specifications; examples include Amazon [52], [53], Facebook [54], Intel [55], NASA [56], [57], Rockwell Collins [58], and several railway and subway systems [59]. This indicates the real-world interest in formal methods as well as the availability of relevant formal specifications, which could be used in APR tools.

A formal behavioral specification is a mathematical description of a system at an abstract level. It clarifies the assumptions and final expected state of the system. It does not control the details of the implementation [60]. A formal behavioral specification can be either an executable program or a logical description, which can use mathematics and quantifier annotations. Design by Contract (DbC) [61] is a form of executable specification that uses preconditions, postconditions, and invariants [62]. Many modern systems use DbC notations and add other logical descriptions that are not necessarily executable; these include: ACSL (for ANSI C) [63], JML [64] for Java, Spec# for C# [65], and Dafny [66].

In this work, we use the extended static checker for JML found in OpenJML. Next we explain the details of JML [67], [68] and OpenJML [69].

1) Java Modeling Language: The Java Modeling Language (JML) [70] is a formal behavioral specification language for Java. JML is based on DbC notations like the Eiffel language [62]; these notations provide Hoare-style specification for Java methods and classes. Preconditions of a method in JML are defined by "requires" clauses, and postconditions are defined by "ensures" clauses. Also, JML specifications can be used at the class level by using "invariant" clauses, and at the statement level, by using "assert", "assume" and "maintaining" clauses.

A JML annotation is written after "//@" on a line or between "/*@" and "@*/" in a Java program. Thus, the Java compiler considers these annotations as comments. Tools based on JML can compile both JML and Java.

Many tools have been created that use JML notations for type-checking, generating tests, run-time assertion checking, static analysis, and verification [68], [71]. Two static verification tools based on JML are OpenJML [72] and KeY [73]. The extended static checker of OpenJML is automatic and does not

Program 4. Unsatisfied absolute

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

Program 5. Satisfied absolute

```
public class Absolute{
    //@   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;
    }
}
```

support direct user interaction, whereas KeY is designed to be a more interactive verifier [74]. In this paper we use OpenJML, as it is more automatic; OpenJML has been used in Amazon Web Services (AWS) for verifying their software [52].

2) OpenJML: The OpenJML tool [69] supports both static verification and run-time assertion checking [75]. The two inputs of OpenJML are Java code and its JML behavioral specification. Static checking in OpenJML can prove the correctness of a program for all possible inputs.

In the static verification process of OpenJML, the Java code and JML specification are translated into first order logic verification conditions for an SMT solver.[3] Then the SMT solver is applied to the verification conditions [79].

In program 4, the repaired `Absolute` class for integer numbers is shown with JML annotations after applying the generated patch (program 3). The Java code returns the unchanged input argument if the input is greater or equal to zero. Otherwise, if the input value is less than zero, then the method returns the negated input value. The JML specification in program 4 has the same behavior; it has two different scenarios for the possible input values, which are separated by the **`also`** annotation. The specification above the **`also`** describes the case when the input is positive; in this case, the return value (which is written `\result`), must be equal to the input value. In the second case (following **`also`**), the input value is less than zero in Java. In this case, based on the postcondition the return value must be equal to the negation of the input.

However, running OpenJML's static verifier on the repaired `Absolute` program generates the following warning.

```
Absolute.java:13: warning: The prover cannot
establish an assertion (ArithmeticOperationRange)
in method absolute: (int negation)
            return -num;
1 warning
```

This shows that the Java method does not satisfy its contract, because Java represents the **`int`** type using two's complement notation, and thus it cannot represent the negation of "`Integer.MIN_VALUE`" as a positive "**`int`**."[4] Note that the arithmetic in the specification is evaluated using mathematical integers, while the Java program uses the wrap-around semantics of Java's fixed-bit-width integers.

To have a correct program it is thus necessary to limit the domain of the program with a precondition that does not allow `Integer.MIN_VALUE` as an argument; that version is shown in program 5. The only change compared to program 4 is revising the second precondition, which says that the argument cannot be equal to `Integer.MIN_VALUE`. The correctness of this program can then be verified statically with OpenJML, which is important for safety critical systems.

III. The Overfitting Problem in Dynamic APR

A test suite is usually not a complete specification, because a program may have an infinite domain. Therefore, when an APR tool uses a test suite for validation it may not verify the correctness of a program for the program's entire domain. Thus, the generated patches may not be reliable and using them automatically without evaluating the correctness of the patch is unwise, especially in safety critical systems. *Overfitting* means that the generated repair patch can pass the test suite but still not be a correct patch based on the requirements of the program.[5]

Evidence for the problem of overfitting comes from the work of Martinez et al. [19]. That work used the defects4J dataset [22], which is a public Java dataset with 438 real world buggy programs and their test suites. Martinez et al. used a subset of defects4J with 224 real bugs to evaluate three APR tools in Java including: jGenProg [20], jKali [20] and Nopol [21]. Their results showed that these three APR tools were not very successful on these real world bugs. Also, most of the patches were not correct, because of the overfitting problem. These tools generated patches for 47 bugs (21% of the bugs). However, only 9 generated patches were correct (about 4%) and Martinez et al. were unsure about the

[3]The SMT solver which is used by default in OpenJML is Z3 [76], but also CVC4 [77] and Yices [78] can be used.

