Empir Software Eng
https://doi.org/10.1007/s10664-017-9577-2

Overfitting in semantics-based automated program repair

Xuan Bach D. Le[1] · **Ferdian Thung**[1] · **David Lo**[1] · **Claire Le Goues**[2]

Abstract The primary goal of Automated Program Repair (APR) is to automatically fix buggy software, to reduce the manual bug-fix burden that presently rests on human developers. Existing APR techniques can be generally divided into two families: semantics- vs. heuristics-based. Semantics-based APR uses symbolic execution and test suites to extract semantic constraints, and uses program synthesis to synthesize repairs that satisfy the extracted constraints. Heuristic-based APR generates large populations of repair candidates via source manipulation, and searches for the best among them. Both families largely rely on a primary assumption that a program is correctly patched if the generated patch leads the program to pass all provided test cases. Patch correctness is thus an especially pressing concern. A repair technique may generate *overfitting* patches, which lead a program to pass all existing test cases, but fails to generalize beyond them. In this work, we revisit the overfitting problem with a focus on semantics-based APR techniques, complementing previous studies of the overfitting problem in heuristics-based APR. We perform our study using IntroClass and Codeflaws benchmarks, two datasets well-suited for assessing repair quality, to

Communicated by: Martin Monperrus and Westley Weimer

✉ Xuan Bach D. Le
dxb.le.2013@phdis.smu.edu.sg

Ferdian Thung
ferdiant.2013@phdis.smu.edu.sg

David Lo
davidlo@smu.edu.sg

Claire Le Goues
clegoues@cs.cmu.edu

[1] School of Information Systems, Singapore Management University, Singapore, Singapore

[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Published online: 02 March 2018

systematically characterize and understand the nature of overfitting in semantics-based APR. We find that similar to heuristics-based APR, overfitting also occurs in semantics-based APR in various different ways.

Keywords Automated program repair · Program synthesis · Symbolic execution · Patch overfitting

1 Introduction

Automated program repair (APR) addresses an important challenge in software engineering. Its primary goal is to repair buggy software to reduce the human labor required to manually fix bugs (Tassey 2002). Recent advances in APR have brought this once-futuristic idea closer to reality, repairing many real-world software bugs (Mechtaev et al. 2016; Le Goues et al. 2012; Long and Rinard 2016b; Kim et al. 2013; Xuan et al. 2016; Le et al. 2015b, 2016a, 2017b). Such techniques can be broadly classified into two families, semantics-based vs. heuristic, differentiated by the underlying approach, and with commensurate strengths and weaknesses. Semantics-based APR typically leverages symbolic execution and test suites to extract semantic constraints, or *specifications*, for the behavior under repair. It then uses program synthesis to generate repairs that satisfy those extracted specifications. Early semantics-based APR techniques used template-based synthesis (Könighofer and Bloem 2011, 2012). Subsequent approaches use a customized component-based synthesis (Nguyen et al. 2013; DeMarco et al. 2014), which has since been scaled to large systems (Mechtaev et al. 2016). By contrast, heuristic APR generates populations of possible repair candidates by heuristically modifying program Abstract Syntax Trees (AST)s, often using optimization strategies like genetic programming or other heuristics to construct good patches (Weimer et al. 2010, 2013; Le Goues et al. 2012; Le et al. 2016c; Qi et al. 2014; Long and Rinard 2016b).

Both heuristic and semantics-based APR techniques have been demonstrated to scale to real-world programs. However, the quality of patches generated by these is not always assured. Techniques in both families share a common underlying assumption that generated patches are considered correct if they lead the program under repair pass all provided test cases. This raises a pressing concern about true correctness: an automatically-generated repair may not generalize beyond the test cases used to construct it. That is, it may be *plausible* but not fully *correct* (Qi et al. 2015). This problem has been described as *overfitting* (Smith et al. 2015) to the provided test suites. This is an especial concern given that test suites are known to be incomplete in practice (Tassey 2002). As yet, there is no way to know a priori whether and to what degree a produced patch overfits. However, the degree to which a technique produces patches that overfit has been used post facto to characterize the limitations and tendencies of heuristic techniques (Smith et al. 2015), and to experimentally compare the quality of patches produced by novel APR methods (Ke et al. 2015).

There is no reason to believe that semantics-based APR is immune to this problem. Semantics-based approaches extract behavioral specifications from the same partial test suites that guide heuristic approaches, and thus the resulting specifications that guide repair synthesis are themselves also partial. However, although recent work has assessed proxy measures of patch quality (like functionality deletion) (Mechtaev et al. 2016), to the best of our knowledge, there exists no comprehensive, empirical characterization of the overfitting problem for semantics-based APR in the literature.

We address this gap. In this article, we comprehensively study overfitting in semantics-based APR. We perform our study on Angelix, a recent state-of-the-art semantics-based APR tool (Mechtaev et al. 2016), as well as a number of syntax-guided synthesis techniques used for program repair (Le et al. 2016b). We evaluate the techniques on a subset of the IntroClass (Le Goues et al. 2015) and Codeflaws benchmarks (Tan et al. 2017), two datasets well-suited for assessing repair quality in APR research. Both consist of many small defective programs, each of which is associated with two independent test suites. The multiple test suites renders these benchmarks uniquely beneficial for assessing patch overfitting in APR. One test suite can be used to guide the repair, and the other is used to assess the degree to which the produced repair generalizes. This allows for controlled experimentation relating various test suite and program properties to repairability and generated patch question.

In particular, IntroClass consists of student-written submissions for introductory programming assignments in the C programming language. Each assignment is associated with two independent, high-quality test suites: a *black-box* test suite generated by the course instructor, and a *white-box* test suite generated by automated test case generation tool KLEE (Cadar et al. 2008) that achieves full branch coverage over a known-good solution. IntroClass has been previously used to characterize overfitting in heuristic repair (Smith et al. 2015). The Codeflaws benchmark consists of programs from the Codeforces[1] programming contest. Each program is also accompanied by two set of test suites: one for the programmers/users to validate their implementations, and the other for the contest committee to validate the users' implementations.

Overall, we show that overfitting does indeed occur with semantics-based techniques. We characterize the relationship between various factors of interest, such as test suite coverage and provenance, and resulting patch quality. We observe certain relationships that appear consistent with results observed for heuristic techniques, as well as results that stand counter to those achieved on them. These results complement the existing literature on overfitting in heuristic APR, completing the picture on overfitting in APR in general. This is especially important to help future researchers of semantics-based APR to overcome the limitations of test suite guidance. We argue especially (with evidence) that semantics-based program repair should seek stronger or alternative program synthesis techniques to help mitigate overfitting.

Our contributions are as follows:

- We perform the first study on overfitting in semantics-based program repair. We show that semantics-based APR can generate high-quality repairs, but can also produce patches that overfit.
- We assess relationships between test suite size and provenance, number of failing tests, and semantics-specific tool settings and overfitting. We find, in some cases, results consistent with those found for heuristic approaches. In other cases, we find results that are interestingly inconsistent.
- We substantiate that using multiple synthesis engines could be one possible approach to increase the effectiveness of semantics-based APR, e.g., generate correct patches for a larger number of bugs. This extends Early Results findings from Le et al. (2016b).
- We present several examples of overfitting patches produced by semantics-based APR techniques, with implications and observations for how to improve them. For example, we observe that one possible source for overfitting in semantics-based APR could be

[1] http://codeforces.com/

due to the “conservativeness” of the underlying synthesis engine, that returns the first solution found (without consideration of alternatives).

The remainder of this article proceeds as follows. Section 2 describes background on semantics-based program repair. Section 3.1 explains the data we use in our experiments; the remainder of Section 3 presents experimental results, and insights behind them. We discuss threats to validity in Section 4, and related work in Section 5. We conclude and summarize in Section 6.

2 Semantics-Based APR

We focus on understanding and characterizing overfitting behavior in semantics-based automated program repair (APR). Semantics-based APR has recently been shown by Mechtaev et al. (2016) to scale to the large programs previously targeted by heuristic APR techniques (Le Goues et al. 2012; Long and Rinard 2016b). This advance is instantiated in Angelix, the most recent, state-of-the-art semantics-based APR approach in the literature.

Angelix follows a now-standard model for test-case-driven APR, taking as input a program and a set of test cases, at least one of which is failing. The goal is to produce a small set of changes to the input program that corrects the failing test case while preserving the other correct behavior. At a high level, the technique identifies possibly-defective expressions, extracts value-based specifications of correct behavior for those expressions from test case executions, and uses those extracted specifications to synthesize new, ideally corrected expressions. More specifically, Angelix first uses existing fault localization approaches, like Ochiai (Abreu et al. 2007) to identify likely-buggy expressions. It then uses a selective symbolic execution procedure in conjunction with provided test suites to infer correctness constraints, i.e., *specifications*.

