S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples

Xuan-Bach D. Le
Singapore Management University
Singapore
dxb.le.2013@smu.edu.sg

Duc-Hiep Chu
Institute of Science and Technology
Austria
duc-hiep.chu@ist.ac.at

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Claire Le Goues
Carnegie Mellon University
Pittsburgh, USA
clegoues@cs.cmu.edu

Willem Visser
Stellenbosch University
South Africa
wvisser@cs.sun.ac.za

ABSTRACT

A notable class of techniques for automatic program repair is known as *semantics-based*. Such techniques, e.g., Angelix, infer semantic specifications via symbolic execution, and then use program synthesis to construct new code that satisfies those inferred specifications. However, the obtained specifications are naturally incomplete, leaving the synthesis engine with a difficult task of synthesizing a *general* solution from a sparse space of many possible solutions that are consistent with the provided specifications but that do not necessarily generalize.

We present S3, a new repair synthesis engine that leverages programming-by-examples methodology to synthesize high-quality bug repairs. The novelty in S3 that allows it to tackle the sparse search space to create more general repairs is three-fold: (1) A systematic way to customize and constrain the syntactic search space via a domain-specific language, (2) An efficient enumeration-based search strategy over the constrained search space, and (3) A number of ranking features based on measures of the syntactic and semantic distances between candidate solutions and the original buggy program. We compare S3's repair effectiveness with state-of-the-art synthesis engines Angelix, Enumerative, and CVC4. S3 can successfully and correctly fix at least three times more bugs than the best baseline on datasets of 52 bugs in small programs, and 100 bugs in real-world large programs.

CCS CONCEPTS

• **Software and its engineering → Programming by example**; *Dynamic analysis*; *Software testing and debugging*;

KEYWORDS

Program Repair, Programming by Examples, Inductive Synthesis, Symbolic Execution

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106309

ACM Reference format:
Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106309

1 INTRODUCTION

Bug fixing is notoriously difficult, time-consuming, and costly [5, 46]. Hence, automating bug repair, to reduce the onerous burden of this task, would be of tremendous value. Automatic program repair has been graining ground, with substantial recent work devoted to the problem [6, 20, 24–26, 29, 32, 34–36, 51, 52], inspiring hope of future practical adoption. One notable line of work in this domain is known as *semantics-based* program repair, most recently embodied in Angelix [36]. This class of techniques uses semantic analysis (typically dynamic symbolic execution) and a set of test cases to infer behavioral specifications of the buggy code, and then program synthesis to construct repairs that conform to those specifications. Such approaches have recently been shown to scale to bugs in large, real-world software [36].

Although scalability has been well-addressed, one pressing concern in program repair is patch quality, sometimes quantified in terms of patch *overfitting* or *generalizability* [43]. Generated repairs can sometimes overfit to the tests used for repair, and fail to generalize to a different set of tests. This may be caused by weak or incomplete tests, or even simply the nature of the repair technique [19, 43]. Various repair approaches have been shown to suffer from overfitting, including GenProg [29], RSRepair [39] and SPR [32]. Semantics-based approaches like Angelix [36], are no exception to this issue, as partially shown in recent studies [27]. Overfitting, and patch quality generally, remains a challenging problem in the program repair field.

One reason for patch overfitting is that the repair search space is often sparse, containing many plausible solutions that can lead the buggy program to pass a given test suite, but that may still be judged incorrect [33]. One way to tackle overfitting is thus to constrain the search space to patches that are more likely to generalize. Other strategies for increasing the quality of output

patches include higher-granularity mutation operators [19], anti-patterns [45], history-based patterns [28], feedback from execution traces [8], or document analysis [51]. Angelix [36] eagerly preserves the original syntactic structure of the buggy program via PartialMaxSMT-based constraint solving [35] and component-based synthesis [16]. However, such enforcement alone may not be enough [8]. Furthermore, incorporating other strategies or criteria into a constraint-based synthesis approach is non-obvious, since doing so typically requires novel, and often complicated constraint encodings (this problem has been pointed out by others, see, e.g., Chapter 7 of [14] or Section 2 of [45]). This motivates the design of a new repair synthesis technique that can consolidate various restrictions or patch generation criteria, enabling an efficient search over a constrained space for potentially higher-quality patches.

We present S3 (Syntax- and Semantic-Guided Repair Synthesis), a new, scalable repair synthesis system. S3 addresses the challenge of synthesizing generalizable patches via our novel design of three main components: (1) An underlying domain-specific language (DSL) that can systematically customize and constrain the syntactic search space for repairs, (2) An efficient enumeration-based search strategy over the restricted search space defined by the DSL to find solutions that satisfy correctness specifications, e.g., as induced by test suites, and (3) Ranking functions that serve as additional criteria aside from the provided specifications to rank candidate solutions, to prefer those that are more likely to generalize. Our ranking functions are guided by the intuition that a correct patch is often syntactically and semantically proximate to the original program, and thus measure such syntactic and semantic distance between a candidate solution and the original buggy program. Unlike other constraint-based repair synthesis techniques, our framework is highly customizable by design, enabling the easy inclusion of new ranking features — its design is inspired by the programming-by-examples (PBE) synthesis methodology [14].

Given a buggy program to repair and a set of test cases (passing and failing), S3 works in two main phases. The first phase automatically localizes a repair to one or more target repair expressions (e.g., branch condition, assignment right-hand-side, etc.). S3 runs dynamic symbolic execution on the test cases to collect *failure-free* execution paths through the implicated expressions. It then solves the collected path constraints to generate concrete expression values that will allow the tests to pass. These specifications, expressed as input- and desired-output examples, are input to the synthesis phase. The synthesis phase first constrains the syntactic search space of solutions via a DSL that we extend from SYNTH-LIB [3]. Our extension allows it to specify a starting *sketch*, or an expression that gives S3 clues about what possible solutions might look like. Here, the sketch is the original buggy expression under repair. Next, S3 forms a solution search space of expressions of the same size as the sketch. Finally, it ranks candidate solutions via a number of features that approximate the syntactic and semantic distance to the specified sketch. If S3 cannot find any solution of the same size as the sketch, it investigates expressions that are incrementally smaller or larger than the sketch, and repeats the process.

We evaluate S3 by comparing its expressive power and the quality of the patches it generates to state-of-the-art baseline techniques (Angelix [36]; and Enumerative [3], and CVC4 [41] two alternative syntax-guided synthesis approaches), on two datasets. The first dataset includes 52 bugs in small programs, a subset of the IntroClass benchmark [30] translated to Java [10].[1] The IntroClass dataset contains only small programs, but provides two high-coverage test suites for each, allowing an independent assessment of repair quality. The second dataset includes 100 large real-world Java bugs that we collected from GitHub. We focus on Java, and build a new dataset of real-world Java bugs, for several reasons. First, Java is the most popular and widely-used programming language, and its influence is growing rapidly.[2] Second, a realistic, real-world dataset with transparent ground truth — fixes submitted by developers — can simplify the critical process of assessing the correctness of fixes generated by program repair tools in the absence of two independent, high-quality test suites. Existing benchmarks often include bug fixes with many changed lines, which can include tangled changes such as new features or code refactoring [15]; even curated datasets such as Defects4J [18] contain many changes involving a large number of lines. This complicates evaluation of generated patch correctness. Our dataset is restricted to bugs whose fixes involve fewer than five lines of code, alleviating the risk of tangled code changes. As many current state-of-the-art repair tools target bugs that require only a small number of changed lines [34–36], our dataset is sufficient for assessing current research.