[4]In Java, the negation of "`Integer.MIN_VALUE`" is itself, which is still negative.

[5]The term "overfitting" [80] is borrowed from machine learning, and long before that from statistics. It applies to all APR techniques that are using the dynamic APR approach (see Figure 1).

correctness of 11 further patches (about 5%). The other 27 generated patches (about 12% of the total number of programs) could pass the unit tests, but were overfit patches.

Another empirical evaluation, by Le et al. [81], on the "Semantic Driven" approach showed that even APR studies, which were using the behavior of the program (when running the test suite) for generating the patches, suffered from the overfitting problem.

A better test suite can describe the behavior of a system more completely. Thus, tools that can generate more complete test suites can help solve the overfitting problem. For example, the symbolic execution tool KLEE [82] is used by Smith et al. [80] to generate a test suite that they call a "white-box" test suite. Developers generate a test suite that they call a "black-box" test suite. Smith et al. then evaluate GenProg [13] and RSRepair [14] by using these two different kinds of test suites: black-box and white-box. Their results show that most of the generated patches that are created with the black-box test suite do not pass the white-box test suite. In their work the generated patches that are created by using symbolic execution are more reliable. Also, Yang et al. [83] create a framework named Opad (Overfitted PAtch Detection) that increases the number of tests in a test suite by generating new test cases, using American Fuzzy Lop (AFL) [84]. Their results show that this tool could filter more than 75% of generated overfitting patches with their new test cases compared to their initial test cases. In another study, Xiong et al. [85] show that APR tools become more reliable and they generate less overfitting patches by increasing the number of test cases. Their work could filter more than 56% of generated overfitting patches.

These results show that even increasing the number of tests does not solve the overfitting problem in the dynamic APR approach. Thus, generated patches are not reliable and a developer has to manually evaluate the correctness of each generated patch. That is, the developer makes a decision about accepting or discarding the generated repair patch. Thus, in dynamic APR techniques the whole process of APR is not completely automatic. Also, evaluating the correctness of a patch is difficult and time- consuming. Without a specification, only the developer will have enough information about the program to evaluate correctness. For example, in the Martinez et al. work [19] the authors, who were not the program's developers, were unsure about the correctness of 11 out of 47 patches.

However, by increasing the number of test cases the performance of APR will be degraded. Running tests is the most expensive operation in dynamic APR tools [3], [14], [86]. There is a trade off between the number of tests in a test suite and the performance and reliability of an APR system. Le Goues [87] shows that 64% of the time of the GenProg process for detecting bugs and generating a repair patch is related to running the tests. Thus there is a dilemma between reliability and performance in dynamic APR.

IV. Approach to Avoiding Overfitting

While testing a program is a necessary part of validation, a formal specification is necessary for verifying correctness [88]. Recall that a test suite as used in dynamic APR for validation, but testing does not usually provide verification. This leads to the overfitting problem that was discussed in section III. Due to overfitting, a developer must manually check the correctness of patches.

In dynamic APR systems there is no realistic approach for completely solving the overfitting problem. The only test suite that can verify the correct behavior of a system completely is a test suite with all possible inputs. Only such a test suite can guarantee the correctness of a program. However, having a complete test suite is unrealistic for most real-world programs.

The approach of this paper for solving the overfitting problem is to use a formal behavioral specification. A formal behavioral specification with pre- and postconditions and invariants can be used to automatically make a decision about the correctness of a repaired program. In the first step of this approach, a dynamic APR tool using a test suite generates a candidate patch for a buggy program. Then the candidate patch is validated by using its test suite. If the generated candidate patch cannot pass the test suite, then it backtracks to generate another candidate patch. This process will continue until it generates a candidate patch that is validated by its test suite, or it times out or runs out of potential repairs. If it generates a patch, then the validated, repaired program and its formal behavioral specification are passed to the verification process as inputs. In the verification process, the repaired program and its specification are evaluated with a static analyzer. If the static analyzer verifies the correctness of the repaired program, then the correctness of the repaired program is guaranteed. Otherwise the tool will tell the user that no correct patch could be found. (In future work, we plan to have the tool backtrack and attempt to generate another patch.) By using validation and verification, overfitting cannot happen. This would allow APR to be used for safety critical systems. Also, by using a static analyzer instead of a human for deciding the correctness of candidate patches, the whole process of APR becomes automatic. Figure 2 shows our proposed validation and verification approach.