We now provide detailed background on Angelix’s mechanics. We first detail the two core components of Angelix: specification inference (Section 2.1) and program synthesis (Section 2.2). We explain various tunable options that Angelix provides to deal with different classes of bugs (Section 2.3). We then provide background on the variants of semantics-based APR we also investigate our experiments: SemFix (Section 2.4), and Syntax-Guided Synthesis (SyGuS) as applied to semantics-based APR (Section 2.5).

2.1 Specification Inference via Selective Symbolic Execution

Angelix relies on the fact that many defects can be repaired with only a few edits (Martinez and Monperrus 2015), and thus focuses on modifying a small number of likely-buggy expressions for any particular bug. Given a small number of potentially-buggy expressions identified by a fault localization procedure, Angelix performs a selective symbolic execution by installing symbolic variables α_i at each chosen expression i.[2] It concretely executes the program on a test case to the point that the symbolic variables begin to influence execution, and then switches to symbolic execution to collect constraints over α_i. The goal is to infer constraints that describe solutions for those expressions that could lead all test cases to pass.

[2] Angelix can target multiple expressions at once; we explain the process with respect to a single buggy expression for clarity, but the technique generalizes naturally.

These value-based specifications take the form of a *precondition* on the values of variables before a buggy expression is executed, and then a *postcondition* on the values of α_i. The precondition is extracted using forward analysis on the test inputs to the point of the chosen buggy expression; The postcondition is extracted via backward analysis from the desired test output by solving the model: $PC \wedge O_a == O_e$. PC denotes the path condition collected via symbolic execution, O_a denotes the actual execution output, and O_e denotes the expected output. The problem of program repair now reduces to a synthesis problem: Given a precondition, Angelix seeks to synthesize an expression that satisfies the postcondition (described in Section 2.2)

Angelix infers specifications for a buggy location using a given number of test cases, and validates synthesized expressions with respect to the entire test suite. Angelix chooses the initial test set for the specification inference based on coverage, selecting tests that provide the highest coverage over the suspicious expressions under consideration. If any tests fail over the course of validation process, the failing test is incrementally added to the test set used to infer specifications for subsequent repair efforts, and the inference process moves to the next potentially-buggy location. This process is repeated until a repair that leads the program to pass all tests is found. We further discuss the number of tests used for specification inference in Section 2.3.

2.2 Generating Repairs via Synthesis and Partial MaxSMT Solving

Angelix adapts component-based repair synthesis (Jha et al. 2010) to synthesize a repair conforming to the value-based specifications extracted by the specification inference step. It solves the synthesis constraints with Partial Maximum Satisfiability Modulo Theories (Partial MaxSMT) (Mechtaev et al. 2015) to heuristically ensure that the generated repair is minimally different from the original program.

Component-Based Synthesis The synthesis problem is to arrange and connect a given set of *components* into an expression that satisfies the provided constraints over inputs and outputs. We illustrate via example: Assume the available components are variables x and y, and binary operator “−” (subtraction). Further assume input constraints of $x == 1$ and $y == 2$, and an output constraint of $f(x, y) == 1$. $f(x, y)$ is the function over x and y to be synthesized. The component-based synthesis problem is to arrange x, y, and “−” (the components) such that the output constraint is satisfied with respect to the input constraints. For our example, one such solution for $f(x, y)$ is $y - x$; Another is simply x, noting that the synthesized expression need not include all available components. The synthesis approach encodes the constraints and available components in such a way that, if available, a satisfying SMT model is trivially translatable into a synthesized expression, that synthesized expression is well-formed, and it provably satisfies the input-output constraints

Partial MaxSMT for Minimal Repair Angelix seeks to produce repairs that are small with respect to the original buggy expressions. Finding a minimal repair can be cast as an optimization problem, which Angelix addresses by leveraging Partial MaxSMT (Mechtaev et al. 2015). Partial MaxSMT can solve a set of *hard* clauses, which *must* be satisfied, along with as many *soft* clauses as possible. In this domain, the hard clauses encode the input-output and program well-formedness constraints, and the soft clauses encode structural constraints that maximally preserve the structure of the original expressions. Consider the two possible solutions to our running example: $f(x, y) = y - x$, or $f(x, y) = x$. If the original buggy expression is $x - y$, synthesis using Partial MaxSMT might produce $f(x, y)$

= $y - x$ as a desired solution, because it maximally preserves the structure of the original expression by maintaining the "−" operator.

2.3 Tunable Parameters in Angelix

We investigate several of Angelix's tunable parameters in our experiments. We describe defaults here, and relevant variances in Section 3.

Suspicious Location Group Size Angelix divides multiple suspicious locations into groups, each consisting of one or more locations. Angelix generates a repaired expression for each potentially-buggy expression in a group. During specification inference, Angelix installs symbolic variables for locations in each group, supporting inference and repair synthesis on multiple locations. Given a group size N, Angelix can generate repairs that touch no more than N locations. For example, if $N = 2$ (the default setting), Angelix can generate a repair that modifies either one or two expressions. Angelix groups buggy expressions by either suspiciousness score, or proximity/location. By default, Angelix groups by location.

Number of Tests used for Specification Inference The number of tests used to infer (value-based) specifications is important for performance and generated patch quality. Too many tests may overwhelm the inference and synthesis engines; too few may lead to the inference of weak or inadequate specifications expressed in terms of input-output examples, which may subsequently render the synthesis engine to generate poor solutions that do not generalize. As described above, Angelix increases the size of the test suite incrementally as needed. By default, two tests are used to start, at least one of which must be failing.

Synthesis Level The selection of which components to use as ingredient components for synthesis is critical. Too few components overconstrains the search and reduces Angelix's expressive power; too many can overwhelm the synthesis engine by producing an overly large search space. Angelix tackles this problem by defining *synthesis levels*, where each level includes a particular set of permitted ingredient components. For a given repair problem, the synthesis engine searches for solutions at each synthesis level, starting with the most restrictive and increasing the size of the search space with additional components until either a repair is found or the search is exhausted. By default, Angelix's synthesis levels include *alternatives, integer-constants,* and *boolean-constants* levels. The *alternatives* synthesis level allows Angelix's synthesis engine to use additional components similar to existing code, e.g., "$\leq$" is an alternative component for the component "$<$". The *integer-constants* and *boolean-constants* levels enable additional integer and boolean constants available to the synthesis engine, respectively.

Defect Classes Angelix can handle four classes of bugs, related to, respectively, *assignments*, *if-conditions*, *loop-conditions*, and *guards*. The "assignments" defect class considers defective right-hand-sides in assignments. "if-conditions" and "loop-conditions" considers buggy expressions in conditional statements. The "guards" defect class considers the addition of synthesized guards around buggy statements. For example, Angelix might synthesize a guard $if(x > 0)$ to surround a buggy statement $x = y + 1$, producing $if(x > 0)$ $\{x = y + 1\}$. The more defect classes considered, the more complicated the search space, especially given the "guard" class (which can manipulate arbitrary statements). By default, Angelix considers assignments, if-conditions, and loop-conditions.

2.4 SemFix: Program Repair via Semantic Analysis

SemFix, a predecessor of Angelix, is a synergy of fault localization, symbolic execution, and program synthesis. The primary differences between SemFix and Angelix are: (1) SemFix's specification inference engine works on only a single buggy location (Angelix can operate over multiple buggy locations at once), (2) SemFix defines the specification as a disjunction of inferred path conditions. Angelix instead extracts sequences of angelic values that allow the set of tests to pass from each path, and uses them to construct a so-called "angelic forest." As a result, the size of Angelix specification is independent of the size of the program (depending only on the number of symbolic variables). This makes Angelix more scalable than SemFix, (3) SemFix's synthesis engine only synthesizes repairs for a single buggy location (Angelix can synthesize multi-expression repairs), and (4) SemFix does not attempt to minimize the syntactic distance between a solution and the original buggy expression using Partial MaxSMT. These differences are particularly important for scalability (Angelix can repair bugs in larger programs than can SemFix), and patch quality, which this article explores in detail.

2.5 Syntax-Guided Synthesis for Semantics-Based Program Repair

Other synthesis approaches are also applicable to semantics-based program repair, with possible implications for repair performance (Le et al. 2016b). We systematically evaluate these implications for repair quality, and thus now describe the Syntax-Guided Synthesis (SyGuS) (Alur et al. 2015) techniques we use in our experiments.