We assess the quality and correctness of generated repairs in several ways. For the IntroClass bugs, we assess correctness on independent, held-out test suites (those provided with the benchmark, as well as additional tests we generate), separate from those used to guide the repair. We use the developer-provided patches as ground truth for the 100 real-world bugs. For these bugs, we consider a generated patch correct if it is either (1) *syntactically* identical to the developer-provided patch, or (2) semantically equivalent via some (basic) transformations. On both datasets, S3 substantially outperforms the baselines. S3 generates correct patches for 22 of 52 bugs from the first dataset; Angelix, Enumerative, and CVC4 can generate correct patches for 7, 1, and 1 bug(s), respectively. On the large real-world dataset, S3 generates correct patches for 20 out of 100 bugs, while Angelix, Enumerative, and CVC4 can only generate correct patches for 6, 6, and 5 bugs, respectively.

In summary, our novel contributions include:

- We present S3, a scalable repair synthesis engine that is geared towards synthesizing generalizable repairs.
- We propose a novel combination of syntax- and semantic-guided ranking features to effectively synthesize high-quality repairs. New features along these lines can be straightforwardly integrated into S3, by design.
- We present a large scale empirical study on the effectiveness of different synthesis techniques in semantics-based program repair context. S3 substantially outperforms the baselines in terms of generated repair quality.
- We present a dataset consisting of several bugs from large real-world software with transparent ground truth, which can enable confident evaluation of machine-generated patch correctness.

[1] We use the subset of IntroClass to which our repair tools can apply, given their applicability to strictly integer and boolean domains.
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

```
1  if (sourceExcerpt != null) {
2    ...
3    -if (excerpt.equals(LINE) && 0 <= charno
4    -    && charno < sourceExcerpt.length()) {
5    +if (excerpt.equals(LINE) && 0 <= charno
6    +    && charno <= sourceExcerpt.length()) {
7    ...
8  }
```

Figure 1: A bug in Closure compiler, revision *1e070472*. The bug is at lines 3–4. The developer fix is shown on lines 5–6; it turns a < to a <= in the second line of the if condition.

	Input			Desired
		(M1)	*(M2)*	
Test	`charno`	`excerpt.equals(LINE)`	`sourceExcerpt.length()`	**Output**
A	7	`true`	7	`true`
B	10	`true`	10	`true`

Figure 2: Input-output examples for both variables and conditions, extracted for the Closure compiler bug described in Figure 1. We use M1 and M2 to refer to the conditions in columns 3–4 in subsequent exposition. The last column represents the desired output of the overall branch decision.

- We release source code for S3 and the aforementioned dataset, along with all results, in support of open science.[3]

The rest of the paper is structured as follows. Section 2 describes a motivating example, followed by Section 3 explaining our approach. Section 4 describes our experiments, results, and observations. Section 5 presents related work; Section 6 concludes.

2 MOTIVATING EXAMPLE

We begin by motivating our approach and illustrating its underlying insight by way of example. Figure 1 shows changes made to address a bug in the Closure compiler at revision *1e070472*. The bug lies in the if-condition expression at lines 3–4; the developer-submitted fix is depicted at lines 5–6. This bug can be repaired by simply changing `charno < sourceExcerpt.length()` to `charno <= sourceExcerpt.length()`, while the rest of the condition expression remains unchanged. Table 2 shows example input and desired-output examples extracted for this bug at the buggy if-condition on two failing test cases. For each test run, the input includes runtime values of variables and method calls at the buggy lines, while the output is the value of the branch condition for the buggy lines that would cause the test to pass. For example, for test 1, the input includes runtime values for method calls `excerpt.equals(LINE)` and the variable `charno`. The desired output of the branch condition is `true`. These input-output examples constitute incomplete specifications for each buggy line considered in the program; although they are incomplete, they are scalably and automatically derivable from provided test cases.

Given these specifications (examples), the space of possible satisfying solutions is large, and contains many undesirable options, such as `excerpt.equals(LINE)`, `excerpt.equals(LINE)|| 0 < charno`, both of which, among others, would lead to the desired outputs on the considered expressions. Such solutions, if returned by a repair synthesis engine, create low-quality, *overfitting* repairs that lead the program to pass all provided tests but are not correct. In fact, Angelix [36] generates an overfitting repair for this bug, substituting `0 < charno` for the entire if-condition expression on lines 3–4 (Section 4 provides details on our straightforward port of Angelix to Java). This repair is quite different from the original expression both syntactically (despite Angelix's use of constraints to enforce minimal syntactic differences from an original expression) and semantically. The generated condition is indifferent to values of `excerpt.equals(LINE)` and `sourceExcerpt.length()`, substantially weakening the branch condition with respect to the original buggy version.

These observations inform insights that can be used to filter trivial solutions. In this case, the correct solution is syntactically and semantically close to the original buggy expression. Fusing syntactic and semantic measures of proximity can help rank the solution space to favor those that are more likely to be correct. Our approach, S3, estimates these distances in several ways to constrain the syntactic solution synthesis space, increasing the likelihood or producing a generalizable patch (see Section 3.2.3). For the example in Figure 1, S3 synthesizes a patch that is identical to the one submitted by the developer.

3 METHODOLOGY

S3 works in two main phases. Given a buggy program and a set of test cases, the first phase (Section 3.1) localizes potentially buggy program locations and, for each buggy location, extracts input and desired output examples that describe passing behavior. The extracted examples are input to the second phase (Section 3.2), which synthesizes repairs that satisfy and also generalize beyond the provided examples.

3.1 Automatic Example Extraction

S3 first uses fault localization to identify likely-buggy expressions or statements in the buggy program. S3 runs the test cases and uses Ochiai [2] to calculate suspiciousness scores that indicate how likely a given expression or a statement is to be buggy. S3 iterates through each identified buggy location (or group of locations in the case of multi-location repair), to extract input-output examples via a selective, dynamic symbolic execution [7].[4] For each buggy location, S3 inserts a symbolic variable to represent/replace the expression at the selected location. It then invokes test cases on the instrumented programs to collect path conditions that do not lead to runtime errors such as assertion errors, array index out of bound errors, etc. Solving these failure-free execution paths returns concrete values of symbolic variables that then can serve as input-output examples. We implement selective symbolic execution procedure on top of Symbolic PathFinder (SPF) [38].

