FrAngel: Component-Based Synthesis with Control Structures

KENSEN SHI, Stanford University, USA
JACOB STEINHARDT, Stanford University, USA
PERCY LIANG, Stanford University, USA

In component-based program synthesis, the synthesizer generates a program given a library of components (functions). Existing component-based synthesizers have difficulty synthesizing loops and other control structures, and they often require formal specifications of the components, which can be expensive to generate. We present FrAngel, a new approach to component-based synthesis that can synthesize short Java functions with control structures when given a desired signature, a set of input-output examples, and a collection of libraries (without formal specifications). FrAngel aims to discover programs with many distinct behaviors by combining two main ideas. First, it *mines code fragments* from partially-successful programs that only pass some of the examples. These extracted fragments are often useful for synthesis due to a property that we call *special-case similarity*. Second, FrAngel uses *angelic conditions* as placeholders for control structure conditions and optimistically evaluates the resulting program sketches. Angelic conditions decompose the synthesis process: FrAngel first finds promising partial programs and later fills in their missing conditions. We demonstrate that FrAngel can synthesize a variety of interesting programs with combinations of control structures within seconds, significantly outperforming prior state-of-the-art.

CCS Concepts: • **Software and its engineering → Programming by example**; *Automatic programming*;

Additional Key Words and Phrases: program synthesis, component-based synthesis, control structures, angelic execution

ACM Reference Format:
Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (January 2019), 30 pages. https://doi.org/10.1145/3290386

1 INTRODUCTION

Programmers often browse software libraries to identify useful functions and combine such functions with loops and conditionals to achieve some desired functionality. Using large libraries is not easy—even experienced programmers can spend hours identifying the few necessary functions in a library [Mandelin et al. 2005]. *Component-based program synthesis* aims to ease this aspect of programming by synthesizing programs using a library of *components* (existing functions).

We focus on synthesis *from examples*, where the user provides input-output examples that the synthesized program must satisfy. Although the desired program behavior can be specified with logical constraints [Gulwani et al. 2011a; Srivastava et al. 2010] or executables [Heule et al. 2015;

Authors' addresses: Kensen Shi, kensens@cs.stanford.edu, Department of Computer Science, Stanford University, California, USA; Jacob Steinhardt, jsteinhardt@cs.stanford.edu, Department of Computer Science, Stanford University, California, USA; Percy Liang, pliang@cs.stanford.edu, Department of Computer Science, Stanford University, California, USA.

2475-1421/2019/1-ART73
https://doi.org/10.1145/3290386

Jha et al. 2010], I/O examples are appealing because they provide an intuitive user experience for programmers (after all, most unit tests are in fact I/O examples) and even end-users with no programming knowledge [Gulwani 2011].

Many works in component-based synthesis use domain-specific knowledge to solve specific classes of problems such as string manipulation [Gulwani 2011; Menon et al. 2013], geometric constructions [Gulwani et al. 2011b], and transformations of data [Feng et al. 2017a; Feser et al. 2015; Harris and Gulwani 2011; Yaghmazadeh et al. 2016]. Similarly, Perelman et al. [2014] describe a general framework that accepts an expert-written domain-specific language (DSL) as input. On the other hand, Gulwani et al. [2011a]; Jha et al. [2010] describe approaches that are domain-agnostic but require formal specifications of the components. However, few libraries have existing specifications, and writing a specification for every library component can be prohibitively time-consuming—such approaches have only been demonstrated in domains with few components and simple specifications (e.g., bit-manipulation algorithms [Jha et al. 2010]).

One goal of this work is to perform synthesis from examples without requiring specifications of components or domain-specific knowledge as in the work described above (this regime was previously considered by SyPet [Feng et al. 2017b]). To this end, we avoid creating a DSL or requiring one as input. Furthermore, without specifications of components, the constraint-solving techniques used in Feng et al. [2017a]; Gulwani et al. [2011a]; Jha et al. [2010] become infeasible. Like SyPet, our work relies on blackbox execution rather than reasoning about formal semantics.

Another goal of our work is to perform synthesis *with control structures*, i.e., loops and conditionals. Loops in particular have been a common source of difficulty in component-based synthesis [Feng et al. 2017b; Gulwani et al. 2011a; Jha et al. 2010; Menon et al. 2013]. Some approaches use specialized loops in a DSL [Gulwani 2011; Perelman et al. 2014] or looping components (e.g., `map` and `filter` operators) [Feng et al. 2017a; Feser et al. 2015; Harris and Gulwani 2011; Yaghmazadeh et al. 2016] to mitigate this issue in specific domains. Yet, relying only on such components is insufficient for general-purpose synthesis, evidenced by the fact that programmers still use loops in everyday code.

We present a system called FrAngel to tackle domain-agnostic component-based synthesis from examples, using control structures and libraries without specifications. The key insight is that *progress in a synthesis search is primarily made by finding programs with distinct meaningful behaviors (i.e., program semantics).* After all, the goal of synthesis is not to find one particular program, but rather any program with the desired behavior, possibly among many acceptable solutions. Furthermore, a meaningful program with nontrivial behavior is composed of simpler "building blocks" that are also meaningful in related ways toward the overall functionality. By identifying and combining programs with distinct behaviors, FrAngel can rapidly discover increasingly-complex behaviors that are likely relevant to the synthesis task.

Checking program equivalence is undecidable, so FrAngel approximately characterizes a program's semantics by observing results on a set of test cases (namely, the I/O examples): a program passing a subset of tests is assumed to "contain" the behavior described by those tests. Additionally, if at least one test is passed, then the behavior is assumed to be relevant to the synthesis task. In this way, every nonempty subset of test cases describes some relevant behavior, and a program has (at least) that behavior if it passes (at least) those test cases.

At its core, FrAngel is an adaptive search that randomly samples programs and evaluates them on the I/O examples. It "remembers" partially-successful programs, specifically the simplest implementation found so far that has each relevant behavior. Then, two techniques are used to guide and factorize the search for programs with new behaviors, respectively:

(1) FrAngel *mines code fragments* to learn from the remembered programs. These code fragments can be used in future random programs, effectively biasing the search toward the known

relevant behaviors. Fragments can be combined and modified to produce new behaviors. We also identify and discuss a common property of programs called *special-case similarity*, hypothesizing that code fragments from partially-successful programs often appear in the desired program, which can justify why this learning approach works well in practice.

(2) FrAngel also uses *angelic conditions* to decompose the synthesis process into two steps: first find a promising program structure, and then determine how to correctly direct its control flow. FrAngel initially generates programs with unspecified control structure conditions ("angelic conditions," based on ideas in Bodík et al. [2010]) and optimistically evaluates these programs by trying many different control flows. If FrAngel finds an "angelic program" with control flows that lead to passing many test cases, it then attempts to resolve, or "fill in," the angelic conditions, resulting in a concrete program. This strategy helps FrAngel combine known behaviors with control structures to discover more complex behaviors.