This approach to solving the overfitting problem for dynamic APR can be formalized as two uses of the *APR* function, $APR(APR(P, S, \epsilon), S', 0)$. The inner *APR* call is the validation process and the outer APR call is the verification process. In the inner *APR* call, P denotes a program, S is a test suite for validation[6] and ϵ is a maximum edit distance in the program (number of changes allowed) as discussed in section II-A. In the outer *APR* call the first argument, $APR(P, S, \epsilon)$, is a validated repaired program, the second argument, S', is a formal behavioral specification and "0" denotes the maximum edit distance in the program. Recall from section II-A that an APR system with zero edit distance performs program verification.

The output of the inner call to *APR* can be either a program P' that satisfies S (test suite) and is such that distance$(P, P') \leq \epsilon$, or a "Not Successful" message. If the answer is a program, then the output of the outer call to

[6]The test suite, S, should correspond to the formal specification in the sense that each input-output pair in the test suite should be correct according to S'.

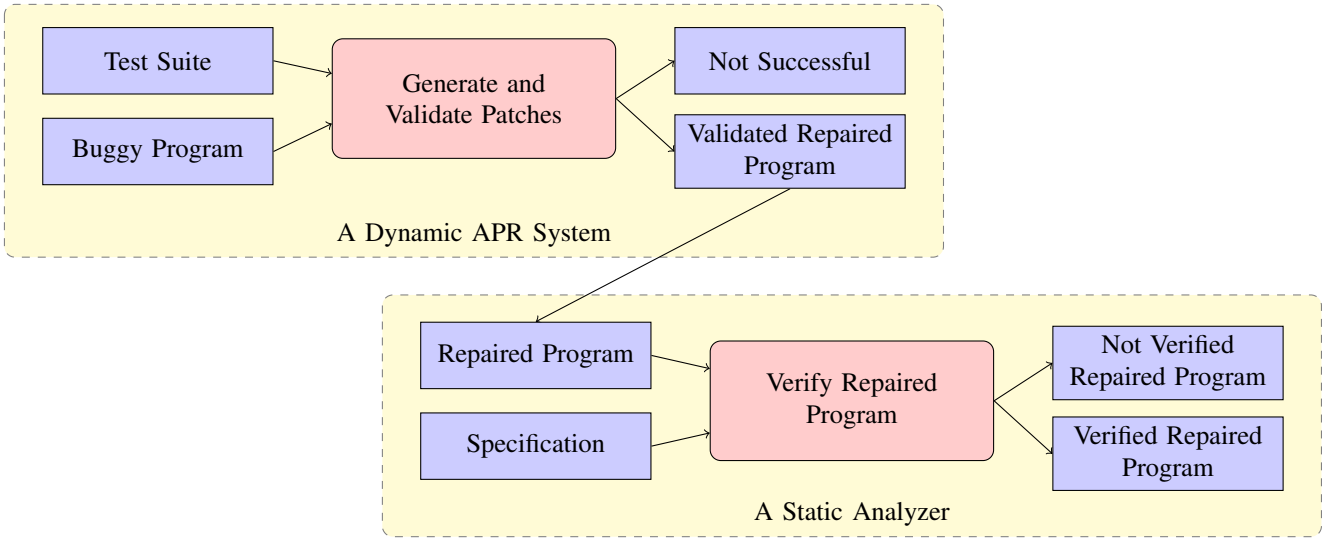

Figure 2. An Abstract Diagram of using Validation and Verification for Avoiding Overfitting

APR can be either the program ($APR(P, S, \epsilon)$), which is thus verified to satisfy S' (the formal behavioral specification), or "Not Verified", which means $APR(P, S, \epsilon)$ is an overfit program. Also, the output of "Not Successful" from the inner *APR* call means that the APR system cannot generate a patch satisfying S. The verified program output from outer *APR* call means that the repaired program is both validated by S and verified by S'.

V. Creating Datasets

To carry out our experiments, we created two datasets. The first is a Java+JML set of verified programs. The second is a buggy Java program dataset, which is derived from the first.

The Java+JML verified programs are a set of 20 Java programs that are all verified using OpenJML's static analyzer to be correct with respect to their JML specifications.

The buggy Java program dataset is based on the Java+JML verified programs. For each verified program P there is a test suite for P and several buggy variants of P. Each variant was created by mutating P to inject exactly one bug into it. This buggy Java program dataset has 190 buggy variant programs overall.

To avoid bias in the evaluation and experimental results, the process of generating each test suite and injecting bugs was automated by using a fuzzing tool and a mutation tool, respectively.

A. Creating a Java+JML Dataset

Existing examples of Java+JML programs, like the KOA voting system [89], cannot be verified by the static checker of OpenJML [69], [79]. There are two reasons for this: (1) the static checker in OpenJML has some limitations at present, such as not supporting Java's double and float types, and (2) the specifications for these existing programs were designed for run time assertion checking and are not entirely suitable for static checking.