For example, consider the buggy code snippet in Figure 1. S3 identifies that the if-condition at lines 3-4 may be buggy. S3 then replaces the buggy if-condition with a symbolic variable α, making the if-condition becomes "`if(` α)". S3 runs dynamic symbolic execution on the instrumented program using the provided test cases to collect failure-free execution paths, runtime variable values, and

[3] https://xuanbachle.github.io/semanticsrepair/

[4] For simplicity, we describe the process with respect to a single location; it extends naturally, by installing symbolic variables at multiple locations at once.

method calls involved in the buggy location. Solving the collected path conditions returns the values in the output column of Figure 2, corresponding to desired values of the symbolic variable α.

Although this phase shares the same spirit as the specification inference step in Angelix [36], there are key differences. Angelix infers specifications by solving models of the form $pc \wedge O_a = O_e$, where pc is a path condition produced by symbolic execution of a test, O_a is the actual output, and O_e is the expected output that is typically manually provided by a user.[5] The models capture the idea that if the expected output matches the actual concrete test output, the corresponding path condition is a test-passing path. Solving all test-passing paths returns specifications that lead all tests to pass. This process, however, can be tedious and error-prone, since it usually requires users to instrument output variables manually. For instance, if the output is a large array of many elements, users must give all expected outputs for all the elements of the array.

S3 extracts examples in an automated manner by building on SPF [38] automatic JUnit-test interpretation abilities. For a location i, S3 extracts examples by solving models of the form $pc \wedge$ *no errors*. pc is the path condition: $\bigwedge_{j=1}^{i} pc_j$. The "*no errors*" notation means that the conditions describe paths that are guaranteed to not yield assertion errors (as described above). If the path condition pc yields an assertion error, S3 automatically discards that path. In another case, if an array-out-of-bound error happens, S3 pops the latest pc_i leading to the error, keeping previous ones: $\bigwedge_{j=1}^{i-1} pc_j$. This frees S3 users from manual effort, while guaranteeing that the examples are still failure-free.

3.2 Repair Synthesis from Examples

Examples extracted in the previous phase are input as correctness specifications to the repair synthesizer. The goal of the synthesizer is to inductively construct a solution that satisfies and also generalizes beyond the provided specifications. This synthesis procedure is composed of three main parts: (1) a domain-specific language (DSL), (2) a search procedure, and (3) ranking features. We begin with an overview, and detail each component subsequently.

We start with a DSL (extended from SYNTH-LIB [3]) over the integer and boolean domains. Given a background theory T permitted by the DSL, let u be the original buggy expression, ϕ a formula over the vocabulary of T representing the correctness specifications (input-output examples), and L a set of expressions over the vocabulary of T of the same type as u. A *candidate fix* is an expression $e \in L$ such that $\phi[u/e]$ is valid modulo T.

Our algorithm then systematically enumerates all candidate fix expressions, considering them in ranked order. The ranking is performed by a set of N ranking functions $r_i (1 \leq i \leq N)$, each of which measures the distance between two expressions e_1 and e_2 of the same type. These ranking features estimate the syntactic and semantic distance between a candidate fix and the original buggy expression. The intuition is that expressions that are closer to the buggy program are more likely to constitute high-quality repairs.

Note, however, that the size of L (the search space) is often too large to be truly exhaustively enumerated. For practical purposes, we *greedily* favor candidate expressions of similar size and syntax to the original buggy expression. As described in Section 3.2.2, we systematically partition the search space, enabling different heuristics to be built without difficulty.

Algorithm 1 presents pseudocode for S3. At a high level, the search procedure enumerates all expressions in the grammar at a certain expression-size range (Line 4). S3 finds all candidate enumerated expressions that are consistent with the specifications (Line 7). Each candidate is assigned a ranking score by calculating the distance between it and the original buggy expression (Lines 8); candidates are sorted by score (Line 12). The process returns the solution in L with the smallest distance, if $L \neq \emptyset$ (Line 14). Otherwise, it continues until all expression size ranges have been exhausted (Line 3). S3 starts enumerating at the size of the original buggy expression (Line 2), and modifies the size range accordingly up to a bound b (Line 3). The original buggy expression and its size are made available to the synthesis procedure through our "*sketch*" extension to the SYNTH-LIB syntax (Section 3.2.1).

Algorithm 1 Enumeration-based synthesis procedure

```
Input:
    u                                   ▷ Original buggy expression
    φ                                   ▷ Correctness specifications
    G                                   ▷ SYNTH-LIB grammar (extended)
    R                                   ▷ Set of ranking features
    b                                   ▷ Synthesis bound
 1: function Synthesis(u, φ, G, R, b)
 2:     i ← size of u
 3:     for k ← 0 to b do
 4:         A ← {e in grammar G | e of size from i − k to i + k }
 5:         L ← {}
 6:         for all e ∈ A do
 7:             if φ[e/u] is valid then
 8:                 e.score ← Σ_{r_i ∈ R} r_i(e, u)
 9:                 L ← L ∪ e
10:             end if
11:         end for
12:         sort(L)                     ▷ by ascending order of score
13:         if L is not empty then
14:             return L.head           ▷ solution found
15:         end if
16:     end for
17:     return FAIL
18: end function
```

We next explain the DSL in detail (Section 3.2.1), the enumeration-based search procedure (Section 3.2.2), and the ranking features that we propose for the program repair domain (Section 3.2.3).

3.2.1 ***Domain-Specific Language via SYNTH-LIB.*** We extend SYNTH-LIB [3] to systematically constrain S3's search space. We choose SYNTH-LIB for three reasons:

(1) Balanced Expressivity. SYNTH-LIB is adequately expressive for various tasks in the program repair domain, while still sufficiently restrictive to allow an efficient search procedure. Figure 3 describes a simplified grammar for SYNTH-LIB. Note that it allows

[5] We refer readers to the Angelix manual: https://github.com/mechtaev/angelix/blob/master/doc/Tutorial.md

$$
\begin{aligned}
\mathit{IntExpr} &::= \mathcal{N} \mid \mathit{Var} \mid \mathit{IntExpr}\ \mathit{BinOp}\ \mathit{IntExpr} \\
\mathit{BoolExpr} &::= \mathit{IntExpr}\ \mathit{RelOp}\ \mathit{IntExpr} \mid \mathit{BoolExpr}\ \mathit{LogOp}\ \mathit{BoolExpr} \\
&\quad \mid \texttt{true} \mid \texttt{false} \mid \mathit{Var} \mid \neg \mathit{BoolExpr} \\
\mathit{RelOp} &::= > \mid < \mid \leq \mid \geq \mid = \\
\mathit{LogOp} &::= \wedge \mid \vee \mid = \quad \mathit{BinOp} ::= + \mid -
\end{aligned}
$$

Figure 3: Simplified SYNTH-LIB grammar used in S3.

the definition of integer expressions (*IntExpr*), including integer constants ($\mathcal{N}$), integer variables, and binary relations. Boolean expressions are defined similarly. Although simple, this grammar is sufficiently expressive for repairs over integers in booleans, including linear computations and logical relationships.