FrAngel synthesizes Java functions given a collection of libraries, the desired function signature, and a set of input-output examples. We created a benchmark suite including 90 tasks of widely varying difficulty, using a variety of libraries and control structure patterns. We also include the 30 tasks used to evaluate SyPet [Feng et al. 2017b], the previous approach most similar to ours. While SyPet solves 28% of the 120 tasks within 30 minutes, FrAngel solves 94% within 30 minutes and 47% within 10 seconds. FrAngel scales to hundreds (sometimes thousands) of components and can solve tasks requiring up to three control structures with different nesting patterns.

Contributions.

- We present FrAngel,[1] a system for domain-agnostic and component-based synthesis from examples (Section 3).
- We present a method to *mine code fragments* that are likely to be useful for a given task (Section 4). We also discuss the *special-case similarity* property of programs, and argue why this property leads to the effective mining of fragments (Section 4.1).
- We describe *angelic conditions*, which factorize the synthesis process (Section 5). Programs with angelic conditions are evaluated optimistically with *angelic execution* (Section 5.2).
- We demonstrate that FrAngel can generate interesting programs using multiple control structures along with libraries containing hundreds of components within seconds (Section 6).

2 MOTIVATING EXAMPLE

Suppose we wish to synthesize the target program `getRange` (Figure 1),[2] which returns a list of integers from the first argument (inclusive) to the second argument (exclusive). Figure 1 also lists the test cases (examples), e.g., for the inputs `(10, 12)`, `getRange` should return the list `[10, 11]`.

One might try to run randomly-generated programs on the test cases, but this strategy alone is infeasible—generating the entire target program by chance is incredibly unlikely. In fact, for this task, our synthesizer FrAngel considers a search space of roughly 30K programs of size 5 (nodes in the abstract syntax tree, or AST), 800K programs of size 6, and over 10M of size 7. Figure 1 is actually FrAngel's solution, which has size 14. In our experiments, a naïve random search was never able to solve this task.

Mining fragments. However, it *is* feasible to randomly generate simpler programs that produce the correct result for some of the test cases—Figure 2 shows a few examples. For instance, returning an empty list will work for all test cases where `start >= end` (e.g., the first and second tests in Figure 1). Can we learn from these partial successes to find other relevant behaviors? Indeed,

[1] The code, benchmarks, and results are available at https://www.github.com/kensens/FrAngel [Shi et al. 2018].
[2] The loop body could simply be `list.add(start + i);`, but FrAngel currently does not handle autoboxing or unboxing.

```
// #1. (10, 9)  -> []      #4. (10, 12) -> [10, 11]
// #2. (10, 10) -> []      #5. (-2, 2)  -> [-2, -1, 0, 1]
// #3. (10, 11) -> [10]
List<Integer> getRange(int start, int end) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 0; start + i < end; i++)
        list.add(Integer.valueOf(start + i));
    return list;
}
```

Fig. 1. An example target program, `getRange`, in Java, with five test cases (examples) in comments.

```
// Works for tests #1 and #2
List<Integer> getRange(int start, int end) {
    return new ArrayList<Integer>();
}

// Works for test #3
List<Integer> getRange(int start, int end) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(Integer.valueOf(start));
    return list;
}

// Works for test #4
List<Integer> getRange(int start, int end) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(Integer.valueOf(start));
    list.add(Integer.valueOf(start + 1));
    return list;
}
```

Fig. 2. Programs that pass some test cases. Our algorithm extracts code fragments from such programs.

notice that these simple programs contain important operations used by the target program, such as `new ArrayList<Integer>()` and `list.add(◦)`. This is no coincidence; in many situations, simple programs that pass some test cases will be similar to the target program. We call this phenomenon *special-case similarity*, discussed in depth in Section 4.1. This motivates the idea of *mining fragments*: we extract code fragments from programs that pass some of the test cases. Newly-generated programs can then use the extracted fragments or modifications of them. For instance, one fragment extracted from the second program in Figure 2 is `list.add(Integer.valueOf(start))`. This fragment helps generate the third program, which uses the fragment twice (once with a modification).

Note that the partially-successful programs depend on the specific examples provided. If the test cases only contained ranges with three or more elements, then the programs in Figure 2 would not pass any test cases. Our approach is most effective if the examples include a variety of small cases or edge cases, as in Figure 1. This requirement is quite natural and is similar to what one might expect from well-constructed unit tests.

Although `list.add(Integer.valueOf(start))` is a useful extracted fragment, generating the full `getRange` program is still difficult. We would have to put the correct modification of the fragment in a loop (changing `start` to `start + i`) and simultaneously generate a correct loop condition.

Angelic conditions. This leads to our second main idea: instead of relying on control structure bodies and conditions to be correct simultaneously, we decompose the problem by allowing these events to occur in sequence. More specifically, we randomly generate programs with unspecified control structure conditions and then attempt to fill in the conditions if the program sketch seems promising. As a concrete example, the following program sketch is an *angelic program* because it uses an *angelic condition*, denoted `<ANGELIC>`, in the loop:

```
List<Integer> getRange(int start, int end) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 0; <ANGELIC>; i++)
        list.add(Integer.valueOf(start + i));
    return list;
}
```

To execute this angelic program, we allow the evaluation of the angelic condition to try different control flows (numbers of loop iterations) for each test case. For example, this program passes test #4 (i.e., `getRange(10, 12)` returns `[10, 11]`) by using 2 loop iterations, and in fact, each test case can be passed using an appropriate control flow. This observation hints that the angelic program could be a high-level description of the desired behavior.

We now *resolve* the angelic condition by replacing it with a Boolean expression that produces the desired control flow for all test cases. In this case the expression `start + i < end` successfully resolves the angelic condition, completing the synthesis task.

Despite the enormous search space considered, FrAngel is able to solve the `getRange` task in only 3.5 seconds on average by mining fragments and using angelic conditions.

3 THE FRANGEL ALGORITHM

We now describe the FrAngel algorithm. We formalize the setting below, summarize the algorithm in Section 3.1, and leave specific details to be discussed in Sections 3.2, 3.3, 4, and 5.

Figure 3 summarizes our problem setting. We focus on synthesizing Java programs, given the signature of the function to synthesize (the *target program*), a set of test cases (synonymous with *examples* in this paper), and a list of libraries (Java classes) from which we draw components (i.e., the classes' public methods, constructors, instance variables, and constants). Our goal is to find a program that is consistent with all examples. Since we only check the provided examples, the resulting program is not guaranteed to match the user's intent; we assume that the user will perform code review and/or further testing to verify correctness.

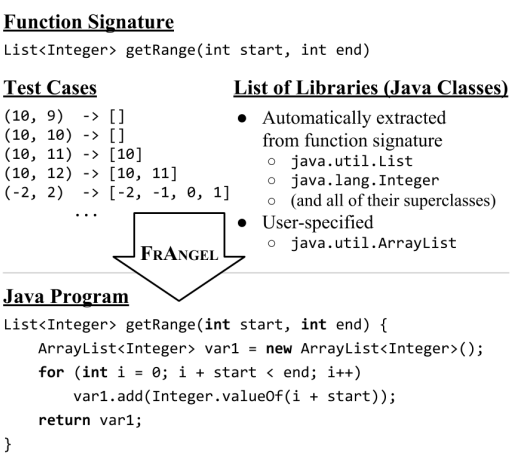

Fig. 3. An overview of the problem setting.