Given a specification of desired behavior, a SyGuS engine uses a restricted grammar to describe (and thus constrain) the syntactic space of possible implementations. Different SyGuS engines vary in the search strategies used to generate solutions that satisfy the specification and conform to the grammar. We investigate two such techniques:

- The *Enumerative* strategy (Alur et al. 2015) generates candidate expressions in increasing size, and leverages specifications and a Satisfiability Modulo Theory (SMT) solver to prune the search space of possible candidates. Since repeatedly querying an SMT solver regarding the validity of a solution with respect to a specification (the *validity query*) is expensive, it uses counter-examples to improve performance. That is, whenever a solution failed to meet the specification, a counter-example is generated and added to the next validity query.
- *CVC4* is the first SyGus synthesizer (Reynolds et al. 2015) implemented inside an SMT solver, via a slight modification of the solver's background theory. To synthesize an implementation that satisfies all possible inputs, it translates the challenging problem of solving universal quantifier over all inputs into showing the unsatisfiability of the negation of the given specification. It synthesizes a solution based on the unsatisfiability proof.

Recent SyGuS competitions suggest that the CVC4 and enumerative engines are the among the best, evaluated on SyGuS-specialized benchmarks.[3]

We follow the approach described in previous work (Le et al. 2016b) to integrate the *Enumerative* and *CVC4* synthesizers into Angelix. At a high level, Angelix infers value-based specifications as usual, and we automatically translate those specifications into a

[3] http://www.sygus.org/

suitable SyGuS format, with optimizations to constrain the repair minimality. Different SyGuS engines can then be run on the same generated SyGuS script to synthesize a repair conforming to the inferred specifications, allowing for a controlled comparison of different synthesis approaches in the context of a semantics-based repair technique.

3 Empirical Evaluation

The primary purpose of our experiments is to systematically investigate and characterize overfitting in semantics-based APR. To this end, we use benchmarks that provide many buggy programs along with two independent test suites. For each run of each repair technique on a given buggy program, we use one set of provided test cases (the *training* tests) to generate a repair, and the other (the *held-out* tests) to assess the quality of the generated repair. If a repair does not pass the held-out tests, we say it is an *overfitting* repair that is not fully general; this is a proxy measure for repair quality (or lack thereof). Otherwise, we call it a *non-overfitting* or *general* repair.[4]

We describe our experimental dataset in Section 3.1. We then begin by assessing baseline patching and overfitting behavior generally (Section 3.2). We then evaluate relationships between overfitting and characteristics of input test suites and input programs (Section 3.3), as well as tunable tool parameters (Section 3.4). Finally, we present and discuss several informative test cases from the considered dataset (Section 3.5) and a qualitative case study on real-world bugs (Section 3.6).

3.1 Experimental Data

We obtained Angelix from https://github.com/mechtaev/angelix/, using the version evaluated in Mechtaev et al. (2016). We set all tunable parameters to their defaults (Section 2.3) unless otherwise noted.

We conduct the majority of our experiments on buggy programs from a subset of the IntroClass benchmark (Le Goues et al. 2015), and the Codeflaws benchmark (Tan et al. 2017).[5] Both benchmarks consist of small programs, but are particularly suitable benchmark for assessing repair quality via overfitting, because they each provide two test suites for each buggy program. One set can be used to guide the repair, while the second set is used to assess the degree to which it generalizes.

IntroClass IntroClass consists of several hundred buggy versions of six different programs, written by students as homework assignments in a freshmen programming class. Each assignment is associated with two independent high-coverage test suites: a black-box test suite written by the course instructor, and a white-box test suite generated by the automated test generation tool KLEE (Cadar et al. 2008) on a reference solution.

We filtered IntroClass to retain only textually unique programs. We then further filter to retain those programs with outputs of type boolean, integer, or character because Angelix's inference engine does not fully support output of other types such as String or float due to the limited capability of constraint solving technique used in Angelix's underlying symbolic

[4] We use the words "repair" and "patch" interchangeably.

[5] We discuss the real-world bugs we describe qualitatively in Section 3.6.

Table 1 Baseline repair results on IntroClass (top) and Codeflaws (bottom)

(a) Baseline repair results on IntroClass. Total benchmark program versions considered (Total), baseline repair results for programs that fail at least one black-box test (Black-box, center columns), and those that fail at least one white-box test (White-box, last columns). The sets of programs that fail at least one test from each set are not disjoint

		Black-box bugs					White-box bugs				
Subject	Total	#	Angelix	CVC4	Enum	SemFix	#	Angelix	CVC4	Enum	SemFix
smallest	67	56	37	39	29	45	41	37	37	36	37
median	61	54	38	28	27	44	45	35	36	23	38
digits	108	57	6	4	3	10	90	5	2	2	8
syllables	48	39	0	0	0	0	42	0	0	0	0
checksum	31	19	0	0	0	0	31	0	0	0	0
total	315	225	81	71	59	99	249	77	75	61	83

(b) Baseline repair results on Codeflaws. The tests available to the users serve as training tests; the contest committee tests serve as held-out tests

Subject	Total	Angelix	CVC4	Enum	SemFix
CodeFlaws	651	81	91	92	56

execution engine. This leaves us with 315 program versions in the dataset, shown in column "Total" in Table 1a (grouped by assignment type).

Codeflaws The Codeflaws benchmark contains 3,902 defects collected from the Codeforces programming contest,[6] categorized by bug types (Tan et al. 2017). Since running all bugs is computationally expensive, we select for our experiments 665 bugs belonging to different bug categories. The selected bugs are from the "replace relational operator" bug category (*ORRN*), and the "replace logical operator" (*OLLN*) and "tighten or loosen condition" (*OILN*) categories. Examples of the selected defect types are shown in Fig. 1. These three selected defect categories are best suited to repair via semantics-based techniques (note that the majority of bugs fixed by Angelix in Mechtaev et al. (2016) belongs to the "if-condition" defect type).

Similar to IntroClass, each program in Codeflaws is accompanied by two test suites: one suite is available to the contest's users to assess their implementation, and the other is *only* available to the contest's committee to assess the implementations submitted by users.

3.2 Baseline Patching and Overfitting

Our first three research questions (1) establish baseline patch generation results, (2) evaluate whether there exists an apparent relationship between the number of tests a program fails and repair success, and (3) assess the degree to which the semantics-based techniques under consideration produce patches that overfit to a training test suite.

[6] http://codeforces.com/

```
// An example of ORRN defect type
-if(a > b)
+if(a >= b)

// An example of OLLN defect type
-if(a || b)
+if(a && b)

// An example of OILN defect type
-if(a)
+if(a || b)
```

Fig. 1 Examples of defect types from the Codeflaws dataset used in our experiments

Research Question 1 How often do Angelix (including various synthesis engines) and SemFix generate patches that lead the buggy programs to pass all training test cases?

IntroClass In these initial experiments, we use the black-box tests as training tests, and the white-box tests as held-out tests. Of the 315 program versions, 225 programs that have at least one failing black-box test case. The center portion of Table 1a shows results (we discuss white-box results in Research Question 6), in terms of the number of patches generated by Angelix, CVC4, Enumerative, and SemFix using black-box tests for training. In total, Angelix generates patches for 81 out of 225 versions (36%). Note that Angelix generated no patches for the *syllables* and *checksum* programs; our manual investigation suggests that this is primarily due to imprecision in the built-in fault localization module. Beyond this, success rate varies by assignment type. Angelix has the most patch generation success (70.4%) for programs written for the *median* problem. The overall results indicate that Angelix generates patches frequently enough for us to proceed to subsequent research questions.

Angelix incorporating the CVC4 and Enumerative SyGuS engines generated patches for 71 and 59 versions, respectively, a lower patch generation success rate comparatively (31.6% for CVC4, and 26.2% for Enumerative). SemFix, on the other hand, generates patches for 99 versions with slightly higher patch generation rate (44%). Despite the lower patch generation rates, CVC4 and Enumerative do generate patches for programs for which Angelix cannot. This raises an interesting question regarding whether it might it be beneficial to use multiple synthesis techniques to increase the effectiveness of semantics-based APR. In subsequent research questions, we investigate whether Angelix, CVC4, Enumerative, and SemFix do indeed generate non-overfitting patches for distinct program versions.

Codeflaws For Codeflaws, we use the tests available to users as training tests, and the tests that are only available to the contest's committee as held-out tests. Table 1b shows results. Angelix, CVC4, Enumerative and SemFix succeed in generating patches for 12.5%, 14%, 14%, and 9% of the buggy programs, respectively. Although this is a much lower patch generation rate as compared to the IntroClass results, the number of generated patches is adequate to allow us to proceed to subsequent research questions.

Research Question 2 Is the number of tests that a buggy program fails related to patch generation success?

Our goal in this research question is to see whether and to what degree repairability appears associated with the number of tests a program is failing. To answer this question, we group programs that fail the same test case proportion, and then calculate the patch generation success rate per group. We aggregate across both benchmarks in these results (we use all defects from Codeflaws). We allow Angelix to generate more patches by using `--generate-all` parameter for all synthesis engines. Our *null hypothesis* ($\mathcal{H}_0$) is that there is no linear correlation between the number of tests a program fails and patch generation success; the alternate hypothesis ($\mathcal{H}_1$) is that there is such a relationship.