(2) Availability. SYNTH-LIB is not esoteric, but instead, broadly available to various tools for Syntax-Guided Synthesis (SyGuS) [3]. This allows for easy comparisons between tools, and indeed we use SYNTH-LIB to compare S3 with two other state-of-the-art SyGuS solvers (*Enumerative* [3] and *CVC4* [41]). We believe that an abundance of synthesis techniques will benefit the program repair domain, given the rapid growth of the SyGuS research community, along with publicly available implementations [3, 4, 41].

(3) Cost Metrics. SYNTH-LIB allows for definition of cost metrics like expression size; this is useful for calculating ranking features. We further extended SYNTH-LIB to allow the specification of a starting *sketch*, which gives clues on where the enumeration procedure should start. In our case, the starting sketch is the original buggy expression, capturing our idea that the correct fix is more likely to be syntactically and semantically close to the original code. The sketch allows ranking features to measure the distance between candidate solutions and the original expression(s).

We illustrate with a SYNTH-LIB script for the example in Figure 1; Figure 4 shows the corresponding SYNTH-LIB script. In Figure 4, the first line sets the background theory of the language to Linear Integer Arithmetic (*LIA*). The function being synthesized f is of type INT→INT→BOOL→BOOL, (keyword *synth-fun*). The permitted solution space for the function f is described in its body, which allows expressions of type boolean. Each boolean expression can then be formed by logical relationships between any two integer or boolean expressions, via relational or logical operators. Expressions can also be variables; $M1$ in this case is a boolean expression. The allowed integer expression in the grammar is defined via *IntExpr*, which includes integer variables such as `charno` and $M2$, and constants such as 0.

We next define the constraints consisting of input-output examples and the starting sketch. Each constraint is defined by the keyword *constraint*. In our example, the first constraint says that if the value of $M2$ is 7, the value of $M1$ is `true`, and the value of *charno* is 7, the expected output of the function f over `charno`, $M2$, and $M1$ is `true`. The second constraint can be interpreted similarly. These constraints corresponding to the extracted input-output examples described in Figure 2. A sketch, the starting-point expression, is defined by the keyword *sketch*. Here, the sketch is the original

```
1  (set-logic LIA)
2  (synth-fun f ((charno Int) (M2 Int) (M1 Bool)) Bool
3      ((Start Bool (
4          (≤ IntExpr IntExpr) (< IntExpr IntExpr)
5          (or Start Start) (and Start Start)
6          M1))
7       (IntExpr Int (
8          charno M2 0
9       ))))
10 (declare-var charno Int)
11 (declare-var M2 Int)
12 (declare-var M1 Bool)
13 (constraint (⇒ (and (= M2 7) (and (= M1 true) (= charno 7)))
14             (= (f charno M2 M1) true)))
15 (constraint (⇒ (and (= M2 10) (and (= M1 true) (= charno 10)))
16             (= (f charno M2 M1) true)))
17 (sketch u ((charno Int) (M2 Int) (M1 Bool)) Bool
18             (and (and M1 (≤ 0 charno)) (< charno M2)))
19 (check-synth)
```

Figure 4: SYNTH-LIB script generated by S3 for the example in Figure 1, derived using the "Alternatives" layer described in Figure 5. M1 stands for `excerpt.equals(LINE)`, and M2 stands for `sourceExcerpt.length()`.

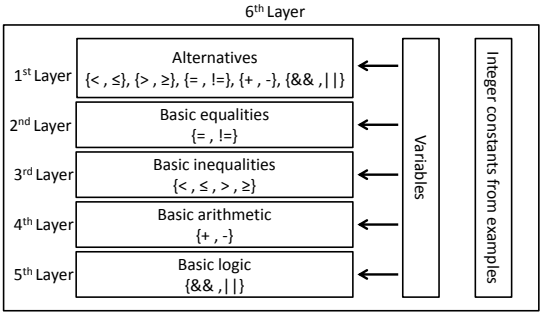

Figure 5: Search space layers specifiable in the grammar.

buggy expression u. Finally, the keyword *check-synth* instructs a synthesizer to start the synthesis process.

3.2.2 ***Enumeration-based Synthesis.*** S3 automatically generates a SYNTH-LIB script for each location under repair, and then uses an enumerative search to synthesize generalizable repair expressions conforming to the generated script. We note that multi-location repair can be achieved by generating the grammar for multiple functions simultaneously; we describe the process with respect to a single function for simplicity. We first explain how the SYNTH-LIB script is generated, and then the search procedure.

We divide the search space into multiple layers, each of which allows different components or operators, to appear in the SYNTH-LIB grammar script. If S3's search procedure cannot find a solution at a lower layer, it advances to the next. This approach tractably constrains the synthesis search space [37]. Figure 5 shows the six layers. The first layer allows *alternatives* of operators existing in the

original buggy expression. For example, a pair {"&&", "||"} means that the operators in the pair are alternatives of one another. If the search procedure cannot find any solution, the grammar then cumulatively allows additional variables that do not exist in the original buggy expression, denoted by the "*Variables*" component in the Figure 5. At the second layer, the grammar allows *basic-inequalities* operators (= and *!=*), in addition to operators in the original expression. Again, if this search fails, it cumulatively allows for additional *Variables*. Subsequent layers can be interpreted similarly. We note that at the last (sixth) layer, the grammar allows all components, including integer constants appearing in the input-output examples. The reason integer constants are considered last is that such constants may unduly allow trivial solutions; this choice is influenced by previous studies [14, 27].

The design of separate sub-search-spaces systematically allows us to either prioritize which space to explore first, or unify the spaces freely. We heuristically prioritize the search space by automatically analyzing the surrounding context of the original buggy statement, such as the method declaration that contains the buggy statement. Particularly, S3 automatically looks for expressions in the surrounding context that use the same variables appearing in the buggy statement, and analyzes the components used in those expressions. This gives S3 clues on which search space to start from. If the prioritized search space does not help find solutions, S3 searches in the unified search space (the sixth layer). If S3 cannot find context to help prioritize the space, it follows the procedure described previously, starting from the first layer.

3.2.3 ***Ranking Features.*** We employ the insight that a correct repair is often syntactically and semantically close to a buggy expression/statement [8]. We thus propose features that measure the syntactic and semantic distance between a candidate solution and the original buggy code. The final ranking score of a candidate solution is the sum of individual feature scores. S3 allows new features to be incorporated without difficulty; by contrast, constraint-based synthesis approaches (e.g. [35, 36]) typically require non-obvious Satisfiable Modulo Theory (SMT) encodings for new features [45].