In this paper we use *program* to refer to any function implementation with the desired function signature. A *test case* (X, Y) contains all inputs X required for a single invocation of the target program, along with the corresponding outputs Y. If the target program changes some mutable inputs, then their new (altered) values are considered outputs. We say that a program P *passes* a test case (X, Y) if P produces the outputs Y when invoked with the inputs X.

Algorithm 1 FrAngel

Input: Target program signature S, set of libraries L, set of test cases C
Output: A program P that passes all tests in C

```
1:  procedure FrAngel(S, L, C)
2:      R ← ∅                                          ▷ A set of remembered programs
3:      F ← ∅                                          ▷ A set of fragments
4:      repeat until timeout
5:          A ← RandBoolean()                          ▷ Whether to generate angelic conditions
6:          P ← GenRandomProgram(S, L, F, A)           ▷ Step (1): randomly generate program
7:          T ← GetPassedTests(P, C)
8:          if T = ∅ then
9:              continue
10:         if P contains angelic conditions then      ▷ Step (2): resolve angelic conditions
11:             P ← ResolveAngelic(P, L, F, T, C)
12:             if failed to resolve conditions then
13:                 continue
14:             T ← GetPassedTests(P, C)
15:         P ← SimplifyQuick(P, T)
16:         T ← GetPassedTests(P, C)
17:         if P is the simplest program to pass T then   ▷ Step (3): mine fragments
18:             R ← RememberProgram(P, R)              ▷ Add P to R and remove worse programs
19:             F ← MineFragments(R)
20:         if T = C then
21:             return SimplifySlow(P, C)              ▷ Solution found
22:     return Failure                                 ▷ Timeout reached
```

Our implementation makes some simplifying assumptions: we ignore side effects (e.g., filesystem modifications), only using object equality or user-defined checks to compare program semantics, and we disallow test cases where the desired action is to throw an exception, although the algorithm in principle can handle this case.

FrAngel infers relevant libraries (Java classes) by automatically using all types in the target program's signature and any user-specified types, plus all of their supertypes and nested classes.

3.1 Algorithm Summary

The overall structure of the FrAngel algorithm is presented in Algorithm 1 and follows the intuition provided in Section 2. The core algorithm structure is quite simple, looping over three basic steps until a solution (i.e., a program passing all test cases) is found or a timeout is reached:

(1) Randomly generate a program P using previously-mined fragments and angelic conditions.
(2) Resolve angelic conditions in P if any, resulting in a program without angelic conditions.
(3) Mine fragments from the program if it is the simplest program so far to pass some subset of test cases. More precisely, for each nonempty subset of test cases, extract fragments from the simplest program so far that passes that subset of cases.

FrAngel also focuses on programs that show signs of (partial) success: FrAngel only proceeds to step (2) if P passes at least one test case, and step (3) is contingent on successfully resolving all angelic conditions in step (2). If these conditions are not met, then FrAngel immediately discards the program and returns to step (1), repeating the process with a new random program.

$$
\begin{array}{rcl}
v & ::= & \textit{any variable name in scope} \\
c & ::= & \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{0.0} \mid \texttt{1.0} \mid \texttt{2.0} \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \texttt{""} \mid \textit{any user-provided constant} \\
op & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{==} \mid \texttt{<} \mid \texttt{<=} \\
f & ::= & \texttt{(}e\texttt{).}\textit{method}\texttt{(}e^*\texttt{)} \mid \textit{ClassName}\texttt{.}\textit{staticmethod}\texttt{(}e^*\texttt{)} \\
e & ::= & v \mid c \mid \texttt{(}e\texttt{)}\ op\ \texttt{(}e\texttt{)} \mid \texttt{!(}e\texttt{)} \mid f \mid \texttt{(}e\texttt{)[}e\texttt{]} \mid \texttt{(}e\texttt{).}\textit{field} \mid \\
 & & \textit{ClassName}\texttt{.}\textit{field} \mid \texttt{new}\ \textit{ClassName}\texttt{(}e^*\texttt{)} \\
s & ::= & v\ \texttt{=}\ e\texttt{;} \mid \texttt{(}e\texttt{)[}e\texttt{] =}\ e\texttt{;} \mid f\texttt{;} \mid \texttt{if (}e\texttt{) \{}\ s^+\ \texttt{\}} \mid \\
 & & \texttt{for (int i = 0;}\ e\texttt{;}\ \texttt{i++) \{}\ s^+\ \texttt{\}} \mid \texttt{for (}\textit{ClassName}\ v\ \texttt{:}\ v\texttt{) \{}\ s^+\ \texttt{\}}
\end{array}
$$

Fig. 4. FrAngel grammar for expressions e and statements s. We use e^* to represent zero or more comma-separated expressions, and s^+ for one or more statements.

Note that FrAngel only generates angelic conditions for half of the programs. By generating non-angelic control structures, FrAngel can extract control structures as fragments without needing to resolve angelic conditions, which might not succeed for partially-correct programs.

Section 3.2 describes the random generation of programs in step (1), Section 5 describes angelic conditions and ResolveAngelic from step (2), and Section 4 discusses RememberProgram and MineFragments from step (3). Before mining fragments or returning a final program, FrAngel uses a simplification procedure to remove unnecessary or overly-verbose code, presented in Section 3.3.

3.2 Random Program Generation

A program is a function with the desired signature consisting of some local variable declarations[3] followed by zero or more statements and, if applicable, a return statement with an appropriately-typed expression. Our grammar for statements and expressions is shown in Figure 4 and specifies a reasonably large subset of Java. Beyond the restrictions in the grammar, we adhere to (a simplification of) Java's type rules and explicitly exclude methods such as `Object.wait()` and `Math.random()` that can stall execution, cause undesirable side-effects, or lead to nondeterministic programs.

Recall that FrAngel collects a set of previously-mined fragments (produced by MineFragments, detailed in Section 4) that can be used as building blocks, possibly with modifications, when generating new programs. A *fragment* is a complete subtree of a previous program's abstract syntax tree (AST), e.g., the code `return a + b.foo(c);` contains fragments including `b.foo(c)` and `c`.

We now describe how a program's statements and expressions are randomly generated, first discussing a simpler case without fragments, and then the case with fragments.

Procedure A (Basic Generation). To generate an expression or statement, we first choose a kind of expression (variable, constant, operator, or function call) or statement (assignment, function call, `if` statement, or `for` loop) uniformly at random. Further decisions, such as the specific operator or function to use, are made uniformly at random among options in the grammar that typecheck. Subexpressions and substatements are generated recursively.