Currently Amazon is using OpenJML for proving correctness for AWS [52], but their libraries are not open source. Thus there was previously no open source Java+JML dataset aimed at static verification. Only seven small programs are available in the OpenJML website [90], and six of them are used in the Java+JML verified program dataset. Also, 14 more Java+JML programs were created for this research and added to the verified program dataset. These 14 Java programs are small programs that are mostly used for teaching Java to beginners, such as factorial, bubble sort, linear search, leap year, and prime numbers. The new Java+JML dataset contains small programs, but the programs are well-known. It makes sense to begin evaluation of APR tools with such small programs before moving on to larger programs, because the small programs will help developers and researchers understand the process we are proposing, including the formal specifications and the evaluation of the correctness of the repaired programs.

This Java+JML verified program dataset is available on a GitHub website [24].

B. Creating Test Suites

Dynamic APR tools take as input a test suite. There is an assumption in work on dynamic APR that at least one of the tests in a buggy program's test suite must trigger the bug in the program.[7] Past research shows that test suite quality is important for dynamic APR tools, and a stronger test suite can repair more buggy programs with less overfitting (see Section III). To avoid bias in the evaluation and comparison of APR tools, we automate the process of creating the test suite. For this purpose we use an off-the-shelf fuzzing tool, Kelinci [91], [92].

Kelinci is a fuzzing tool for Java, based on American Fuzzy Lop (AFL) [84]. Kelinci automatically generates files (sequences of bytes) in an attempt to cover all the paths of the program being tested. The user must provide a driver that

[7]This assumption is reasonable, as bugs are only identified through testing.

converts the sequence of bytes generated by Kelinci into the program's arguments. The driver is written by considering the JML precondition of each program (drawn from the dataset of Java+JML verified programs). In this paper we run Kelinci five times for each program in the dataset of verified programs; each run of Kelinci was used to generate at least one JUnit test, each of which is based on a sequence of bytes that Kelinci discovered as covering some path in the program. Test suites are formed from these tests by removing tests that are equivalent to other tests. Thus there is one test suite for each verified program.

The JUnit test suites generated in this way are available on GitHub [93].

C. Creating a Buggy Program Dataset

To avoid bias in the evaluation and comparison of APR tools, the process of creating the buggy program dataset is also automated. Since most of the dynamic APR tools can only fix a buggy program with one bug, we wanted to create a buggy dataset in which each program has exactly one bug. PITest, [94]–[96], a well-known mutation tool for Java programs, is used for injecting a single bug into each verified Java program. This process generated the buggy mutants that were added to our dataset of buggy Java programs.

In our work the inputs for PITest are all of the verified programs in the Java+JML verified program dataset (along with each program's JUnit test suite).

In total PITest created 197 unique buggy variants of our Java+JML dataset programs; each of these 197 buggy programs has exactly one bug. At first PITest created more than 197 mutants, but some of them were either semantically equal to other generated buggy programs or to the corresponding verified program; these equivalent programs were discarded. Also, OpenJML's static analyzer was able to find at least one warning for each of these 197 buggy programs (based on the specification of the correct version of the program in the Java+JML dataset of verified programs). These 197 unique buggy programs are available on GitHub [23], [25].

Among these 197 buggy programs, three of the bugs were not triggered with the test suite for the correct program that was created by Kelinci; these were: "PrimeNumbers bug3," "GCD bug7," and "GCD bug8." Recall there is an assumption in dynamic APR that at least one of the tests in the test suite must trigger the bug. Thus, these three buggy programs were not used in our experimental evaluation. Also, four other buggy programs go into an infinite loop when running their test suite: "BinarySearch bug3," "BinarySearch bug7," "BinarySearch bug8," and "PrimeNumbers bug9." Going into an infinite loop is not acceptable for dynamic APR tools when running their test suite, as the tools will time out. In total, these results show that Kelinci was successful in generating suitable test suites for dynamic APR. It generates a suitable test suite for 190 out of 197 buggy programs (about 96%).

In sum, 190 buggy programs are available in the buggy program dataset, each program has only one bug, and the injected bug can be triggered with at least one of the JUnit tests; furthermore, these buggy programs do not go into an infinite loop when running their test suite. Also, a JML specification is available for each program that can evaluate a program's correctness after a repair.

VI. Experimental Results

In this section our approach to avoiding overfitting is evaluated. The first part of our assessment is an evaluation of five APR tools for Java (those introduced in section II-B). Then the validated programs are evaluated for correctness by using the OpenJML program verifier. We also discuss the possibility of false negatives and false positives that may result from the use of OpenJML to verify repaired programs.

The discussion of OpenJML also points out two new problems that other APR research does not consider: changing the expected time complexity of programs and numeric problems. These are discussed in section VII.