Syntactic Features. Syntactic features look at differences between candidate solutions and the original buggy expression at the Abstract Syntax Tree (AST) level. We do this in three ways:

- **AST differencing**. We use GumTree [12] to compare ASTs. GumTree produces transformations between ASTs in the form of actions on AST nodes such as insert, delete, update, or move. We measure the number of actions needed to transform the original buggy AST to the candidate solution AST. This feature can be easily calculated by directly applying GumTree on the ASTs produced by parsing the SYNTH-LIB grammar script.
- **Cosine similarity**. An AST can also be represented as a vector of node occurrence counts [17]. The occurrence of each node type (e.g., integer variables or constants, or a binary operation) in an AST, represent a vector of the AST. The similarity of two ASTs can then be represented by the cosine similarity of their representative vectors, denoted as *cosine_score*. We then define the distance from the solution's AST to the original AST as: $1 - cosine\_score$ (*cosine_score* of 1 denotes that two vectors are identical). A SYNTH-LIB grammar explicitly enables type checking, meaning this feature is easy to calculate via an AST traversal to collect type information.
- **Locality of variables and constants**. Variables and constants are the primary ingredients of expressions. Thus, in addition to capturing abstract changes on the AST, we capture lower-level differences via the locations of variables and constants in expressions. We compute the Hamming distance between two vectors representing locations of variables and constants in each expression.[6] For example, consider $a \wedge (b < 1)$ as the original expression, $a \wedge (b \leq 1)$ as the first solution, and $(b \leq 1) \wedge a$ as the second solution. The hamming distance from the original expression for the first and second solutions are 0 and 3 respectively. Although both solutions are semantically equivalent, we may want to prefer the first in the interest of change minimality.

Semantic Features. Semantic features look at either the difference between a solution S_i and the original expression u, or the semantic quality of S_i itself. We propose three semantic features:

- **Model counting**. Model counting (c.f. [48]) is often used to count the number of models satisfying a particular formula. We use this feature to measure the level of "*disagreement*" between any two boolean expressions. That is, we say that a solution S_i and the original expression u disagree with each other if the formula $(S_i \wedge \neg u) \vee (\neg S_i \wedge u)$ is valid, meaning that S_i and u cannot be both valid at the same time. We then define the level of disagreement between S_i and u by the number of models that satisfy the formula, which accounts for the semantic distance between them. As a simple example, assume that we have: $a < 10$ as the original expression u, $a \leq 13$ as a solution S_1, and $a \leq 15$ as a solution S_2. The semantic distance via model counting between these solutions and u is 4 and 6, respectively. This simple example generalizes naturally to the typical off-by-one bug in Figure 1.
- **Output coverage**. This feature looks at how much a solution covers the set of outputs in the set of input-output examples. For instance, assume input-output examples (constraints) for two tests T_1 and T_2, on an input i, and an output o:
  T_1: $i = 5 \rightarrow o = 5$
  T_2: $(i = 6 \rightarrow o = 5) \vee (i = 6 \rightarrow o = 6)$
  A trivial solution for this example is simply the constant 5; Another solution is the expression i. The first solution overfits to only one output despite the presence of three examples that have two distinct outputs. The second solution covers all output scenarios in the provided examples, making it intuitively less overfitting as compared to the first. A solution S_i receives a O_i^{cov} score of N_c/N_o, where N_o is the number of output scenarios in the provided input-output examples, and N_c is the number of output scenarios that the solution S_i covers. The feature score of a solution S_i is defined as $1 - O_i^{cov}$. The higher O_i^{cov}, the better the solution S_i.
- **Anti-patterns**. This feature aims to heuristically prevent synthesis from generating trivial solutions. Particularly, these patterns are anti-duplicate and -constant expressions, e.g., $a < a$, $0 \neq 1$, etc. Expressions containing these patterns typically evaluate to a constant *true* or *false*, and are thus likely to overfit.

[6] https://en.wikipedia.org/wiki/Hamming_distance

We filter out these expressions during the synthesis process. Again, this can be easily done by traversing the AST produced by the SYNTH-LIB grammar. The utility of anti-patterns has been explored for search-based program repair [45], but not for semantics-based counterparts, partially because it is difficult to integrate additional such measures directly in the constraint-based synthesis approach [45].

4 EVALUATION

This section describes our comparison between S3 and state-of-the-art semantics-based program repair techniques. We describe experimental setup and research questions in Section 4.1; answer those research questions in Sections 4.2–4.3; and present discussion, limitations, and threats in Section 4.4.

4.1 Experimental setup

We ran all experiments on a Intel Corei5 machine with 4 cores and 8GB of RAM.

Baseline approaches and settings. We compare S3 to Angelix [36], Enumerative [3], and CVC4 [41]. Angelix offers its specification inference engine and synthesis engine in separate code packages. Although the specification inference engines behind Angelix and S3 work on C and Java programs, respectively, Angelix's synthesis engine takes as input example-based specifications like the synthesis engine of S3. Thus, to enable comparisons between S3 and Angelix, we instruct S3's inference engine to generate the same type of specifications that Angelix's synthesis engine uses, and instruct both S3's and Angelix's synthesis engines to synthesize the repair based on the same provided specifications. Enumerative [3] and CVC4 [41] are state-of-the-art Syntax-Guided Synthesis (SyGuS) engines which both take input in the form of SYNTH-LIB scripts, like S3.[7] This allows straightforward comparison between the tools.

For *single-line patches*, we run a repair synthesis tool on each buggy location of each program in parallel, and stop once a repair is found. The timeout for synthesis task is set to three minutes each. For *multi-line-patches*, we implement the approach described bellow.

Angelix tackles patches involving multiple lines [36] by grouping multiple buggy locations, and synthesizing repairs for several locations at once. Angelix clusters buggy locations into groups of a user-specified size by either locality or suspiciousness score produced by fault localization. We reimplemented this feature, following Angelix's source code.[8] Angelix's synthesis engine are run on these specifications.

We implemented our own strategy to tackle *multi-line patches* for S3, *Enumerative*, and *CVC4*. Each buggy location is repaired separately, after which patches for certain locations are grouped. Given a test suite T, and patches $\{P_i\}$ generated by a repair synthesis tool for location i. Assuming each patch $p \in P_i$ leads the program to pass a set of tests $T_i \subset T$, we iterate through all patches and combine those that have $\cup T_i = T$. The intuition is that combining these patches may render the whole test suite T to pass, which we then verify dynamically.

[7] We refer interested readers to [1] and http://www.sygus.org/ for a full comparison between SyGuS engines

[8] https://github.com/mechtaev/angelix. The implementation for this feature in Angelix's source is approximately 70 lines of Python code.

Table 1: Top 5 largest programs that S3 can correctly patch. Math refers to the Apache Commons Math library

	Closure	OrientDB	Math	Molgenis	Heritrix
KLoc	237	203	175	54	48

Datasets. We consider two datasets of buggy programs:

- **Small programs associated with high coverage test suites.** We experiment with 52 Java bugs in the *smallest* subject programs of the IntroClass program repair benchmark [30] translated to Java [10]. The programs are student-written homework assignment from an introductory programming class; the goal of the programs is to find the smallest number between four integer numbers. Although the programs are small, they feature possibly complicated fixes involving changes in multiple *if-then-else* structures. We include only syntactically distinct programs. We focus on *smallest* because it only includes integer- and boolean-related fixes. Neither Angelix nor our framework can yet handle, e.g., floating point numbers or strings, primarily due to the limited capability of the constraint solving techniques used in symbolic execution.