This procedure also takes a *size* parameter that upper-bounds the size (number of AST nodes) of the resulting AST or subtree. The tree's root node must be chosen to adhere to the size constraint, taking into account the number of children it would require. Then, the node's remaining size is randomly partitioned among its children. The *size* parameter is primarily used to avoid generating infinitely-large trees, since otherwise this random generation procedure is a Galton-Watson process [Harris 2002]. In our experiments, programs are generated with size at most 40. This is a much larger limit than necessary, as most of our benchmarks are solved by programs smaller than size 20.

[3]The local variable's type is set to the program's return type half of the time and is chosen randomly among all available types otherwise. Local variables are initialized with appropriate constants, zero-argument constructors if available, or `null`.

Procedure B (Generation with Fragments). We now explain how FrAngel takes into account the mined code fragments. Empirically, correct programs often contain mined fragments (perhaps with different variable names) as exact copies or with modifications (we elaborate on this observation in Section 4). However, correct programs can also include code that is not similar to any mined fragment. To handle these cases, FrAngel can use an existing fragment as-is or with random parts modified, or generate code from scratch. This strategy biases the random program generation toward the mined fragments while maintaining the flexibility of random search.

Specifically, FrAngel uses the following procedure to generate an expression or statement X:

(1) With $\frac{1}{2}$ probability, we generate the root of X randomly from scratch using *Procedure A*, except that subtrees are generated recursively with *Procedure B*.
(2) Otherwise, we sample uniformly, from the set of mined fragments, an expression or statement X' of the same type as X (defaulting to the previous case if no such X' exists). Then,
   (a) We make the variable names in X' compatible with the surrounding program P by changing the names in X' to match existing variable names in P of the same type. If such names do not exist, we accommodate by first declaring new variables in P for the missing types.
   (b) With $\frac{1}{2}$ probability, we use the entire fragment X' as-is by setting $X = X'$.
   (c) Otherwise, we generate X to be a random modification of X'. More precisely, a rooted connected component of X will match X', while the rest is generated randomly without fragments. We start by making the root of X the same as the root of X'; then, as we walk down the tree, at each node we continue to make X match X' with probability $\frac{3}{4}$, and otherwise we generate that subtree of X randomly using *Procedure A*.

3.3 Program Simplification

FrAngel applies two simplification strategies (SimplifyQuick and SimplifySlow in Algorithm 1) to the mined fragments and the final solution. This eliminates unnecessary code, resulting in a slight speedup (by increasing the relevancy of mined fragments) and more natural final solutions.

Before mining fragments from a program P, FrAngel simplifies P with SimplifyQuick. Let T be the set of test cases originally passed by P, and for any AST node N, let $N_\top$ represent the AST subtree rooted at N. SimplifyQuick can be described as follows:

(1) For every node N in the AST of P:
   (a) Consider all of the following AST subtrees, which will act as "replacements" for $N_\top$:
      - The empty tree, if $N_\top$ is a statement in a block.
      - Single-node trees corresponding to all variables and constants in the grammar.
      - $D_\top$ for all descendant nodes D of N.
   (b) For each of the above "replacement" subtrees R:
      (i) Temporarily replace $N_\top$ with R, if doing so follows Java type and syntax rules, and if R is smaller than $N_\top$ (comparing first by the number of AST nodes and then by code length).
      (ii) If the new code passes all tests in T, then make the replacement permanent. Otherwise, undo the temporary replacement.
(2) Repeat until P cannot be simplified further (i.e., all remaining replacements are unsuccessful).

Before returning a final program, FrAngel uses a more extensive simplification strategy, SimplifySlow. This is the same as SimplifyQuick, except with the following modifications:

- Only statements, control structure conditions, and the return expression of P can be replaced. Hence, SimplifySlow focuses on larger-scale changes compared to SimplifyQuick.
- Replacement subtrees R are generated randomly.
- Instead of proceeding until no more replacements are possible, SimplifySlow runs for 10% of the total time spent before SimplifySlow.

- After the time limit is reached, SimplifySlow concludes with one run of SimplifyQuick.

Although the final program behaves identically on the test cases before and after SimplifySlow, simplification can greatly improve readability, e.g., by replacing `(str + "").equalsIgnoreCase(str)` with the equivalent `str != null`. SimplifySlow can also improve awkward expressions that might not generalize fully. For instance, if `n` is an `int`, then replacing `1 < n + 1` with `0 < n` can eliminate an integer overflow bug that might not be evident from executions on the given test cases. In this way, simplification can reduce the number of false positives produced by FrAngel (i.e., programs that pass all test cases but do not completely match the user's intent).

4 MINING FRAGMENTS

As we observed in the motivating example (Section 2), programs passing some of the tests often share code with the target program. FrAngel uses this observation by mining fragments from such programs and using those fragments when generating new programs. This biases the search toward previous partial successes and potentially new relevant behaviors, such as the target program.

Given a program P, a *fragment* is any complete subtree of P's abstract syntax tree (AST). For example, in the program `return a + b.foo(c)`, the fragments are `a + b.foo(c)`, `b.foo(c)`, `a`, `b`, and `c`. Neither `+` nor `b.foo(∘)` are fragments because they are not *complete* subtrees. However, when FrAngel uses fragments to generate new programs using *Procedure B* as described in Section 3.2, it will be able to generate code like `b.foo(∘)`, where ∘ is generated randomly from scratch.

In Algorithm 1, RememberProgram chooses which programs to "remember," and MineFragments extracts all fragments from all currently-remembered programs. A program is remembered if it is the simplest program encountered by FrAngel so far that passes some nonempty subset of test cases.[4] Simplicity is measured by the number of AST nodes and code length. Note that a program that is remembered at one point can later be "forgotten" when FrAngel finds an even-simpler program passing at least the same test cases. Hence, the set of remembered programs gradually improves as FrAngel runs—it grows in size and captures more behaviors as more distinct subsets of test cases are passed,[5] and the remembered programs can be replaced if better (simpler) versions are found. Also note that MineFragments returns a *set* of fragments, i.e., a fragment that appears multiple times in the remembered programs occurs only once in the set of mined fragments.

While the provided libraries might have hundreds of components, mining fragments allows FrAngel to focus on the ones that are likely to be useful for the current synthesis task. Furthermore, since FrAngel can mine fragments from code that was generated using previously-mined fragments, the fragments often grow larger by composition and better through random improvements.

4.1 Special-Case Similarity

Why should we expect the mined fragments to be useful for synthesizing the solution? To answer this, we describe a common property of programs called *special-case similarity* (SCS).

We begin with an analogy on the real line for intuition. In Figure 5, the target program P maps inputs to outputs, represented as a function (solid blue curve). Consider a set S of inputs.

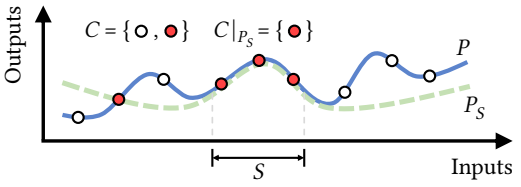

Fig. 5. Diagram illustrating *special-case similarity* and *exposure* using an analogy on the real line.