A. Validation using Dynamic APR Tools

In this section we evaluate the first phase of our approach to avoiding overfitting in existing APR tools. Five open source dynamic APR tools are used for this evaluation: Cardumen, jGenProg, jKali, jMutRepair, and Nopol. We applied each of these tools to each buggy program in our buggy program dataset; Table I shows the results. Generated patches in this step are validated by their test suites. Recall that test suites are generated by Kelinci.

Table I
Validation Results for APR Tools

APR Tools	Not Repaired	Validated
Cardumen	164	26
jGenProg	171	19
jKali	184	6
jMutRepair	170	20
Nopol	174	16

The data in Table I shows that each of these dynamic APR tools could not repair most of the buggy programs. In total these tools generated 87 patches. In some cases several APR tools generated a patch for one buggy program, for example all five tools generated a patch for program "BinarySearch bug13," although their generated patches were different from each other. Also, in some cases only one of the tools generated a patch, for example program "Calculator bug6" was repaired only with the Cardumen tool.

In total these five APR tools together generated a validated patch for 46 individual buggy programs in our dataset, which is about 24%.

B. Verification Results using OpenJML

To detect overfitting, we verified the validated repaired programs using OpenJML and the JML specifications that were created for our Java+JML verified program dataset (see section V-A).

Table II shows the results using OpenJML's static analyzer to verify the correctness of the repaired programs from each APR tool. (Recall that in this evaluation, if the validated

repaired program cannot be verified with OpenJML's static analyzer, then the process does not backtrack to ask the APR tool for another patch.)

Table II
VERIFICATION OF VALIDATED PATCHES USING OPENJML

APR Tools	Validated	Verified	Not Verified
Cardumen	26	18	8
jGenProg	19	10	9
jKali	6	5	1
jMutRepair	20	20	0
Nopol	16	15	1

The results shown in Table II show that most of the validated repaired programs were correct, which is a tribute to both the APR tools and the effectiveness of the test suites created with Kelinci. Based on the results, 68 out of 87 validated repaired programs were verified with respect to their formal behavioral specification, which is about 78%; thus 19 repaired programs (about 22%) were not verified and identified as overfitted.

With our validation and verification approach the correctness of 38 individual repaired programs verified by using these 68 validated and verified repaired programs, because in some cases different APR tools repaired the same buggy program. However, in some cases only one of the validated repaired programs was verified by OpenJML's static analyzer. For example three APR tools (jGenProg, jkali and jMutRepair) generated individual repaired programs for program "Smallest bug1" in our buggy dataset. However, the validated repaired programs of jGenProg and jkali were not verified, and only jMutRepair's output was verified with OpenJML.

In total, among the 190 buggy programs in the dataset, 38 of them were repaired correctly by at least one APR tool. Thus, these five APR tools together generated at least one validated and verified patch for 20% of the buggy programs.

C. Evaluating OpenJML's Precision and Recall

It is important to evaluate the extent to which OpenJML correctly classifies patches as "Verified" and "Not Verified". OpenJML is sound, so it will not verify a repaired program that is not correct, thus there are no false positives; we manually checked the "Verified" repaired programs, mentioned in Table II, and found that all of them were in fact correct.

However, evaluating the "Not Verified" repaired programs is more complicated and shows some interesting results. Table III shows the result of evaluating the "Not Verified" repaired programs from Table II.

Table III
RESULTS FOR OPENJML'S VERIFICATION ATTEMPTS

APR Tools	Not Verified	Overfitting (True Negatives)	Not Overfitting (False Negatives)
Cardumen	8	5	3
jGenProg	9	5	4
jKali	1	1	0
jMutRepair	0	0	0
Nopol	1	1	0

In Table III, 19 repaired programs are in the "Not Verified" class. After our evaluation 12 of them are absolutely overfitting: they do not work correctly for all input domain values based on their behavioral specification from the Java+JML dataset. However, the other 7 patches are correct for all inputs allowed by the program's input domain, and are not overfitting. On the other hand, 5 of these seven need more discussion, although they correctly realize the specified relation between inputs and outputs. In two repaired programs the order of time complexity is changed dramatically compared to the original version of programs in Java+JML dataset (see subsection VII-A). Also, integer overflow can happen in three of the repaired programs, which prevents OpenJML from verifying these programs (see subsection VII-B).

Two other repaired programs were not verified by OpenJML, but not for the reasons mentioned above. Instead they are not verified because the structure of the programs was changed by the APR tools and JML would require added loop invariants. These programs, "Factorial bug2" and "BinarySearch bug2" are repaired with jGenProg and Cardumen, respectively. Program 6 is the buggy program of "Factorial bug2". Modifying the condition of "for loop" from "<" to "<=" is the simplest patch in program 6. However, the jGenProg tool repaired this buggy program by adding a new "for loop" as a patch, as shown in program 7. The repaired program looks strange, but it is correct for its entire domain, which is the integers between 0 and 20 (to avoid overflowing the result). Also in the second example, the generated repaired program of "BinarySearch bug2" with Cardumen was not verified by OpenJML. In this case, the modification before entering the loop increased the time complexity by a constant, but it did not change the order of the time complexity.