  A key benefit of focusing on these small programs is that the problems in IntroClass are associated with two independent, high-quality test suites. We use one test suite to guide the search for a repair and the other to assess produced patch quality. We further augment the dataset by using Symbolic PathFinder [38] to generate additional tests. We do this by manually adding correctness specifications such as logical assertions, on the buggy programs, and use SPF to generate test inputs that expose bugs, e.g., assertion violations. This results in 16 additional tests.
- **Large real-world programs.** Our second dataset consists of 100 large real-world Java bugs from 62subject programs, featuring ground truth bug fixes submitted by developers. Our dataset only includes bugs with patches that change fewer than five lines of code. This simplifies quality and correctness assessment of machine-generated patches, which is especially important because real-world test cases can be incomplete or weak specifications of desired behavior [40, 43].

  We build our dataset based on a previously-proposed bug fix history dataset [28], which originally consists of around 3000 likely bug-fixing commits of fewer than five lines of code collected from GitHub. To further ensure that the collected commits are actually bug fixes, we randomly sampled 500 commits, and manually checked them to ensure that the commits compile and that the program test cases expose bugs pre-commit (as compared to post-commit test behavior). We treat tests that fail in the before-patched version but pass in the patched version as the failing tests addressed by the bug fixing commit. Since this process is time consuming, we stopped once we found 100 bugs from 62 programs. Table 1 shows the top five largest programs for which S3 can correctly patch bugs. "KLoc" depicts the number of lines of Java code in each project.

Table 2: Repair tool performance on 52 IntroClass bugs.

				Angelix			
	S3	Enum	CVC4	1	2	3	4
Produced	22	13	13	17	18	17	20
Pass all held-out tests	22	1	1	3	7	4	4
% Overfit	0%	92%	92%	82%	61%	76%	80%

```
1  if ((a < b) && (a < c) && (a < d)) {
2     // By S3: ((a <= b) && (a <= c) && (a < d))
3     // By Angelix: (a <= c) && (a < d)
4     System.out.println(a);
5  } else if ((b < a) && (b < c) && (b < d)) {
6     // By S3: (b <= a) && (b <= c) && (b < d)
7     // By Angelix: no change
8     System.out.println(b);
9  } else if ((c < a) && (c < b) && (c < d)) {
10    // By S3: (c <= a) && (c <= b) && (c < d)
11    // By Angelix: (c.value < d.value)
12    System.out.println(c);
13 } else {
14    System.out.println(d);
15 }
```

Figure 6: A bug in a *smallest* program correctly fixed exclusively by S3. We show the patches from S3 and Angelix.

Research questions and metrics. Our core metric is the number of buggy programs that a tool correctly patches. Fully assessing repair quality and correctness is an open problem in program repair research, and thus we approximate in several ways. For the IntroClass bugs, we designate a patch *correct* if it passes all held-out test cases, described above. We divide the SPF-generated tests randomly, using half to augment the tests used to repair and the other half to augment the held-out tests. For the real-world bugs, a patch is deemed *correct* if it is syntactically identical to the developer-produced patch. We also manually inspect all the results (produced by all repair tools) as a sanity check. In our inspection, if it is possible for a machine-generated patch to be converted into the corresponding developer's patch via basic transformations, we also consider it as correct. These patches are the minority in our evaluation; we separate these in our results and present the patches in prose. We report *overfitting rate*, or the percentage of produced patches that are incorrect, for each tool (lower is better); and *expressive power* in terms of the unique buggy programs each tool correctly patches. Our two research questions are then divided by dataset:

RQ1. How does each tool perform on the dataset of small programs associated with high coverage test cases, in terms of correct patches generated, overfitting rate, and expressive power?

RQ2. How does each tool perform on the dataset of real-world programs, in terms of correct patches generated, overfitting rate, and expressive power?

4.2 Performance on IntroClass

Table 2 shows the results of each repair synthesis tool on 52 bugs from the IntroClass dataset. The "Produced" column shows the total number of patches that each tool generated that pass the provided test cases, while the "Pass held-out tests" shows the number of produced patches that generalize to pass all held-out evaluation tests (and that we thus consider correct). "% Overfit" shows the percentage of produced patches that do not generalize to the held-out tests (lower is better). Note that Angelix's multi-line patch facility is driven by two parameters: number of buggy locations in a group (1–4), and the criterion used to group them (either by locality or suspiciousness score). These results are based on score-based grouping, which uniformly outperformed the alternative in our experiments (results not shown). When the group size is set to 1, we allow Angelix to try our own multi-line patch strategy, in case single-line repair is unsuccessful.

Table 3: Repair tool performance on 100 real-world bugs.

	S3	$S3_{syn}$	$S3_{sem}$	Enum	CVC4	Angelix
Produced	20	15	12	13	12	13
Syntax match	16	11	7	5	4	4
Manual	4	1	4	1	1	2
Overfit, Syn	20%	27%	42%	62%	67%	69%
Overfit, Both	0%	20%	8%	54%	58%	54%

Table 2 shows that S3 substantially outperforms the baselines, generating significantly more patches, *all* of which generalize to the held out test cases. The degree to which Angelix patches overfit varied by lines considered, ranging from a minimum of 61% to a maximum of 82%. Enumerative and CVC4 perform comparably, with a very high percentage of overfitting patches. S3 generates correct patches for all the bugs for which Angelix, Enumerative, and CVC4 can fix. S3 also generated almost exclusively multi-line patches (with one exception).

We speculate that the underlying synthesis techniques are the primary source of the baselines' weak performance. Enumerative enumerates expressions in increasing size, while CVC4 uses unsatisfiability (unsat) cores to synthesize solutions; neither rank candidate solutions, but instead conservatively return the first satisfying solution identified. Angelix encodes a simple patch minimality preference criteria in constraints suitable for PartialMax SMT. However, in these experiments, we observed that Angelix frequently generated patches that are quite different from the original buggy expressions (typically much smaller in size). These results and observations suggest that S3's combination of a customizable search space, an appropriately-managed expression-size-wise search strategy, and numerous ranking functions, all contribute to its successful generation of generalizable patches.

Figure 6 shows an example of a bug that S3 patches correctly but to which the baselines overfit. For brevity, we only show patches from S3 and Angelix. This code snippet requires a multi-line patch to multiple if-conditions. We show the replacement if-expressions from S3 and Angelix in the code comments. From the first if-condition, the Angelix fix is already incorrect, as it fails to capture the necessary relationship between variables a and b. The condition from S3 shares the structure of the original buggy expression, capturing the relationships between all variables. Producing this patch is likely assisted by S3's expression-size-wise enumerative search, which starts from the size of the original buggy expressions.