Results Figure 2 shows a scatter plot relating test suite failure rate and patch generation rate. Our statistical test shows that the correlation of linear regression between these two rates is not statistically significant ($p > 0.05$). Thus, we cannot reject the null hypothesis. However, we can observe interesting trends, as the the scatter plot suggests that there *may* be a general downtrend between test suite failure rate and patch generation rate across all configurations (after discounting outliers). That is, as test suite failure rate increases, the patch generation rate decreases. This is consistent with prior results for heuristic repair, and matches the intuition that producing a fully satisfactory patch may be more difficult the more test cases are initially failing. In the extreme cases in which all tests fail, no patches were generated. We manually investigated several such programs, and found that such bugs in these datasets are usually caused by typos in a constant string, which none of the synthesis techniques can modify.

Research Question 3 How often do the produced patches overfit to the training tests, when evaluated against the held-out tests?

In this question, we evaluate whether the generated patches generalize, indicating that they are more likely to be correct with respect to the program specification. An ideal program

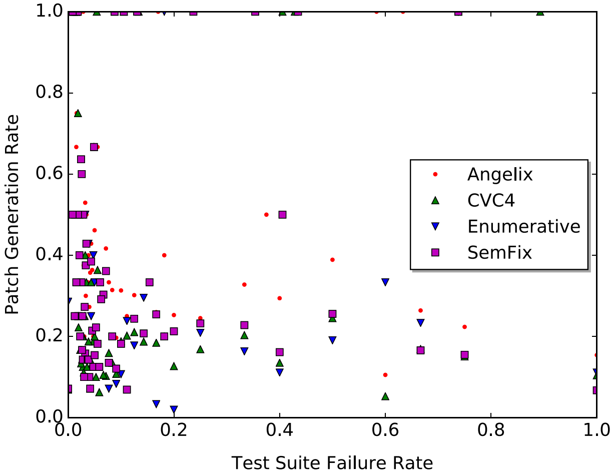


Fig. 2 Patch generation rate (i.e., producing at least one patch that passes all tests in a test suite) vs. test suite failure rate

repair technique would often generate general patches, and produce overfitting patches infrequently. We test all patches produced for Research Question 1 against the held-out to measure rate.

Results Table 2a and b show the number of patches produced for each subject program that fail at least one held-out test for the IntroClass and Codeflaws datasets, respectively. On IntroClass, 61 of the 81 (75%) Angelix-produced patches overfit to the training tests, while 80%, 81%, and 90% of the CVC4-, Enumerative-, and SemFix-produced patches do, respectively. On Codeflaws, 44 of the 81 (54%) Angelix-produced patches overfit, while 83.5%, 87%, and 68% of patches generated by CVC4, Enumerative, and SemFix do, respectively. This suggests that, although semantics-based repair has been shown to produce high-quality repairs on a number of subjects, overfitting to the training tests is still a concern. We present case studies to help characterize the nature of overfitting in semantics-based APR in Section 3.5.

One possible reason that CVC4 and Enumerative underperform Angelix's default synthesis engine is that the SyGuS techniques do not take into account the original buggy expressions. We observed that the resulting patches can be very different from the originals they replace, which can impact performance arbitrarily. However, the CVC4 and Enumerative techniques do generate non-overfitting patches for programs that default Angelix cannot produce non-overfitting patches, as shown in Fig. 3a and b. Similarly, SemFix, CVC4, and Enumerative also have non-overlapping non-overfitting patches (results omitted). This phenomenon also happens between Angelix and SemFix. This suggests that using multiple synthesis engines to complement one another may increase the effectiveness of semantics-based APR.

3.3 Training Test Suite Features

Our next three research questions look at the relationship between features of the training test suite and produced patch quality, looking specifically at (4) test suite size, (5) number of failing tests, and (6) test suite provenance.

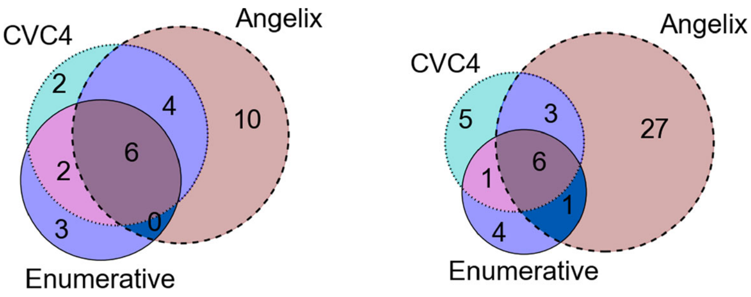
Table 2 Baseline overfitting results on IntroClass (top) and Codeflaws (bottom). In both tables, A/B denotes A overfitting patches out of B total patches generated

(a) IntroClass overfitting rates for each APR approach, using black box (center columns) and white box (right-most columns) as training tests. We omit syllables and checksum, for which no patches were generated

	Black box				White box			
Subject	Angelix	CVC4	Enum	SemFix	Angelix	CVC4	Enum	SemFix
smallest	27/37	33/39	24/29	36/45	31/37	33/37	33/36	33/37
median	29/38	21/ 28	21/27	40/44	25/35	36/36	23/23	38/38
digits	5/6	3/4	3/3	10/10	0/5	2/2	2/2	2/8

(b) Codeflaws overfitting rates for each APR approach

Subject	Total	Angelix	CVC4	Enum	SemFix
Codeflaws	651	44/81	76/91	80/92	38/56



(a) Non-overfitting patches generated by Angelix, CVC4, and Enumerative on IntroClass.

(b) Non-overfitting patches generated by Angelix, CVC4, and Enumerative on Codeflaws.

Fig. 3 Non-overfitting patches by Angelix, CVC4, and Enumerative on IntroClass and Codeflaws benchmarks

Research Question 4 Is the training test suite's *size* related to patch overfitting?

To answer this question, we vary the training test suite size and observe the resulting overfitting rate. To achieve this, we randomly sample the black-box test suite (for the IntroClass dataset) and user's test suite (for the Codeflaws dataset) to obtain 25%, 50% and 75% of the suite as training tests, and use the resulting tests to guide repair. We vary the number of training tests, but keep the pass-fail ratio of tests in each version consistent. We repeat this experiment five times and aggregate the results for each repair technique.

Results Table 3a and b show results on the IntroClass and Codeflaws benchmarks, respectively. Interestingly, the overfitting rate fluctuation is very small. Table 3a, shows that on IntroClass, using 25%, 50%, and 75% of black-box tests as training tests, Angelix has an overfitting rate of 84%, 94%, and 78%, respectively. This highlights an interesting trend: When training suite size increases, Angelix appears to generate fewer patches, but without a major change in overfitting rate. For example, considering *smallest* programs, Angelix generates 29, 25, and 20 non-overfitting patches when 25%, 50%, and 75% of black-box tests are used, respectively. We conclude that that it may be slightly more difficult to generate patches in response to higher-coverage test suites. However, as test suite coverage increases,

Table 3 Overfitting rate of Angelix, CVC4, Enumerative, and SemFix when varying the number of tests used for training on IntroClass (top) and Codeflaws (bottom)

	Angelix			CVC4			Enum			SemFix		
Subject	25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
	(a) Overfitting by number of tests used, IntroClass											
Smallest	29/38	25/34	20/29	28/36	22/32	17/27	26/35	23/33	19/28	32/41	36/45	36/45
Median	35/40	29/34	25/30	33/39	26/32	26/30	33/38	27/32	25/30	36/43	40/45	41/46
Digits	8/8	6/6	7/7	8/8	6/6	6/6	8/8	6/6	6/6	9/9	12/12	12/12
	(b) Overfitting by number of tests used, Codeflaws											
	87/99	78/95	58/73	86/90	85/94	104/111	73/78	77/88	112/120	89/97	77/85	57/64

overfitting rate does not appear to substantially decrease. Similar trends appear to apply to CVC4, Enumerative, and SemFix.

Table 3b shows the results on the Codeflaws benchmark. We can see that Angelix and SemFix follow the same trend as described above on the results on the IntroClass dataset. CVC4 and Enum, however, depict an opposite trend in terms of patch generation rate, wherein the number of generated patches increases with training test suite size.

These results are particularly interesting when contrasted with prior results characterizing overfitting for heuristic repair (Smith et al. 2015). Smith et al. (2015) found that lower-coverage test suites posed a risk for heuristic repair, leading to patches that were less likely to generalize. By contrast, our results for semantics-based repair do not show this relationship; test suite coverage overall may not influence the quality of semantics-based patches to the same degree they do in heuristic techniques. As a result, semantics-based approaches may be safer to use than heuristic techniques when only lower-coverage or lower-quality test suites are available.