[4] The program can also pass other test cases, so a program can be the simplest for many subsets of tests simultaneously. In other words, a program is not remembered if there is a simpler program passing a superset of tests.

[5] Although there are exponentially many subsets of test cases, empirically the number of remembered programs is less than 100, even for tasks with 12 test cases.

Let P_S (dashed green curve) be the "simplest" function matching P for the inputs in S. If P_S is "similar" to P, then P is special-case similar (SCS) with respect to S, and we say that S is a *special case* with the corresponding *special-case program* P_S. That is, P is SCS if there are inputs (e.g., S) where the simplest solution (i.e., P_S) is similar to P.

In the realm of programs, we measure *simplicity* first by the number of AST nodes and then by the code length, but other reasonable metrics exist, e.g., preferring natural code as done by Maddison and Tarlow [2014]. We say that two programs are *similar* if there is nontrivial overlap, i.e., duplicate connected subgraphs of the AST, between their sets of fragments. For instance, the fragment `a + b.foo(1)` significantly overlaps with `bar(1 + b.foo(c))` since the 3-node portion `∘ + b.foo(∘)` is equal. However, it only has trivial overlap with `b.baz()` since only node `b` is duplicated.

Ideally, we want FrAngel to remember and mine fragments from special-case programs such as P_S. Compared to the target program P, special-case programs are easier to find because they are simpler and more likely to be generated. By mining fragments from remembered special-case programs, which by definition are similar to P, FrAngel guides its search toward P even without explicit knowledge of P, thus reaching a final solution much faster.

The following definition formalizes this intuition of SCS.

Definition 1: special-case similarity (SCS). *Assume a fixed grammar G over programs. Consider a program $P \in G$. For any nonempty set X of inputs to P, let $P_X \in G$ be the simplest program such that P and P_X have identical functionality when restricted to X. We define P to be* special-case similar *with respect to a nonempty set S of inputs (a "special case") if P_S (a "special-case program") is similar to P and $P_S \neq P$. More generally, we say that P is* special-case similar (SCS) *if there exists such a set S.*

Note that if P is SCS, then a special-case program P_S must be strictly simpler than P, and in most cases, P_S is actually a *simplification* of P. Also, P can be SCS with respect to multiple sets of inputs.

Our specific notion of similarity is based on the way that FrAngel makes use of fragments. As a general intuition, P_S is "similar enough" to P if the knowledge of P_S allows the synthesis algorithm to produce P faster or with higher probability. We also point out that the FrAngel algorithm itself does not use any notion of similarity, which is discussed here only for intuitive justification for fragment mining.

Exposure. Whether FrAngel remembers a particular special-case program P_S depends largely on the user-provided test cases C, denoted by circles with any fill color in Figure 5. Recall that FrAngel remembers the simplest program that passes each nonempty subset of test cases. Hence, as long as P_S is the simplest program solving some subset $C' \subseteq C$ with $C' \neq \emptyset$, then FrAngel is guaranteed to remember (and never forget) P_S once it is randomly generated as a candidate program. In this case, we say that S (i.e., the special case corresponding to P_S) is *exposed* by the test cases C.

It actually suffices to only consider the test cases in C that are consistent with P_S, denoted by circles with red fill in Figure 5. Let the notation $C|_{P_S}$ refer to that subset of C. Then, C exposes S if and only if $C|_{P_S} \neq \emptyset$ and P_S is the simplest program solving $C|_{P_S}$. To see this, note that if P_S is the simplest program for some nonempty $C' \subseteq C$, then all elements of C' are in C and are consistent with P_S, i.e., $C' \subseteq C|_{P_S}$. Hence, any program solving $C|_{P_S}$ must also solve C'. Since P_S is the simplest program for C', the simplest program for $C|_{P_S}$ cannot be simpler than P_S. Therefore, as P_S is consistent with $C|_{P_S}$ by construction, P_S is also the simplest program for $C|_{P_S}$. This leads to the following definition:

Definition 2: exposure. *Let P be a program that is SCS with respect to a special case S (a set of possible inputs). Let C be a set of test cases that are consistent with P, and let $C|_{P_S}$ be the subset of test cases in C that are also consistent with P_S (the special-case program for S). We say that C* exposes *the special case S if $C|_{P_S} \neq \emptyset$ and P_S is the simplest program consistent with $C|_{P_S}$.*

Examples. The `getRange` program (Figure 1) is SCS because the programs in Figure 2 are all special-case programs. For instance, the second program (duplicated below) is the simplest program for the special case where `end` equals `start + 1`. This is similar to the target program because both contain fragments of the form `list.add(Integer.valueOf(◦))`:

```
// Works for test case #3, and any other input where end == start + 1
List<Integer> getRange(int start, int end) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(Integer.valueOf(start));
    return list;
}
```

This special case is also exposed by the test cases in Figure 1, since the above program is also the simplest in the grammar to pass test case #3. If test case #3 were removed, then the special case would no longer be exposed. Overall, special cases are more likely to be exposed if the test cases cover a variety of scenarios, and adding more tests cannot decrease exposure.

As another example, consider a `capitalize` program that accepts a String `str` and returns `str.toUpperCase().charAt(0) + str.toLowerCase().substring(1)` if `str` is nonempty or `""` otherwise. This has several special-case programs, some of which are shown below with the corresponding special cases in comments:

```
return ""; // str is empty

return str.toUpperCase().charAt(0) + str.toLowerCase().substring(1); // str is nonempty

return str.charAt(0) + str.toLowerCase().substring(1); // First char needs no change

return str.toLowerCase(); // First character is uncased

return str.toUpperCase(); // All characters after the first are uncased
```

Note that fragments from the simpler special-case programs are useful in generating the larger special-case programs, which in turn help FrAngel generate the target program. By using fragments, FrAngel can quickly find programs with new complex behaviors that are relevant to the task.

Relation to mining fragments. Special-case similarity justifies our approach of mining fragments by the following informal argument. When FrAngel remembers a program P' that passes a subset of test cases $T \subseteq C$, we optimistically assume three properties:

(P1) The target program P is SCS with a special case S exposed by the test cases C
(P2) The subset T is responsible for exposing S, i.e., $T = C|_{P_S}$ and $P_T = P_S$
(P3) $P' = P_T$, i.e., the program remembered is actually the simplest solution for T

If these properties hold, then $P' = P_T = P_S$ is similar to the target program P, so fragments from P' are useful for generating P. Property **(P1)** is determined by the task (which might not be SCS) and the user's choice of examples (which might not expose special cases, if any); FrAngel has little control over these. But if **(P1)** holds, then **(P2)** must hold for some $T \subseteq C$. Finally, as soon as FrAngel randomly generates P_T as a candidate program, **(P3)** is guaranteed.

Of course, the three properties do not always hold for every remembered program, so some mined fragments might not help FrAngel find the target program. In practice however, the positive effect of the helpful fragments generally outweighs the negative effect of the unhelpful ones.