4.3 Performance on real-world programs

Table 3 shows the results of applying each considered repair tool on 100 real-world bugs from our second dataset. The first row

```
...First bug...
- if (Character.isDigit(next)// Buggy if-condition
+ if (Character.isDigit(next) || next == '.') // fix by developer
+ if ((46 == next) || Character.isDigit(next)) // fix by S3

...Second bug...
- return (csvBuffer.getMark() >= (bufferIndex - 1))// fix buggy expression
+ return (bufferIndex) < (csvBuffer.getMark() + 1)// fix by developer
+ return (csvBuffer.getMark() > (bufferIndex - 1))// fix by S3

...Third bug...
- while (newLength > offset)// fix buggy expression
+ while (newLength < offset)// fix by developer
+ while (offset > newLength)// fix by S3

...Fourth bug...
if(this.runningState != STATE_RUNNING && this.runningState !=
      STATE_SUSPENDED) {
    throw new IllegalStateException("...");
}
-       stopTime = System.currentTimeMillis();
+       if(this.runningState == STATE_RUNNING) { // fix by developer
+       if(this.runningState != STATE_SUSPENDED) // fix by S3
+           stopTime = System.currentTimeMillis();
+       }
```

Figure 7: Bugs for which S3 generates patches that are not syntactically identical but semantically equivalent to the developer fixes.

shows the total number of bugs for which each tool generated a patch. Because we lack second independent test suites for these programs, we use a direct syntactic match to the developer patch to define correctness (row "Syntax match"). We additionally found, via manual inspection, a small number of additional patches that appear semantically identical to the developer patches; we describe these patches for S3 below. The last two rows show the percentage of produced patches that fail to generalize to capture the developer-written patch, as judged via strict syntactic match ("Overfit, Syn") or via both syntactic match and manual inspection ("Overfit, Both").

S3 again substantially outperforms the baseline techniques, generating correct patches for many more programs. Only 4 of the 20 S3 patches fail to strictly syntactically match the developer fixes. Although manual author inspection, is an inadequate mechanism for rigorously assessing patch quality, simple syntactic transformation rules can convert these patches to their developer equivalents; we separate these out in Figure 7.

In terms of overfitting, only 20% of S3's patches fail to generalize when judged by perfect syntactic fidelity; when manual inspection is considered, none of the patches overfit. For Angelix, Enumerative, and CVC4, 54%, 58%, and 54% of the produced patches overfit, respectively.

In these experiments, we also evaluate the relative contribution of S3's syntactic versus semantic feature sets for ranking — $\mathbf{S3_{syn}}$ and $\mathbf{S3_{sem}}$ in the table, respectively. When only either syntactic or semantic features are used to rank the solution space, the performances of S3 varies. $\mathbf{S3_{syn}}$ and $\mathbf{S3_{sem}}$ generate fewer correct patches, with slightly higher overfitting rates, suggesting that both kinds of features are beneficial for S3's performance.

All programs that are correctly fixed by other tools are also fixed by S3. We note that the number of correctly-fixed bugs by the three baselines can be increased (to 9 bugs) if we combine all bugs correctly repaired by them. This combination is, however, still inferior to S3's performance.

The first bug in Figure 7 is an example of a bug that S3 fixes correctly, while the others do not. Enumerative and CVC4 generate the same fix with each other, that does not ultimately pass all tests (both synthesize `(0 == 0)` to replace the if condition); Angelix generates no fix for this bug. S3's fix is not syntactically identical but it is semantically equivalent to the developer's fix. This can be demonstrating by transforming S3's patch using basic transformation rules, e.g., swapping both left and right hand sides of the "||" operator, and converting the integer 46 to the character ".". The fix generated by Enumerative and CVC4, on the other hand, cannot be transformed to the developer's fix. We note that the incorrect fix generated by Enumerative and CVC4 is largely destructive, since it converts the branch condition to always evaluate to `true`. This kind of destructive fix can be prevented in S3 via the anti-patterns feature, as described in Section 3.2.3. In general, S3 generates more correct patches than the other approaches, judged via both syntactic fidelity to the developer fix and via fidelity with respect to basic syntactic transformations.

4.4 Discussion, Limitations, and Threats

Discussion and Limitations. Semantics-based repair in general exclusively modifies expressions in conditions or on the right-hand side of assignments. Additionally, such techniques can only synthesize or reason about replacement code including boolean or integer types. Our experience suggests that these limitations are the primary reasons for unrepaired bugs in our experiments. Some bugs require large changes to semantic or control-flow structure (e.g., a change from `if(...){A};if(...){B}` to `if(...){A} else if(...){B}`), the insertion of new statements, or manipulation of variables of types that existing constraint solving technology cannot handle. Resolving these challenges remains future work, and can progress apace with progress in the synthesis domain. However, it is noteworthy that semantics-based repair techniques are reasonably expressive despite these limitations.

Threats to validity. Our results may not generalize to other subject programs beyond those upon which our experiments were conducted. We mitigate this risk by evaluating our solution on 100 real bugs from many real-world programs. The size of this bug set is commensurate with those used to evaluate prior automated repair techniques, e.g., [29]. Another threat to the validity of our results is our reimplementation of the multi-line patch feature of Angelix (Section 4.1). However, we note this feature is simple, and only takes around 70 lines of Python code, and that we used the existing released implementation as reference.

Finally, we seek to assess the quality of the produced patches, in terms of the degree to which they overfit to the provided test cases (or, by contrast, generalize beyond them). Patch quality, especially in an automated repair context, is an unsolved research problem. We assess patch quality using several objective and established measures. We use independent test suites, when possible, to quantify overfitting (an established methodology [30, 43]). For real-world programs, we use syntactic fidelity to the developer patches as the gold standard for correctness. Bugs may often be patched multiple ways, and thus this standard is likely stricter than correctness

truly requires. We also manually inspect all produced patches, a process subject to bias but important to safeguard against mistakes. We present a number of these patches in this paper, and publicly release all the results, experimental data, and our code for open investigations.[9]

5 RELATED WORK

Program repair. General program repair techniques can typically be divided into two main branches: heuristic- and semantics-based repair. Heuristics-based repair includes techniques like GenProg [29, 31, 50], which heuristically searches for repairs via genetic programming algorithm. RSRepair [39] and AE [49] replace the search strategy in GenProg by random and adaptive search strategies, respectively. These techniques, despite scaling well, have been shown to produce patches that overfit to the provided test suites [40, 43]. PAR [20] generates repairs based on repair templates manually learned from human written patches. More recently, Prophet [34] and HDRepair [28] also use heuristic search to generate patches, augmented with mined repair models from historical data to rank patches, preferring those that match frequent human fix patterns. Tan *et al.* propose anti-patterns to prevent heuristic tools from generating trivial repairs [45]. ACS [51] targets if-condition defects by using fix templates (rules) to generate patches. It then leverages document analysis (such as on `javadoc` comments) as an additional criterion to rank patches.

Semantics-based repair techniques, such as SemFix [37], DirectFix [35], and Angelix [36], use symbolic execution and program synthesis to synthesize repairs. Such techniques, however, either do not have a notion of patch ranking or only include simple ranking criteria such as syntactic structural differences. As such, semantics-based repair approaches can also produce overfitting patches [27], motivating stronger techniques that can generalize beyond weak specifications inferred from tests. Other semantics-based techniques include SPR [32], which targets defects in if-conditions; SPR can also produce trivial or functionality-deleting repairs [36]. Nopol [52] works in a similar spirit to SPR and SemFix, targeting if-condition defects using SMT-based synthesis. Qlose [8] uses program execution traces as an additional criteria to rank patches, and encode program repair problem into a program synthesis tool namely SKETCH [44]. SearchRepair [19] lies between heuristic- and semantic-based repair, using semantic search as its underlying mutation approach to produce higher-granularity, high-quality patches. However, it does not yet scale as well as other approaches.