Note that these semantics-based APR techniques generate repairs eagerly. That is, they generate one plausible repair at a time, and if that repair leads the program to pass all tests, it is returned without considering other candidates. Since there can exist many plausible patches that pass all tests, but are not necessarily correct (this has been empirically characterized for heuristic techniques (Long and Rinard 2016a)), a potentially fruitful future direction for semantics-based APR may be to lazily generate a number of candidates using the synthesis strategy, and then employ an appropriate ranking function to heuristically rank candidates according to predicted correctness, combining various elements of both heuristic and semantics-based approaches.

Research Question 5 Is the number of tests a buggy program fails related to patch overfitting?

As in question 2, we group buggy programs by test suite failure rate and calculate patch overfitting rate (patches that overfit divided by patches generated) per group. We aggregate across both benchmarks in these results (we use all defects from Codeflaws). We allow Angelix to generate more patches by using `--generate-all` parameter. Our *null hypothesis* ($\mathcal{H}_0$) is that there is no linear correlation between the number of tests that a buggy program fails and patch overfitting; our alternate hypothesis ($\mathcal{H}_1$) is that there is such a relationship.

Results Figure 4 shows a scatter plot relating test suite failure rate and overfitting rate. The correlation coefficient of the linear regression between these two rates is not statistically significant ($p > 0.05$). Thus, we cannot reject the null hypothesis. However, the visual trend is instructive, suggesting potentially similar trends across all synthesis configurations: there may be an downtrend between test suite failure rate and overfitting rate (after discounting outliers). That is, the more tests that fail on the original buggy program), the higher the overfitting rate. Since the amount of data that we have do not give us a statistical significant result, we cannot draw strong conclusions; we leave the addition of such data to future work.

Research Question 6 How is the training test suite's *provenance* (automatically generated vs. human-written) related to patch overfitting?

Automatic test generation may provide a mechanism for augmenting inadequate test suites for the purposes of program repair. However, previous work assessing overfitting for heuristic repair found that patch quality varied based on the origin (or *provenance*) of the

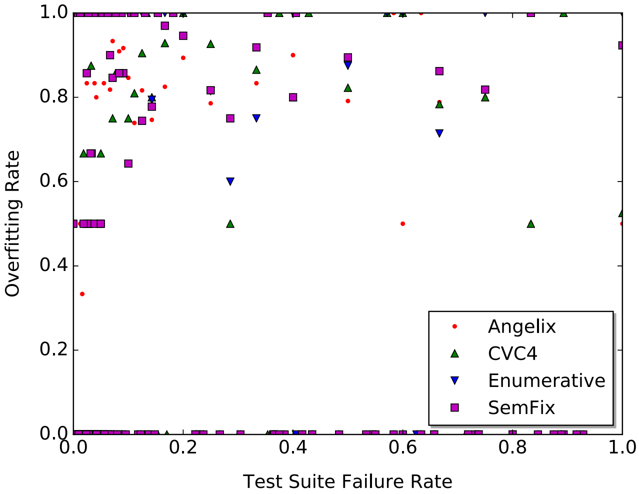


Fig. 4 Overfitting rate (i.e., number of overfit patches over generated patches) vs. the test suite failure rate

training test suite on the IntroClass dataset. That is, the human and instructor-provided black box tests led APR techniques to produce higher-quality repairs, i.e., the ones that pass more held-out tests, than the automatically generated tests (generated by KLEE) (Smith et al. 2015). We assess the same concern for semantic-based APR by comparing the quality of patches generated using the white-box (KLEE-generated) tests to those of the black-box (human-generated) tests from the IntroClass dataset. We only use IntroClass for this question since its held-out tests are automatically generated; the provenance of the held-out tests in Codeflaws is unspecified (Tan et al. 2017).

Results The right-hand-side of Table 1a shows baseline patch results using the white-box tests for training; the right-hand side of Table 2a shows how many of those patches overfit. Angelix generates patches for 77 buggy programs using these test suites, including 37, 35, and 5 versions for subjects *smallest*, *median*, and *digits*, respectively. Of those, 31, 25, and 5 patches fail to generalize, respectively. Overall, when using white-box tests as training tests, Angelix generates patches with an overfitting rate of 72.7% on average. This is very slightly lower as compared to the rate for the black-box tests, seen in Research Question 3 (75% versus 72.7%).

This is particularly interesting as compared to the results from the heuristic case, suggesting that automated test generation could be particularly useful in helping Angelix to mitigating the risk of overfitting. As above, lower-quality test suites may pose a smaller risk to the output quality of this technique type.

By contrast, the performance of CVC4 and Enumerative suffers when using the white-box as compared to the black-box tests. CVC4 and Enumerative can only generate non-overfitting patches for 4 and 3 versions of *smallest*, respectively. This indicates a very high overfitting rate (around 95%). The performance of SemFix is almost the same when either using black-box or white-box tests as training tests (overfitting rate of around 87%). Thus, there remains a need to improve automated test generations before it can be used across the board for automatic repair, and to understand the source of this quality discrepancy.

Table 4 Overfitting rate when using all training tests for specification inference for IntroClass (top; we omit syllables and checksum, for which no patches were generated) and Codeflaws (bottom)

Subject	Angelix	CVC4	Enum	SemFix
smallest	10/20	27/39	27/39	12/19
median	18/22	13/ 20	24/32	18/23
digits	5/5	3/3	3/3	5/5
Codeflaws	102/126	102/108	79/84	100/112

3.4 Tunable Technique Parameters

Our next two research questions concern the relationship between patch generation success and quality and (7) number of tests used for specification inference and (8) the Angelix group size feature.

Research Question 7 What is the relationship between the number of tests used for specification inference and patch generation success and patch quality?

Theoretically, the more tests used for specification inference, the more comprehensive the inferred specifications, which may help synthesis avoid spurious solutions.[7] Thus, we investigate the relationship between the number of considered tests and patch generation and quality for all considered techniques.

IntroClass Results We use black-box tests as training tests, and white-box tests as held-out tests to answer this question, and instruct the inference engine to use *all* available tests for specification inference. The top of Table 4 shows results. Angelix generates 47 patches, of which 33 do not fully generalize, indicating an overfitting rate of 70.2% on average. As compared to the results from Research Question 3, in which we use Angelix's default setting (starting with two tests), the overfitting rate is slightly reduced (from 75 to 70.2%).

CVC4 and Enumerative generate patches with overfitting rate of 69.4% (43 incorrect patches over 62 generated patches), and 73% (54 incorrect patches over 74 generated patches), on average, respectively. The effect on overfitting rate is more dramatic for these approaches. CVC4's overfitting rate decreases, from 80.3 to 69.4%. Similarly, SemFix's overfitting rate decreases from 90 to 74%: it generates 35 incorrect patches over 47 generated patches. Overall, these results suggest that using more tests for specification inference helps semantics-based program repair to mitigate overfitting, supporting our hypothesis.

Codeflaws Results We use tests that are available to users as training tests, and tests that are available only to the contest's committee as held-out tests. The last row of Table 4 shows results. Angelix generates 126 patches, of which 102 do not generalize, indicating an overfitting rate of 81%. Compared to the results shown in Table 2b in research question 3, which uses two tests for specification inference, Angelix generates more patches (increased from 81 to 126 patches) but escalates the overfitting rate (from 54 to 81%). The similar trend can be seen for CVC4, Enumerative, and SemFix. This results actually contradict our hypothesis.

[7] Recall the explanation in Section 2.3 on the number of tests used for specification inference.

We believe that this fact could be due to a combination of several reasons. When using all repair (training) tests for inference task, once a solution is synthesized consistent with the specifications, it satisfies the whole repair test suite and thus regarded as a patch. Therefore, if the repair test suite is weak enough to allow such a situation, it results in an increase in patch generation rate. However, the in-comprehensiveness of the repair test suite also brings about a reasonably high probability of the overfitting rate since the generated patches may not generalize. In fact, the size of repair test suite in the Codeflaws benchmark is quite small (only 3 tests on average), while the held-out test suite's size is much larger (40 tests on average) (Tan et al. 2017).

Research Question 8 How does the number of fault locations grouped together affect patch generation rate and overfitting?

The second tunable feature we study is the effect of grouping faulty locations. The larger the group, the expressions considered for repair at once. We observe the behaviors of different repair techniques when the group size is set to 3 and 4, respectively. We note that SemFix is left out in this research question since it is not able to fix multi-line bugs (Nguyen et al. 2013).