The false negatives that result in changing the time complexity of the program could conceivably be addressed by writing specifications that address the time complexity of the programs, but such a fix is not easy for JML and is likely to remain a source of incompleteness for that specification language. The false negatives that result from integer overflows are due to the verification logic that JML uses, which considers such overflows to be bugs, although as in these three programs the code was actually correct. This incompleteness in JML is likely to remain because the designers of JML believe that overflow is more likely to be a sign of a bug than to be used correctly.[8] The issue of overflow and underflow (loss of precision) needs to be further addressed for floating point numbers in JML, and thus it is likely that numerical problems will continue to be a source of false negatives, due to the verifier's incompleteness.

In summary, based on our results the presented validation+verification approach for avoiding overfitting is sound, but not complete. Thus there are no false positives, but some false negatives that result from using a verifier to check for overfitting. For the reasons mentioned above, it would be

[8]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.

Program 6. Program "Factorial Bug2"

```
public class Factorial {
       public long Facto(int n)
       {
          int c;
          long fact = 1;
           if ( n == 0) {
              return fact;
           }
          for (c = 1; c < n; c++){
                fact = fact*c;
             }
          return fact;
         }
   }
```

Program 7. Repaired Program by jGenProg

```
public class Factorial {
       public long Facto(int n)
       {
          int c;
          long fact = 1;
           if ( n == 0) {
              return fact;
           }
          for (c = 1; c < n; c++){
              for (c = 1; c < n; c++){
                  fact = fact*c;
              }
              fact = fact*c;
          }
          return fact;
         }
   }
```

difficult to eliminate these false negatives.[9]

VII. 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 the time complexity of the repaired program or introducing potential numerical problems. These issues occurred in our experiments, but to our surprise they were not detected by the programs' test suites. We will discuss them in detail in the following subsections.

A. Time Complexity

Our results show that time complexity can change and increase dramatically by using current dynamic APR tools (at least by using Cardumen). Increasing the time complexity is not an overfitting problem, because the expected behavior is still correct. However, it could be a significant issue.

The time complexity changed from $O(log(n))$ to $O(n/2)$ in two generated repaired programs by using Cardumen:

[9]In theory, since Hoare logic is inherently incomplete [97], [98], it will be impossible to eliminate all potential false negatives.

Program 8. Binary method of "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 + (high - low) * 2;
        }
        if (low >= high){
            return -1;
        }
        return mid;
    }
}
```

"BinarySearch bug9," and "BinarySearch bug10" programs in our buggy dataset. These two repaired programs pass all of the tests and they are even correct for all possible inputs. However, in reality changing the order of time complexity of "Binary Search" from $O(log(n))$ to $O(n/2)$ changes the character of the program from binary search to a linear search. Program 8 shows the "Binary" method of "BinarySearch bug9" program. Here the simplest patch is "`mid = low + (high - low) / 2;`" instead of "`mid = low + (high - low) * 2;`" in the program. Also, the "Binary" method of program 9 is the repaired program by Cardumen tool; its time complexity is $O(n/2)$. The static analyzer of OpenJML does not prove the correctness of the program, because the changes in the loop invalidate the program's loop invariant when the order of time complexity is changed. Thus, to prove their correctness it would be necessary to update the loop invariant in the "repaired" program.[10]

B. Numeric Problems

There are several potential numeric problems with programs that are hard to test, because they may only occur in a small number of cases. 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 specifically specified to be an intended overflow.

Therefore, it may be important that APR tools only generate repairs that do not cause integer overflow, because sometimes such overflow can be harmful [99], [100].

[10]Future tools that infer loop invariants from the program would mitigate this problem.

Program 9. Repaired Program by Cardumen

```
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;
    }
}
```

In the three programs repaired by jGenProg in our dataset integer overflow can happen in some scenarios, including repaired version of "AddLoop bug2," "AddLoop bug7," and "'BinarySearch bug2." However, these integer overflow problems are harmless and they have no effect on the final results. Thus, we cannot consider them as an incorrect patches. However, not all integer overflows are benign and harmless, they may cause bugs (or overfitting). This shows that a dynamic APR tool could create a repaired program with an integer overflow (as jGenProg does).

Program 10 and program 11 show the "AddLoop" method of "AddLoop bug2" and its repaired version with jGenProg, respectively. The simplest patch for program 10 is using "**while** (n > 0)" instead of "**while** (n >= 0)". Integer overflow in program 11 happens for "sum", when add of "x" and "y" is equal to the maximum 32-bit signed integer.

2) Floating Point Problems: Although OpenJML is not able to verify programs that use floating point arithmetic (at the time of this writing), floating point arithmetic could cause similar problems to integer overflow if the verification logic is conservative. That is, there may be some programs where there is the possibility of floating point problems, such as underflow or overflow, but those problems do not actually occur, and thus the verification logic would generate a false negative. This is an area of future work for our approach.

VIII. Related work

Overfitting is one of the main problems of dynamic APR. Recall that, in dynamic APR, the test suite is used for bug localization and for validating the generated patches. Thus, several approaches in related work try to solve the overfitting problem by creating a better test suite that describes the behavior of a system more completely.

Program 10. AddLoop method of "AddLoop bug2"