Why SCS is common. Special-case programs often arise through simplifications in many natural ways (these are not mutually exclusive, and multiple simplifications can be applied at once):

- *Fixpoints*: If a type-preserving operation f is applied to an expression e, and f has a fixpoint, then we can omit f for all inputs that cause e to be a fixpoint. For example, we can replace `str.toLowerCase()` with `str` whenever `str` contains no uppercase characters.
- *Simpler expressions*: If an expression e is equivalent to a simpler expression e' under certain conditions, then whenever those conditions hold, we can replace e with e'. For example, we can replace `arr[arr.length - 1]` with `arr[2]` whenever `arr` has length 3.
- *Degenerate values*: Some operations, especially arithmetic ones, can be simplified if an operand is degenerate. For example, we can replace `a + b` with `b`, and `a * b` with `0`, whenever `a` is `0`.
- *Control structures*: Inputs follow a specific control flow through a program's control structures, so we can partition inputs based on their control flows. For each group of inputs, we can tailor the program to that control flow by removing, simplifying, or unrolling control structures.
- *Edge cases*: If the program has edge-case logic, then all such logic can be omitted for general-case inputs. Also, the edge cases can sometimes be handled without the general-case code.

The above points show that simplifications are often possible, leading to similar and simpler programs. Usually, such simplifications result in special-case programs and thus special-case similarity. However, occasionally there is an even simpler program that is completely different. For instance, consider the task of returning the expression `(a * b) - a - b`. For the special case where `a` is `1`, we might simplify this to `(b) - a - b`. But the simplest program for this special case is actually `return -1;`, which is not at all similar to the target program. Hence, this program is not SCS with respect to the special case where `a` is `1`.

4.2 Limitations of Mining Fragments

Special-case similarity does not always hold. For example, the following program (a benchmark used by SyPet [Feng et al. 2017b]) is not SCS:

```
Rectangle2D scale(Rectangle2D rect, double x, double y) {
    return new Area(rect).createTransformedArea(
            AffineTransform.getScaleInstance(x, y)).getBounds2D();
}
```

Note that this program mostly contains datatype conversions, which often cannot be simplified—they cannot be fixpoints, and they are already the simplest way to convert between the types. We could eliminate `.createTransformedArea(...)` whenever `x` and `y` are `1.0`, but the program then collapses to `return rect;`, which is not similar to the target program (the overlap is trivial—since reasonable programs should use their arguments, the fragment `rect` provides no useful information).

Furthermore, mining fragments is not always successful, since the following issues might occur:

- *Task not SCS*: Without special-case programs, the mined fragments might be completely unrelated to the target program. In this scenario, property **(P1)**, described above, fails.
- *Unexposed special cases*: If the test cases are insufficient to expose any special cases, failing **(P1)**, then FrAngel is unlikely to remember or benefit from special-case programs.
- *Too much noise*: Even if FrAngel remembers a special-case program P_T that is similar to P, there might be many other subsets of test cases solved by irrelevant programs not similar to P. That is, **(P2)** might hold very rarely for subsets of test cases solved by candidate programs. Mining fragments from those irrelevant programs can dilute the positive signal provided by P_T. This often arises when there are many wrong ways of computing the right result, such as in functions that return Booleans or small integers. For those tasks, FrAngel often remembers programs like `return 1 + 2;`, but fragments from such programs are usually not helpful.

```
double sumPositiveDoubles(double[] arr) {
    double sum = 0.0;
    for (int i = 0; <ANGELIC>; i++)
        if (<ANGELIC>)
            sum = sum + arr[i];
    return sum;
}
```

Fig. 6. An example angelic program with two angelic conditions.

- *Special-case programs too complex*: FrAngel cannot benefit from special-case programs that are too complex to generate in the first place using the currently-mined fragments. In this case, **(P3)** might not hold within a reasonable amount of time, thus stalling progress.
- *Special cases eliminating structure*: Consider summing the elements of an array. A good solution stores the running total in a local variable, for instance with `ans += arr[i];` in a loop. However, special-case programs eliminate that variable by summing elements directly, e.g., `return arr[0] + arr[1] + arr[2];`. More generally, despite being helpful, the mined fragments might not reveal all aspects of the target program.

5 ANGELIC CONDITIONS

Recall that FrAngel uses an adaptive random search over programs. Angelic conditions factorize this search, allowing FrAngel to first find a code sketch without control structure conditions (an *angelic program*) and later search for the correct conditions once a satisfactory angelic program is found. This factorization helps FrAngel discover programs with complex behaviors involving control structures. When angelic conditions are used with the strategy of mining fragments, FrAngel can also identify the "purposes" of certain fragments—for instance, some functionality is only sometimes necessary and should go inside a conditional, and some functionality is meaningful if repeated and can go inside a loop. Recognizing these situations allows FrAngel to generalize and extend known behaviors.

An *angelic condition*, denoted by `<ANGELIC>`, can appear in an angelic program anywhere a control structure condition would normally appear. Informally, each time an angelic condition is evaluated, it can choose to be `true` or `false`, whichever would lead to a correct result if possible (as if angels were controlling the program's execution).

To provide intuition, consider the task of summing the positive values in an array, so the input `[-1.2, 3.4]` produces the output `3.4`. Suppose we generated the angelic program in Figure 6, which is a correct solution but with angelic conditions in the `for` loop and `if` statement. Hence, it represents summing an unspecified subset of the array elements. The precise execution of an angelic program can be described by a *code path*, which lists how the angelic conditions evaluate in the order that they are encountered. For instance, this angelic program produces the correct output with the code path *TFTTF*: the loop is entered (T), `arr[0]` is not summed (F), the loop is continued (T), `arr[1]` is summed (T), and the loop exits (F). The existence of a code path that produces the correct output hints that the angelic program might be correct, i.e., that the angelic conditions can be *resolved* (filled in) to create a correct non-angelic program.

Using this intuition, FrAngel optimistically executes angelic programs. In GetPassedTests from Algorithm 1, if P is angelic, FrAngel considers P to pass a test case (X, Y) if it finds a code path that causes P to produce the outputs Y when given the inputs X. Note that this is not based on the *existence* of a good path, only whether FrAngel finds one.

In Section 5.1 below, we formally describe how angelic programs are executed using code paths. In Section 5.2, we provide an algorithm that searches over code paths to determine if an angelic program passes a given test case. Finally, Section 5.3 discusses how we resolve angelic conditions.

5.1 Bitstring Code Paths

We previously explained that code paths describe a particular way of executing an angelic program. We now formalize this idea and introduce some terminology.

Code paths are represented as bitstrings over $\{T, F\}$, representing `true` and `false`, respectively. When we execute an angelic program P using a bitstring code path $b_1 \dots b_n$, we make the i-th angelic condition (as encountered during program execution) evaluate to b_i for $1 \leq i \leq n$, or `false` for $i > n$. With this convention, we do not need to consider any bitstring that ends in F, since it is semantically identical to the bitstring obtained by removing trailing Fs, e.g., the bitstrings *TFTFF* and *TFT* lead to identical executions. We choose this convention because repeatedly choosing the F branch of control structures will eventually lead to the method's termination, while repeatedly choosing the T branch can cause infinite loops.