Results Table 5 shows the results on both IntroClass and Codeflaws benchmarks. Overall, the number of generated patches and the overfitting rate when group size varies only slightly between the group sizes. On IntroClass, Angelix generates 49 and 46 patches, with overfitting rates of 67% and 74%, when group size is set to 3 and 4, respectively. As compared to research question 3, which uses default group size of two, the number of generated patches substantially decreases, e.g., from 81 to 49 and 46 for Angelix. CVC4 and Enumerative show a similar trend. We hypothesize that increasing the number of likely-buggy locations being fixed proportionally enlarges the search space, and subsequently makes it harder to generate patches.

The same trend generally holds on the Codeflaws dataset, with an interesting exception for Angelix. Angelix generates more patches (87 vs 108), while reducing the overfitting rate slightly (86% vs 82%) when group size is varied from 3 to 4. This shows that Angelix's ability in fixing multi-line bugs is potentially helpful in this case.

3.5 Examples from the IntroClass Dataset

We now present and discuss several examples that may provide deeper insights into the overfitting issue for semantics-based APR.

Table 5 The overfitting rate of Angelix, CVC4, Enumerative, and SemFix in subject programs from IntroClass (omitting syllables and checksum, for which no patches were generated) and Codeflaws, when group size is set to three and four, respectively

	Size 3			Size 4		
Subject	Angelix	CVC4	Enum	Angelix	CVC4	Enum
smallest	8/20	10/19	10/20	12/18	10/19	11/21
median	18/24	17/ 23	18/23	17/23	17/23	19/25
digits	5/5	5/5	5/5	5/5	5/5	5/5
Codeflaws	75/87	84/86	57/60	89/108	43/44	48/49

```
1   void median(int n1, n2, n3) {
2     int small;
3     if (n1 < n2){
4       small = n1;
5       if (small > n3)
6         printf("%d\n", ANGELIX_OUTPUT(int, n1, "stdout"));
7       else if (n3 > n2)
8         printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
9       else
10        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
11    } else {
12      small = n2;
13      if (small > n3)
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n3 > n1 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
17      else
18        printf("%d\n", ANGELIX_OUTPUT(int, n1, "stdout"));
19    }
20  }
```

Fig. 5 Example of a buggy *median* program (simplified slightly for presentation). The buggy line is shaded in blue at line 15. The *ANGELIX_OUTPUT* macro is explicitly required Angelix instrumentation; it indicates output variables to Angelix

Figure 5 shows an example of a buggy *median* program. The goal of a *median* program is to identify the median value between three integer inputs. The buggy line in our example is colored blue, at line 15. We now consider Angelix-generated patches for this example program using 25% and 50% of the black-box tests for training. Figure 6a shows the Angelix patch for the program in Fig. 5 using 25% of black-box tests for training. The patch considers the expressions at lines 13 and 15, respectively, for repair (colored red). Line 13 remains unchanged, while the true buggy condition at line 15 is changed. This patch

```
11    else {
12      small = n2;
13      if ( small > n3 )
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n1 > n1 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
```

(a) Patch created using 25% of black-box tests, changing lines 17 and 19 of the original program, shaded in red.

```
11    else {
12      small = n2;
13      if ( small > n3 )
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n1 > n3 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
```

(b) Patch created using 50% of the black-box tests, changing lines 17 and 19 of the original program, shaded in red.

Fig. 6 Patches generated by Angelix for the program in Fig. 5 using 25% of the black-box tests (top) and 50% of the black-box tests (bottom) as training tests. Line numbers are aligned with those in Fig. 5

Table 6 Specifications inferred by Angelix for the example in Fig. 5 using 50% of black-box tests for training

Test ID	n1	n2	n3	Expected State
#3	6	2	8	(L13 → false) ∧ (L15 → false)
#5	8	2	6	(L13 → false) ∧ (L15 → true)

(The first row shows the specification inferred using only 25% of the tests for training). The first column shows the test id. The next three columns show values of *n1*, *n2*, and *n3*, respectively. The last column shows the expected states at different lines given the input values. For example, (L13 → false) ∧ (L15 → false) in test id #3, means the expected state of the *if-conditions* at lines 13 and 15 are both false

overfits, such that the resulting program does not pass all held-out tests (such as the test $\{n1 = 8, n2 = 2, n3 = 6\}$).

To better understand this issue, consider the specification inferred by Angelix that lead to this erroneous patch. The first line of Table 6 shows the specification that lead to this patch, produced on a test with input values of $n1 = 6, n2 = 2$, and $n3 = 8$. This specification indicates that this test would pass if the states of the *if-conditions* at lines 13 and 15 are both *false*, which the patch in Fig. 6a satisfies. This shows the danger of weak specifications.

Returning to the example program (Fig. 5), consider adding an additional test, with associated inferred specification, shown in the second line of Table 6. Adding this test to the training set leads Angelix to find the patch shown in Fig. 6b, which is generally correct in the way it changes the logic of the *if-condition* at line 15. In this case, increasing the number of training tests (from 25 to 50% of black-box tests) provided a huge benefit: This patch fully generalizes to the held-out tests, and it better matches our intuition. One conclusion is that additional tests can help guide synthesis to a better repair, which is especially satisfying in this case, where only two total are required.

Figure 7 shows patches generated by Angelix's synthesis engine and SyGuS engines for another *median* program. Angelix's patch replaces line 4 with line 5; the SyGuS engines (including CVC4 and Enumerative) replace line 4 with line 6. Angelix's generated patch is incorrect; the SyGuS- generated patch is correct. Angelix's synthesis engine does not force generalization, where a generalized solution involves as few constants as possible (Gulwani et al. 2016). SyGuS engines, on the other hand, are more flexible in forcing generalization by simply emphasizing permitted constants after variables in its grammar. This suggests a straightforward strategy to improve the generality of patches produced by such techniques.

Figure 8 shows an example of an overfitting patch for the *smallest* subject program, generated by Angelix when using all black-box tests as training tests. The goal of the *smallest* program is to return the smallest of three integer numbers. The patch generated by Angelix replaces line 2 with line 3, loosening the *if-condition*. As with our prior example, this patch

```
1      if (num1 > num2) {...}
2      else {
3        big = num2;
4  -     small = num2;
5  +     small = 6; // by Angelix
6  +     small = num1 // by SyGuS
```

Fig. 7 Patches generated by Angelix's synthesis engine, and SyGuS engines for a *median* program

```
 1        ...
 2  -   if((a < b) && (a < c) && (a < d))
 3  +   if(((a < d) && (a < d)))
 4          printf("%d\n",ANGELIX_OUTPUT(int,(int) a, "stdout"));
 5  -   else if ((b < a) && (b < c) && (b < d))
 6  +   else if ((((b < a) && (b < c)) && (b < d)))
 7          printf("%d\n",ANGELIX_OUTPUT(int,(int) b, "stdout"));
 8      else if ((c < a) && (c < b) && (c < d))
 9          printf("%d\n",ANGELIX_OUTPUT(int,(int) c, "stdout"));
10  }
```

Fig. 8 Example of an overfitting patch for the *smallest* subject program, generated by Angelix when using all black-box tests as training tests

clearly overfits to a particular set of tests. This example demonstrates that overfitting can occur even when a full set of black-box tests is used.

This bug would likely benefit from multi-location patch that adds equality signs to each of the conditions in lines 2, 5 and 8. Yet, Angelix's ability to generate multi-location patch does not help in this case. Angelix's ad-hoc approach to deciding how many buggy locations to consider, and how to group them, could be improved with stronger heuristics, more accurate fault localization, or more precise dataflow information to better group the three implicated conditions. Generally, however, these results call for the development of stronger or alternative synthesis engines that are more resilient to overfitting.

3.6 Qualitative Study on Real-World Bugs

Our results in preceding sections describe program repair as applied to small programs with two independent test suites. We now present qualitative results assessing the performance of different synthesis engines on defects in large, real-world programs. Automatically generating independent full-coverage test suites on real-world programs is prohibitive. As such, we employ a stricter proxy to assess the correctness of machine-generated patches: A machine-generated patch is considered correct if it is equivalent to the patch submitted by developers. Two patches are considered equivalent if: (1) they are syntactically identical, or (2) one patch can be transformed into the other via basic syntactic transformation rules. For example, `a || b` and `b || a` are considered equivalent if no observable side effects occur when evaluating either a or b, e.g., exceptions thrown, modifications to global variables, etc. We choose syntactic transformations as the proxy for correctness validation because checking for semantic equivalence is a hard problem, and undecidable in general. A semantic equivalence check may involve deep human reasoning, which may be subjective. We thus use syntactic equivalence to ease the validation process and avoid subjectivity, although it may be overly strict in certain cases.