```
/*@ 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;
}
```

Program 11. Repaired Program by jGenProg

```
/*@ 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;
}
```

For instance Yang et al. [83] 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) overfitted patches that were generated by GenProg/AE [101], Kali [49], and SPR [102]. Note that the APR tools and AFL target C. Yu et al. [103] describe related techniques targeting Java programs, using the Defects4J dataset; they show that generating new tests can lead to fewer overfitted repairs, however their work is not effective in generating correct new patches.

Xiong et al. [85] 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 [104] and HDRepair [105]. The larger test suite leads to 56.3% fewer overfitted repaired programs, but their approach does still have some overfitting.

All of the above techniques try to avoid overfitting by improving the completeness of the test suite; these approaches lead to less overfitting. However, improving the test suite does not completely prevent overfitting. Recall that a test suite cannot be complete unless it covers all possible inputs, which is unrealistic for most of the programs. In contrast, we show that by using a formal behavioral specification, we can prevent overfitting, and can also prove the correctness of the repaired programs. Thus, this approach could be used in safety critical systems. Also, a specification is reusable: it can be written once and reused for any future changes in a program.

IX. Conclusion

Dynamic APR uses test suites for validating automatically generated patches. However, for many real programs (those with infinite input domains) a test suite cannot be a complete specification for describing the program's behavior. Thus, dynamic APR suffers from overfitting. That is, an APR tool may generate a repaired program that can pass all of the tests, but it is not correct based on the program's requirements. Existing approaches aim to prevent overfitting by increasing the number of tests, which makes the test suite a more complete specification of the program's desired behavior. While this helps in certain situations, it cannot solve the problem completely.

We created a dataset of 20 Java programs with their JML specifications, and we verified their correctness with OpenJML. Then, we generated a test suite for each program by using a fuzzer, Kelinci. The use of Kelinci was successful because about 96% of the tests created by Kelinci satisfied the assumptions of the APR tools. Using these 20 verified programs we created a dataset of 190 buggy programs, using a mutation tool, PITest, such that each buggy program has exactly one bug and for each buggy program, OpenJML gives at least one warning. This dataset is the first such dataset derived from verified programs, and thus the first for which it is guaranteed that there is exactly one bug per program. Since the test suites and mutants were created by tools, this dataset is unbiased towards any APR tool.

The core idea of this paper is that using formal methods solves the overfitting problem. Furthermore, formal specifications are reusable. We experimentally validated this idea using five APR tools and JML specifications.

The five APR tools created 87 validated patches for 46 unique programs (about 24% of the buggy programs). Out of the 87 validated patches, 19 (about 22%) were overfitted. The process of verifying the patches with OpenJML completely avoids overfitting. OpenJML did have a few false negatives (programs that OpenJML rejected but which were actually correct); however, some of these false negatives were not completely correct because the order of their time complexity was dramatically increased. A few other false negatives resulted from incompleteness of the verification process in OpenJML.

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

Future work for our approach involves creating a tool that uses formal specification and verification to automatically repair programs, and which can backtrack to generate new patches when verification fails. Another idea is to use information from the formal specification to synthesize correct patches for buggy programs directly, avoiding the need for a test suite.

Acknowledgment

Thanks to Matias Martinez, Martin Monperrus, and Jifeng Xuan for help with APR tools. Thanks to Elaine Weyuker and Tom Ostrand for discussions about decidability.

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] 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.

[4] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 55–64.

[5] T. Britton, L. Jeng, G. Carver, and P. Cheak, "Reversible debugging software "quantify the time and cost saved using reversible debuggers"," 2013.

[6] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, pp. 429–489, 2002.

[7] "Software fail watch says 1.1 trillion in assets affected by software bugs in 2016," https://www.tricentis.com/news/software-fail-watch-says-1-1-trillion-in-assets-affected-by-software-bugs-in-2016, accessed: 2019-07-25.

[8] "White papers software fail watch: 5th edition," https://www.tricentis.com/resources/software-fail-watch-5th-edition/, accessed: 2019-11-17.

[9] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 286–295.

[10] 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.

[11] 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*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 376–379. [Online]. Available: http://doi.acm.org/10.1145/3092703.3098225

[12] 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.

[13] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[14] 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.

[15] 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.

[16] R. Könighofer and R. Bloem, "Repair with on-the-fly program analysis," in *Haifa Verification Conference*. Springer, 2012, pp. 56–71.

[17] 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.

[18] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[19] 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.

[20] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[21] 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.

[22] 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.

[23] "Java-jml-datasets-apr/not_suitable_buggy_programs," https://github.com/Amirfarhad-Nilizadeh/Java-JML-Datasets-APR/tree/master/Not_suitable_Bugg_Programs, accessed: 2020-01-21.

[24] "Java-jml-datasets-apr/java+jml," https://github.com/Amirfarhad-Nilizadeh/Java-JML-Datasets-APR/tree/master/Java%2BJML, accessed: 2020-01-21.

[25] "Java-jml-datasets-apr/buggy_programs_pitest," https://github.com/Amirfarhad-Nilizadeh/Java-JML-Datasets-APR/tree/master/Buggy_Programs_Pitest, accessed: 2020-01-21.