When FrAngel begins to execute an angelic program P using a bitstring code path, we call this the *attempted* code path. Then, there is also a corresponding *actual* code path that takes into account the time of P's termination (returning or crashing). The actual code path will always be either the attempted code path with zero or more Fs appended, or a prefix of the attempted code path. For instance, suppose we run the angelic program in Figure 6 with the input `[-1.2, 3.4]` and the attempted code path *TTT*. After the three Ts, we need two more Fs to reach the `return` statement: one to bypass the inner `if` statement, and one more to break out of the loop. Hence, the actual code path is *TTTFF*. However, if the input were instead the empty array `[]`, then the actual code path would be *TT* because the program crashes when accessing `arr[0]`, before reaching the third T. We will use actual code paths to prune a search, described below.

5.2 Angelic Execution

To determine if an angelic program P passes a given test case, FrAngel searches for a code path that causes P to behave consistently with that test case. This must be done carefully; a brute-force search over all code paths (e.g., via recursive backtracking) is exponential-time if an angelic condition is inside a loop, since the angelic condition can evaluate to `true` or `false` independently on each iteration. We instead use an enumerative partial search over a bounded number of code paths. This prioritizes *simple* code paths, i.e., shorter paths with fewer Ts. Focusing on such paths allows us to assess simple test cases with short control flow traces, since following the T branch of a control structure causes more code to be executed than the F branch (at least for loops and `if` statements[6]).

Following this intuition, we enumerate bitstrings ordered first by increasing number of Ts, and then in lexicographical order where $T < F$, so *TFT* comes before *TFFT*, which comes before *TTT*. As mentioned in Section 5.1, we do not include bitstrings that end in F. Each enumerated bitstring is executed as an attempted code path, and the actual code path is recorded. During this enumeration, we also avoid all bitstrings that start with a previously-recorded actual code path. For instance, if we attempt the code path *T* and obtain the actual path *TFF*, we do not need to later attempt *TFFT*, since we already know that termination occurs after the *TFF*. This allows FrAngel to quickly prune the search and avoid all redundant code executions. FrAngel declares that the angelic program P passes the test case as soon as it finds some code path that causes the test to pass.

[6]Note that our grammar (Figure 4) does not include `else` statements, although these ideas can be extended to `else` and `else if` structures. For instance, we could force the smallest block to execute in the F case.

After executing simple code paths, further executions give diminishing returns as there are more paths of similar complexity. Hence, a constant maximum number M of paths are attempted; if none are successful, then FrAngel declares that P does not pass the test case. (Because this is only a partial search, there could be a correct but unattempted code path.) This gives us a parameterized tradeoff: if M is larger, angelic execution takes longer but searches more complex code paths. We set $M = 55$ in our experiments.[7]

As a concrete example, consider executing the angelic program in Figure 6 using the input `[-1.2, 3.4]`. The process is summarized below:

# *T*s	Attempted path	Actual path	Output
0	ϵ	*F*	`0.0`
1	*T*	*TFF*	`0.0`
	Skip *FT*, *FFT*, *FFFT*, etc. (starts with *F*)		
2	*TT*	*TTF*	`-1.2`
	TFT	*TFTFF*	`0.0`
	Skip *TFFT*, *TFFFT*, etc. (starts with *TFF*)		
	Skip *FTT*, *FTFT*, etc. (starts with *F*)		
3	*TTT*	*TTTFF*	`-1.2`
	Skip *TTFT*, *TTFFT*, etc. (starts with *TTF*)		
	TFTT	*TFTTF*	`3.4`
	Correct output; test case `[-1.2, 3.4]` passed.		

Note that after attempting *T*, the next smallest bitstring with one *T* is *FT*, which we skip because it starts with *F* (the actual code path for the previously attempted path ϵ). FrAngel simultaneously skips the infinite set of bitstrings *FFT*, *FFFT*, etc., for the same reason. Angelic execution ends after attempting *TFTT*, since this results in the correct output of `3.4` for this test case. We only needed 6 attempts to find a good code path of length 5 due to the heavy pruning of enumerated bitstrings.

So far, we have only described how an angelic program is executed on a single test case. Naturally, we would like to repeat this process for all test cases, but this can be slow as it involves trying many code paths for each test case. To avoid excessive slowdown, FrAngel terminates the overall angelic execution early when a $1 - \sigma$ fraction of the test cases have already failed (we set $\sigma = 0.75$ in our experiments). Note that even a correct angelic program might not pass all test cases if the required code path for some test case is too complex.

If an angelic program does pass at least a σ-fraction of the test cases, then FrAngel proceeds to the next step: resolving the angelic conditions to produce a non-angelic program.

5.3 Resolving Angelic Conditions

Once we have a promising angelic program P, the final step is to resolve the angelic conditions, replacing them with concrete expressions to obtain a non-angelic program passing the same tests.

Pseudocode is given in Algorithm 2. To resolve an angelic condition, we replace it with random Boolean expressions generated using *Procedure B* in Section 3.2. We evaluate the resulting program on the test cases, using angelic execution if appropriate. If it passes all test cases that P previously passed, then we have successfully resolved a condition and can proceed to resolve the next one. When resolving an angelic condition, we impose a time limit roughly proportional to the amount of time elapsed since the previous attempt to resolve conditions in a different angelic program.

We first try to resolve conditions from the innermost to the outermost. If that fails, we try again in the reverse order. If we are unable to resolve conditions, the program is discarded.

[7] If we have an `if` statement inside a `for` loop (and no other control structures), then the 55th enumerated code path is 8 consecutive *T*s. That is, within 55 code paths we will have tried all ways of performing at most 4 loop iterations.

Algorithm 2 ResolveAngelic

Input: Angelic program P, set of libraries L, set of fragments F, initially-passed tests T, all tests C
Output: A non-angelic program that passes all of T, or Failure

```
1:  procedure ResolveAngelic(P, L, F, T, C)
2:      while P contains angelic conditions do
3:          success ← ⊥
4:          repeat until timeout
5:              B ← GenBooleanExpression(L, F)
6:              P′ ← ReplaceNextAngelic(P, B)        ▷ Use B in place of one angelic condition
7:              T′ ← GetPassedTests(P′, C)
8:              if T′ ⊇ T then                       ▷ Successfully resolved that angelic condition
9:                  P ← P′
10:                 T ← T′
11:                 success ← ⊤
12:                 break
13:         if ¬ success then
14:             return Failure                       ▷ Timeout reached
15:     return P                                     ▷ All angelic conditions resolved
```

5.4 Limitations of Angelic Conditions

Angelic conditions are not very helpful when the desired program's output is a Boolean or small nonnegative integer. This is because the following angelic programs can pass all test cases for such tasks with an appropriate code path:

```
boolean unhelpfulBoolean(String str) {
    boolean var = false;
    if (<ANGELIC>)
        var = true;
    return var;
}

int unhelpfulInteger(String str) {
    int var = 0;
    for (int i = 0; <ANGELIC>; i++)
        var = i;
    return var;
}
```

These angelic programs are almost never correct, leading to wasted effort trying to resolve the conditions. Such tasks are difficult for FrAngel, and for synthesis in general, because the input-output examples provide less information, so many examples are necessary before the desired behavior becomes apparent and unambiguous.