To ease this manual process, we require a transparent baseline, in that the developer-submitted patches (ground truth) must be sufficiently concise and transparent to support a manual patch-equivalence check. Unfortunately, existing benchmarks (such as ManyBugs (Le Goues et al. 2015)) often include changes of multiple lines, complicating manual correctness assessment. To this end, we reuse a benchmark consisting of nine real-world bugs in large programs (such as the Common Maths library, consisting of 175 kLOC), and tool named JFIX, from our work in Le et al. (2017a). JFIX adapts the Angelix specification inference engine to Java programs (Le et al. 2017a), and uses the synthesis engine of Angelix, CVC4, and Enumerative to synthesize repairs conforming to the inferred specifications. We omit SemFix, because it generally does not scale to these programs (Nguyen et al. 2013).

Table 7 shows results. Note that this study extends our previous work (Le et al. 2017a) by further studying the effect of a majority voting scheme. That is, we check whether we can choose a correct patch from a set of patches suggested by the synthesis engine by employing majority voting (i.e., patches that are generated by most of the synthesis engines are chosen). Machine-generated patches that are equivalent to patches submitted by developers are indicated by "✔", and "✗" indicates otherwise in Table 7.

As Table 7 shows, Angelix, Enumerative and CVC4 can complement one another. There are no bugs for which all three techniques produce equivalent patches, but for all bugs, at least one technique does succeed. For example, there are three bugs for which Enumerative and CVC4 can generate patches equivalent to their developer counterparts, where Angelix does not. This shows that the nature of each repair technique may lead to different kind of patches, suggesting that employing an agreement (majority voting) on generated solutions between different synthesis engines may increase confidence when choosing a correct patches. The high level idea is that the more different synthesis engines agree on a solution, the higher the confidence that the solution is correct. For example, CVC4 and Enumerative generate the same (correct) solutions for *SFM* and *EWS*, while Angelix generates a different solution. We note that this majority voting method can still lead to incorrect patches. For example, Angelix generates a correct patch for *Qabel*, but CVC4 and Enumerative both generate the same incorrect patch. Thus, if majority voting is employed, it leads to an incorrect patch in this case. However, as the overall results indicate, the majority voting method is more effective than any individual technique in identifying correct patches.

4 Threats to Validity

Threats to internal validity relate to errors in our instrumentation and experiments. Angelix requires users to instrument the program under repair with its provided C macros in order to

Table 7 Real bugs collected from real-world software

Project	Rev	Type	Angelix		Enum		CVC4		Majority Voting
			Time	Dev	Time	Dev	Time	Dev	
Math	09fe	II	23s	✔	26s	✔	36s	✔	✔
	ed5a	I & II	168s	✔	NA	✗	NA	✗	✔
Jflex	2e82	II	NA	✗	70s	✔	72s	✔	✔
Fyodor	2e82	II	20s	✔	19s	✔	31s	✔	✔
SFM	5494	II	12s	✗	10s	✔	13s	✔	✔
EWS	299a	I	NA	✗	14s	✔	258s	✔	✔
Orientdb	b33c	II	20s	✔	22s	✔	NA	✗	✔
Qabel	299c	II	37s	✔	22s	✗	23s	✗	✗
Kraken	8b0f	II	12s	✔	13s	✗	15s	✗	✗

Column "Rev" shows the revisions that fix the bugs. Column "Type" shows the bug types: "I" denotes method call, "II" denotes arithmetic. Column "Time" indicates the time required (in seconds) to generate the repair ("NA" denotes not available). Column "Dev" indicates whether a generated repair is equivalent to the repair submitted by developers. The "✔" denotes equivalent, and the "✗" denotes otherwise. The "Majority Voting" column shows the result of using majority voting method to choose correct patches capture the program's output. This task involves manual effort and some degree of understanding the program under repair. Thus, it is a somewhat error-prone process. There could be hidden errors that we did not notice despite our effort on rechecking our instrumentation and experiments. Our experiments are reproducible, using the release of Angelix, mitigating this risk. The results of our experiments are hosted at Zenodo with DOI: https://doi.org/10.5281/zenodo.1012686, accessible from the following link: https://goo.gl/4BwK3s.

Threats to external validity correspond to the generalizability of our findings. We have analyzed several hundreds of bugs from a dataset that has been used to systematically evaluate overfitting in heuristic repair techniques (Smith et al. 2015) and assess quality of patches produced by new program repair approaches (Ke et al. 2015). More programs with real bugs would further help mitigate this threat, though such a dataset (with two comprehensive test suites) may be difficult to construct. Additionally, we could complement our study with more repair and synthesis techniques to compare and contrast the strengths and weaknesses of them.

Threats to construct validity correspond to the suitability of our evaluation metrics. We use the fact of a patch's generality to a held-out set of tests as a proxy for its quality. Note that this does not necessarily mean that the patch is fully correct, nor that it would be accepted by humans with other quality goals in mind. However, this metric is consistent with that used in other recent studies Smith et al. (2015) and Le et al. (2016b), and repair quality is an important and unsolved problem in program repair research. We also used a stricter metric to evaluate patch correctness by checking whether a machine-generated patch is equivalent to the patch submitted by developer via manual syntactic transformations. Still, this metric could be too strict and overstate the incidence of overfitting patches. We leave the consideration of other such criteria to future work.

5 Related Work

In this section, we described related work in automated program repair, including repair techniques, datasets and empirical studies on program repair.

Program Repair Recent years have seen a proliferation in the development of program repair techniques. Repair techniques can generally be divided into two families: heuristic vs semantics-based families. Heuristic approaches generate a large number of repair candidates and employ search or other heuristics identify correct repairs among them. Semantics-based approaches extract semantics constraints from test suites, and synthesize repairs that satisfy the extracted constraints. GenProg (Weimer et al. 2010; Le Goues et al. 2012) is an early APR tool that uses genetic programming as a heuristic to search for a repair that causes a program to pass all provided tests cases among a possibly huge pool of repair candidates. AE (Weimer et al. 2013) leverages an adaptive search approach to search for similar syntactic repairs. Qi et al. (2014) propose RSRepair, which uses a random search approach to find repairs, and show that RSRepair is better than GenProg on a subset of GenProg's benchmark. Other recent techniques belong to the heuristic repair (Long and Ribard 2015, 2016b; Le et al. 2016c) use condition synthesis, and development history (Kim et al. 2013) to guide the search for repair. Semantics-based repair approaches (Könighofer and Bloem 2011; Nguyen et al. 2013) typically use symbolic execution and program synthesis to extract semantic constraints and synthesize repairs that satisfy the constraints; other work exists at the intersection of the two families (Ke et al. 2015). There further exists work that is farther afield that uses abstract interpretation, unguided by test suites, but requiring specially-written, well-specified code (Logozzo and Ball 2012). We study Angelix (Mechtaev et al. 2016), a recently-proposed repair tool that makes the semantics-based approach scale to repair large real-world programs by using a selective symbolic execution procedure to replace classical symbolic execution used for repair in previous techniques. In a similar vein, for Java, Nopol (DeMarco et al. 2014) translates the Object-oriented program repair problem into SMT formulae and uses a constraint solver to synthesize repairs. Our recent work JFIX translates and extends Angelix to work on Java programs, which we adapt to study real-world bugs here (Le et al. 2017a).

In this work, we do not propose a new program repair technique, but rather assess and compare the quality of output patches from a well-known recent technique in semantics-based repair.

Dataset Researchers have created a number of benchmarks intended for research and empirical studies in testing, fault localization, and program repair. Defects4J (Just et al. 2014) includes more than 300 real-world defects from five popular Java programs. Defects4J is originally intended for to facilitate fault localization research, but it is likely suitable for program repair research as well. We do not focus on it because Angelix targets C programs. The ManyBugs and IntroClass benchmarks (Le Goues et al. 2015) provide collections of bugs for C programs, including large real-world C programs, and small C programs as students' homework assignments. The two benchmarks serve different empirical purposes, and are suitable for different types of studies. In this work, we consider the IntroClass benchmark, which contains several hundreds of small C programs, written by students as homework assignments in a freshmen class. Although the IntroClass benchmark only contains small programs, its unique feature is that it includes the two high-coverage test suites: black-box test suite generated by the course instructor, and white-box test suite generated by the automated test generation tool KLEE (Cadar et al. 2008). This feature makes it suitable for assessing overfitting in automated program repair; one test suite can be used for repair, and the remaining test suite can be used for assessing the quality of generated patches. Recently, Codeflaws was proposed as another benchmark for assessing automatic repair techniques following the spirit of the IntroClass benchmark. Codeflaws contains 3,902 defects from 7,436 small programs from programming contests hosted by Codeforces,[8] each of which contains two independent test suites.