[26] 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.

[27] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 471–482.

[28] 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.

[29] 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.

[30] "Program repair," http://program-repair.org/index.html, accessed: 2019-08-20.

[31] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[33] D. Le, M. A. Alipour, R. Gopinath, and A. Groce, "Mucheck: An extensible tool for mutation testing of haskell programs," in *Proceedings of the 2014 international symposium on software testing and analysis*. ACM, 2014, pp. 429–432.

[34] 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.

[35] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[36] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[37] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, 2019, to appear.

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

[39] 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.

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

[41] 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.

[42] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[43] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 61–72.

[44] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer, "Automated program repair in an integrated development environment," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 681–684.

[45] 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.

[46] 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.

[47] 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.

[48] R. Abreu, W. Mayer, M. Stumptner, and A. J. van Gemund, "Refining spectrum-based fault localization rankings," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 409–414.

[49] 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*. ACM, 2015, pp. 24–36.

[50] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.

[51] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. ACM, 2014, pp. 30–39.

[52] B. Cook, "Formal reasoning about the security of amazon web services," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 38–47.

[53] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Model checking boot code from aws data centers," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 467–486.

[54] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 2018, pp. 13–25.

[55] C.-J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.

[56] A. Narkawicz and C. A. Munoz, "Formal verification of conflict detection algorithms for arbitrary trajectories." *Reliable Computing*, vol. 17, no. 2, pp. 209–237, 2012.

[57] A. E. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson, "Verification of numerical programs: From real numbers to floating point numbers," in *NASA Formal Methods Symposium*. Springer, 2013, pp. 441–446.

[58] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods Symposium*. Springer, 2012, pp. 126–140.

[59] "Prover certifier," https://www.prover.com/software-solutions-rail-control/prover-certifier/, accessed: 2019-08-21.

[60] G. T. Leavens and Y. Cheon, "Design by contract with jml," 2006.

[61] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[62] ——, "Eiffel: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.

[63] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "Acsl: Ansi c specification language," 2008.

[64] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1127878.1127884

[65] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: An overview," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 2004, pp. 49–69.

[66] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[67] 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.

[68] 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.

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

[70] G. T. Leavens, A. L. Baker, and C. Ruby, "Jml: a java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. Citeseer, 1998, pp. 404–420.

[71] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *OOPSLA 2000 Companion, Minneapolis, Minnesota*. ACM, Oct. 2000, pp. 105–106. [Online]. Available: ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz

[72] J. Sánchez and G. T. Leavens, "Static verification of ptolemyrely programs using openjml," in *Proceedings of the 13th workshop on Foundations of aspect-oriented languages*. ACM, 2014, pp. 13–18.

[73] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, "Deductive software verification–the key book," *Lecture Notes in Computer Science*, vol. 10001, 2016.

[74] J. Boerman, M. Huisman, and S. Joosten, "Reasoning about jml: Differences between key and openjml," in *International Conference on Integrated Formal Methods*. Springer, 2018, pp. 30–46.

[75] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with jml and esc/java2," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 342–363.

[76] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[77] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[78] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[79] D. R. Cok, "OpenJML: Software Verification for Java 7 using JML, OpenJDK, and Eclipse," *Workshop on Formal Integrated Development Environments. EPTCS 149*, 2014.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2019-08-20.

[85] 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.

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

[87] 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.

[88] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[89] 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.

[90] "Does your program do what it is supposed to do?" http://www.openjml.org/, accessed: 2019-11-28.

[91] 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.

[92] 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.

[93] "Java-jml-datasets-apr/junitdataset," https://github.com/Amirfarhad-Nilizadeh/Java-JML-Datasets-APR/tree/master/JUnitDataset, accessed: 2020-01-21.

[94] 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.

[95] M. Delahaye and L. Du Bousquet, "A comparison of mutation analysis tools for java," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 187–195.

[96] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, 2018.

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

[98] 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.

[99] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, p. 2, 2015.

[100] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," *Acm sigplan notices*, vol. 49, no. 1, pp. 439–452, 2014.

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] 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.