Our technique, S3, belongs to the semantics-based family, and thus, is different in kind from the heuristic techniques. S3 can target more bug types than SPR, Nopol, and ACS (which focus on if condition-based defects) including incorrect assignments, if- and loop-conditions, and expressions in return statements. Unlike ACS, S3 does not use explicit fix templates or document analysis to generate or rank patches; integrating such approaches in ranking especially is a possible avenue for future work. S3 is more scalable compared to various semantics-based counterparts such as SemFix, DirectFix, Qlose, and SearchRepair. S3 also allows the inclusion of a variety of ranking features beyond syntactic structural differences considered in prior work. Indeed, S3's ability to incorporate new ranking criteria is an important novelty, overcoming a known challenge in SMT-based synthesis [14, 45].

Program synthesis. Generally, techniques in this area include inductive (example-based) synthesis and deductive (logical reasoning based) synthesis. S3 belongs to the inductive family. FlashFill [13] synthesizes programs that work on string domain. FlashExtract [23] synthesize programs that automate the data extraction process. Singh *et al.* use programming by examples (PBE) to automatically transform spreadsheet data types [42]. FlashNormalize [21] automatically normalizes texts using PBE. Refazer uses a PBE-based approach to automatically learn program transformations. NoFAQ synthesizes command repairs from input-output examples [9]. Programming via sketching [44] uses a sketch as partial specifications, and search for an implementation that satisfies the specification; we use a similar idea in the DSL that uses a starting sketch to help rank candidate solutions.

6 CONCLUSIONS

We proposed S3, a new repair synthesis system that is able to generate high-quality, general patches for bugs in real programs. S3 consists of two main phases, which serve to: (1) Automatically extract examples that serve as a specification of correct behavior, using dynamic symbolic execution on provided test cases, and (2) Use a synthesis procedure inspired by the programming-by-examples methodology to synthesize general patches. The efficiency and effectiveness of the synthesis procedure is enabled by our novel designs of three main parts, including a domain-specific language, which we extend from SYNTH-LIB [3]; an expression-size-wise enumerative search; and syntax- and semantic-guided ranking features that help rank the highest quality solutions highest in the solution space. Our results showed that S3 generates many more high-quality bug fixes than even the best performing baseline from prior work.

Beyond these results, our approach opens a number of opportunities for future repair synthesis techniques. The specifications, in the form of input-output examples, can be strengthened with specifications inferred by specification mining and other inference techniques [11, 22], possibly enabling integration of inductive and deductive synthesis for a more expressive overall system. Our framework's flexible design allows more features to be investigated and easily integrated into our ranking technique, such as, for example frequent fix patterns mined from human written patches [28]. Our dataset can also be extended, and used to evaluate many more repair systems. We plan to extend the SYNTH-LIB grammar to represent more tasks in the program repair domain, e.g., nonlinear computations on the integer domain. Finally, machine learning might be useful in automatically classifying bug types [47], to more effectively deal with different kinds of defects automatically.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant CCF-1563797. Duc-Hiep Chu was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

[9] https://xuanbachle.github.io/semanticsrepair/

REFERENCES

[1] 2016. Syntax-guided Synthesis. (2016). http://www.sygus.org/

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. 89–98.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* (2015).

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. *Scaling Enumerative Program Synthesis via Divide and Conquer*. Technical Report. University of Pennsylvania.

[5] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. *Reversible Debugging Software*. Technical Report. University of Cambridge, Judge Business School.

[6] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *International Conference on Software Engineering (ICSE'11)*. 121–130.

[7] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective symbolic execution. In *Workshop on Hot Topics in System Dependability (HotDep)*.

[8] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)*. Springer, 383–401.

[9] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017 (to appear). NoFAQ: Synthesizing command repairs from examples. In *Joint Conference on European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE '17)*.

[10] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. Technical Report. Universite Lille 1. https://hal.archives-ouvertes.fr/hal-01272126/document

[11] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

[12] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering (ASE'14)*. 313–324.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[14] Sumit Gulwani, Javier Esparza, Orna Grumberg, and Salomon Sickert. 2016. Programming by Examples (and its applications in Data Wrangling). *Verification and Synthesis of Correct and Secure Systems* (2016).

[15] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Working Conference on Mining Software Repositories (MSR)*. 121–130.

[16] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *International Conference on Software Engineering (ICSE)*. Cape Town, South Africa, 215–224. DOI : http://dx.doi.org/10.1145/1806799.1806833

[17] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *International conference on Software Engineering (ICSE)*. IEEE, 96–105.

[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA '14)*. 437–440.

[19] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering (ASE)*. 295–306.

[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE '13)*. 802–811.

[21] Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization.. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 776–783.

[22] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (t). In *International Conference on Automated Software Engineering (ASE)*. IEEE, 115–125.

[23] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 542–553.

[24] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017 (to appear). JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA'17)*.

[25] Xuan Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing Automated Program Repair with Deductive Verification. In *International Conference on Software Maintenance and Evolution (ICSME)*. 428–432.

[26] Xuan-Bach D Le, Tien-Duy B Le, and David Lo. 2015. Should fixing these failures be delegated to automated program repair?. In *International Symposium on Software Reliability Engineering (ISSRE)*. 427–437.

[27] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. Empirical study on synthesis engines for semantics-based program repair. In *International Conference on Software Maintenance and Evolution (ICSME'16)*. 423–427.

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

[29] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE'12)*. 3–13.

[30] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *Transactions on Software Engineering (TSE)* 41, 12 (Dec. 2015), 1236–1256.

[31] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[32] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 166–178.

[33] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering (ICSE)*. ACM, 702–713.

[34] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL)*. 298–312.

[35] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 448–458.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*. IEEE, 691–701.

[37] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 772–781.

[38] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

[39] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *International Conference on Software Engineering (ICSE)*. ACM, 254–265.

[40] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 24–36.

[41] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *International Conference on Computer Aided Verification (CAV)*. 198–216.

[42] Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 343–356.

[43] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. 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, 532–543.

[44] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*. ACM, 281–294.

[45] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *International Symposium on Foundations of Software Engineering*. ACM, 727–738.

[46] G. Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *Planning Report, NIST* (2002).

[47] Ferdian Thung, Xuan-Bach D Le, and David Lo. 2015. Active semi-supervised defect categorization. In *International Conference on Program Comprehension*. IEEE Press, 60–70.

[48] Willem Visser. 2016. What makes killing a mutant hard. In *International Conference on Automated Software Engineering (ASE)*. ACM, 39–44.

[49] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering (ASE)*. 356–366.

[50] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*. IEEE, 364–374.

[51] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 416–426.

[52] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *Transactions on Software Engineering* (2016).