Empirical Studies on Program Repair The rapid growth of program repair techniques has motivated empirical studies that compare and reveal strengths and weaknesses of different repair techniques. Qi et al. (2015) introduce the idea of a plausible versus correct patch, manually evaluating patches produced by previous heuristic techniques to highlight the risks that test cases pose when guiding repair search. The previous study that is closest to our work is by Smith et al. (2015), empirically and systematically studying the overfitting issue for heuristic repair techniques, including GenProg and RSRepair. Our study complements this previous study, in that we investigate the overfitting issue in the semantics-based repair family. Long and Rinard (2016a) study the search space to find repair of the heuristic approaches, showing that the search space is often large and correct repair sparsely occur within the search space. Le et al. (2016b) empirically study the effectiveness of many

[8] http://codeforces.com/

synthesis engines when employed for semantics-based program repair, suggesting that many synthesis engines could be combined or used at the same time to enhance the ability of semantics-based repair to generate correct repairs. We leverage the technique from Le et al. (2016b) to use multiple SyGuS engines in the context of a semantics-based repair approach.

Overfitting or manual annotation are not the only measure by which patch quality may be assessed. In proposing Angelix, Mechtaev et al. (2016) assess functionality deletion as a proxy for quality. Le Goues et al. (2012) evaluate generated patches in a case study context, quantitatively assessing their impact in a closed-loop system for detection and repair of security vulnerabilities. Kim et al. (2013) assess relative acceptability of patches generated by a novel technique via a human study. Fry et al. (2012) conduct a human study of patch maintainability, finding that generated patches can often be as maintainable as human patches. While overfitting as measured by high-quality test suites provide one signal about patch quality, human acceptability and real-world impact are also important considerations, if not more so, and should also be considered in characterizing the pragmatic utility of novel APR techniques.

6 Conclusions

Recent years have seen a proliferation of automated program repair techniques (APR), including heuristic and semantics-based APR, each with different strengths and weaknesses. Although the techniques have successfully tackled some of the main issues in APR (e.g., scalability), one special concern that still remains unaddressed is patch quality. The quality of patches generated by these APR techniques is not always assured, partly but not entirely due to the inadequacy of test suites in practice.

In this work, we perform the first study on overfitting issue in semantics-based APR. We show that semantics-based APR techniques do indeed produce patches that overfit. We further study the nature behind the overfitting in semantics-based APR by assessing the relationships between test suite coverage and provenance, number of failing tests, and semantics-specific tool settings and overfitting. Particularly, we find that in some cases results are consistent with those found for heuristic approaches, while in other cases results are interestingly inconsistent. We also present several case studies of overfitting patches produced by semantics-based APR techniques, with implications and observations for how to improve them. For example, we observe that one possible source for overfitting in semantics-based APR could be due to the "conservativeness" of the underlying synthesis engine, that returns the first solution found (without consideration of alternatives). Also, to mitigate overfitting in semantics-based APR, we substantiate that using multiple synthesis engines could be one possible approach, as mentioned in Le (2016). In the future, we plan to develop a synthesis technique that can combine the strengths of many synthesis techniques to help semantics-based APR overcome the overfitting issue. We also plan to develop a specification inference technique, e.g., specification mining techniques such as SpecForce (Le et al. 2015a), that can infer a stronger specifications to help better capture the semantics of the program under repair. Another future direction is to use machine learning techniques to automatically classify defect types, e.g., Thung et al. (2015), which could help deal with each bug type more effectively.

Acknowledgements We thank the authors of Angelix for making the tool publicly available. We also thank the authors of CVC4 and Enumerative synthesis engines for making these engines accessible.

References

Abreu R, Zoeteweij P, Van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: Testing: academic and industrial conference practice and research techniques-MUTATION, 2007. TAICPART-MUTATION 2007. IEEE, pp 89–98

Alur R, Bodik R, Juniwal G, Martin MM, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2015) Syntax-guided synthesis. Dependable Software Systems Engineering 40:1–25

Cadar C, Dunbar D, Engler DR et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating systems design and implementation, OSDI, pp 209–224

DeMarco F, Xuan J, Le Berre D, Monperrus M (2014) Automatic repair of buggy if conditions and missing preconditions with SMT. In: Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis, pp 30–39

Fry ZP, Landau B, Weimer W (2012) A human study of patch maintainability. In: International symposium on software testing and analysis (ISSTA), pp 177–187

Gulwani S, Esparza J, Grumberg O, Sickert S (2016) Programming by examples (and its applications in data wrangling). Verification and synthesis of correct and secure systems

Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, ICSE, pp 215–224

Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, ISSTA, pp 437–440

Ke Y, Stolee KT, Le Goues C, Brun Y (2015) Repairing programs with semantic code search. In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering, pp 295–306

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: ACM/IEEE international conference on software engineering, ICSE, pp 802–811

Könighofer R, Bloem R (2011) Automated error localization and correction for imperative programs. In: Formal methods in computer-aided design, IEEE, FMCAD, pp 91–100

Könighofer R, Bloem R (2012) Repair with on-the-fly program analysis. In: Haifa verification conference. Springer, pp 56–71

Le XBD (2016) Towards efficient and effective automatic program repair. In: International conference on automated software engineering (ASE). IEEE, pp 876–879

Le TDB, Le XBD, Lo D, Beschastnikh I (2015a) Synergizing specification miners through model fissions and fusions. In: International conference on automated software engineering (ASE). IEEE, pp 115–125

Le XBD, Le TDB, Lo D (2015b) Should fixing these failures be delegated to automated program repair? In: International symposium on software reliability engineering (ISSRE). IEEE, pp 427–437

Le XBD, Le QL, Lo D, Le Goues C (2016a) Enhancing automated program repair with deductive verification. In: International conference on software maintenance and evolution (ICSME). IEEE, pp 428–432

Le XBD, Lo D, Le Goues C (2016b) Empirical study on synthesis engines for semantics-based program repair. In: International conference on software maintenance and evolution (ICSME). IEEE, pp 423–427

Le XBD, Lo D, Le Goues C (2016c) History driven program repair. In: Proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering, SANER, pp 213–224

Le XBD, Chu DH, David L, Le Goues C (2017a) Jfix: sematics-based repair of java programs via symbolic pathfinder. In: Proceedings of the 2017 ACM international symposium on software testing and analysis, ISSTA

Le XBD, Chu DH, Lo D, Le Goues C, Visser W (2017b) S3: syntax-and semantic-guided repair synthesis via programming by examples. In: Joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM, pp 593–604

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) 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, pp 3–13

Le Goues C, Nguyen T, Forrest S, Weimer W (2012) Genprog: a generic method for automatic software repair. IEEE Trans Softw Eng 38(1):54–72

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans Softw Eng 41(12):1236–1256

Logozzo F, Ball T (2012) Modular and verified automatic program repair. SIGPLAN Not 47(10):133–146

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM, pp 166–178

Long F, Rinard M (2016a) An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th international conference on software engineering. ACM, pp 702–713

Long F, Rinard M (2016b) Automatic patch generation by learning correct code. In: ACM SIGPLAN notices, vol 51. ACM, pp 298–312

Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 20(1):176–205

Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: Proceedings of the 37th ACM/IEEE international conference on software engineering, ICSE, pp 448–458

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th ACM/IEEE international conference on software engineering, ICSE, pp 691–701

Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 international conference on software engineering, ICSE, pp 772–781

Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th international conference on software engineering, ACM, ICSE, pp 254–265

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA, pp 24–36

Reynolds A, Deters M, Kuncak V, Tinelli C, Barrett C (2015) Counterexample-guided quantifier instantiation for synthesis in SMT. In: International conference on computer aided verification. Springer, pp 198–216

Smith EK, Barr ET, Le Goues C, Brun Y (2015) Is the cure worse than the disease? Overfitting in automated program repair. In: Joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ACM, ESEC/FSE, pp 532–543

Tan SH, Yi J, Yulis, Mechtaev S, Roychoudhury A (2017) Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In: ICSE Poster, to appear

Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI project 7007(011)

Thung F, Le XBD, Lo D (2015) Active semi-supervised defect categorization. In: International conference on program comprehension (ICPC). IEEE Press, pp 60–70

Weimer W, Forrest S, Le Goues C, Nguyen T (2010) Automatic program repair with evolutionary computation. Commun ACM 53(5):109–116

Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: models and first results. In: Proceedings of the 28th international conference on automated software engineering, IEEE, ASE, pp 356–366

Xuan J, Martinez M, DeMarco F, Clément M, Lamelas S, Durieux T, Le Berre D, Monperrus M (2016) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans Softw Eng https://doi.org/10.1109/TSE.2016.2560811. https://hal.archives-ouvertes.fr/hal-01285008/document

Xuan Bach D. Le is PhD students at Singapore Management University since 2014 and 2013, respectively.

Ferdian Thung is PhD students at Singapore Management University since 2014 and 2013, respectively.

David Lo is associate professor at Singapore Management University, received PhD from National University of Singapore in 2007.

Claire Le Goues is assistant professor at Carnegie Mellon University, received PhD from the University of Virginia.