6 EXPERIMENTS

We implemented FrAngel in about 7500 lines of Java and designed experiments to answer the following questions:

(Q1) How do mining fragments and angelic conditions contribute to FrAngel's performance?
(Q2) How does FrAngel compare to previous work?

Table 1. Benchmarks used in our experiments. We also list the number of components available to the synthesizer, and the number of provided examples, averaged over all tasks contained in each benchmark.

Benchmark	Tasks	Avg. # Components	Avg. # Examples
Geometry	25	400.8	4.72
ControlStructures	40	101.0	5.42
GitHub	25	292.0	4.36
SyPet	30	3639.4	1.43

(Q3) How complex are the tasks solved by FrAngel?
(Q4) How often does FrAngel produce incorrect programs?

We imposed a 30 minute timeout per synthesis task and a 4 GB memory limit for FrAngel. Results were averaged over 10 trials, running on Intel Xeon E5-2673 v3 (2.40 GHz) processors.

6.1 Benchmarks

We evaluated FrAngel on four benchmarks, comprising 120 tasks in total. Table 1 shows the number of tasks and the average number of components and examples for the benchmarks.

- *Geometry*: We designed 25 tasks of varying difficulty that use a variety of types from the `java.awt` and `java.awt.geom` libraries. These tasks are meant to demonstrate that our approach is domain-agnostic, since we use no domain-specific knowledge about these geometry tasks. 12 of these tasks also require control structures.
- *ControlStructures*: We designed 40 tasks that require using control structures. These tasks were designed to cover many common ways of using and combining control structures. Various types are involved, including primitives, strings, arrays, and several standard Java collections. Some tasks require combining three control structures (sequential and/or nested).
- *GitHub*: We selected 25 interesting tasks based on methods from 4 popular Java repositories on GitHub. 17 tasks require control structures, and 13 involve custom types defined in the open-source projects. Examples were drawn from existing unit tests if available, with some additional examples added to clarify functionality and expose special cases where needed.
- *SyPet*: We also use the 30 tasks used in the evaluation of SyPet [Feng et al. 2017b], none of which involve control structures. To match SyPet's search space as closely as possible, for these tasks we prevent FrAngel from using control structures, primitive operators, literals, and object fields. Although FrAngel handles polymorphism naturally, for this benchmark we also use the same hardcoded polymorphism information used by SyPet.

We created the *Geometry*, *ControlStructures*, and *GitHub* benchmarks based on the following general principles:

- If FrAngel is able to infer all necessary classes from the desired method signature, then we did not explicitly specify any other classes to use. This occurred for 58 of the 90 new tasks. Otherwise, we specified the extra required classes individually if the class containing the desired component(s) is easily identified (e.g., `java.lang.Math`), or by an entire package (e.g., `java.awt.geom.*`) if a specific class would not be obvious without knowing the solution.
- Examples were generally produced by writing one or two large general-case examples, duplicating them with slight modifications, and simplifying them to produce special cases. This frequently involved replacing different subsets of the inputs with degenerate values, and/or altering the examples to cover different control structure code flows.

- Creating examples was a slightly interactive process. By manually observing the fragments mined by FrAngel, the examples passed, and FrAngel's final outputs, we could identify underspecified tasks, incorrect examples, and examples that did not cover the special cases as intended. We gradually improved the benchmarks in this manner.
- We deliberately included tasks that FrAngel currently fails to solve, so that future work has room to demonstrate improvement when using our benchmarks.

6.2 SyPet Modifications

Our experiments compare FrAngel to SyPet [Feng et al. 2017b], an existing component-based synthesizer using a datastructure called a Petri net. We note that SyPet does not support control structures, but aside from SyPet, we are not aware of any other synthesizer that can generate an entire (multi-line) function implementation using arbitrary Java classes. Hence, we believe SyPet to be the existing work most suited to solve our benchmarks.

In order to run SyPet on our new benchmarks, we modified the open-source version of SyPet, primarily to provide support for JAR files compiled with newer versions of Java. As a result, our modified SyPet implementation is not functionally identical to the original open-source version—the modified version does not consider methods and classes that are deprecated or refer to unavailable classes, effectively pruning the space of programs considered by SyPet, leading to better efficiency overall. To increase our confidence in our modifications, we ran the modified and original versions of SyPet on SyPet's benchmarks. We observed that the original version solves 25 out of 30 tasks, while our modified version solves a superset of 26 tasks. Additionally, the modified version is faster for 25 tasks and slower for 1 task, with a median speedup of 1.95×.

We also point out that the modified version of SyPet is still slower than the times reported in the SyPet paper [Feng et al. 2017b] for two reasons. First, the "synthesis time" in the SyPet paper does not include the time required to construct an ILP representation of the Petri net and set up a solver for that ILP, a step necessary to find reachable paths in the Petri net (leading to solution sketches). Our measurements of SyPet's runtime do include this step.[8] Second, the SyPet paper reports times for a closed-source version of SyPet using some optimizations not included in the open-source version. Because we had to modify SyPet to address the above issues, we extended the open-source version without those optimizations.

For the rest of the discussion, we use "SyPet" to refer to our modified version of SyPet. When running SyPet on our new benchmarks, we provided SyPet with all of the constants and operators that are used by FrAngel (by wrapping them inside functions that SyPet can call). SyPet and FrAngel were also given the same libraries (i.e., Java classes). Because SyPet is deterministic and memory-intensive, we ran it once on each task with 12 GB of memory.[9] Even so, SyPet still crashed from insufficient memory on some tasks.

6.3 Results and Discussion

Using our benchmarks, we evaluated FrAngel and SyPet, together with versions of FrAngel that that only mine fragments ("Fragments") or only use angelic conditions ("Angelic"), or neither ("Random," a naïve random search). The results are shown in Figure 7, which plots the time required (per task) to solve a fraction of the benchmarks, e.g., 92.8% of tasks in *ControlStructures* were solved by FrAngel before the 30 minute per-task timeout. The plots show that Random outperforms SyPet, Fragments and Angelic are generally better than Random, and FrAngel performs the best.

[8] As in the SyPet paper, we exclude the time used by SyPet to compile candidate programs. But to be conservative in our claims, FrAngel's equivalent (interpreting candidates via reflection without compilation) is *included* in our reported times.
[9] Our experiments used machines with only 14 GB of RAM total. Experiments in the SyPet paper used 32 GB of memory.

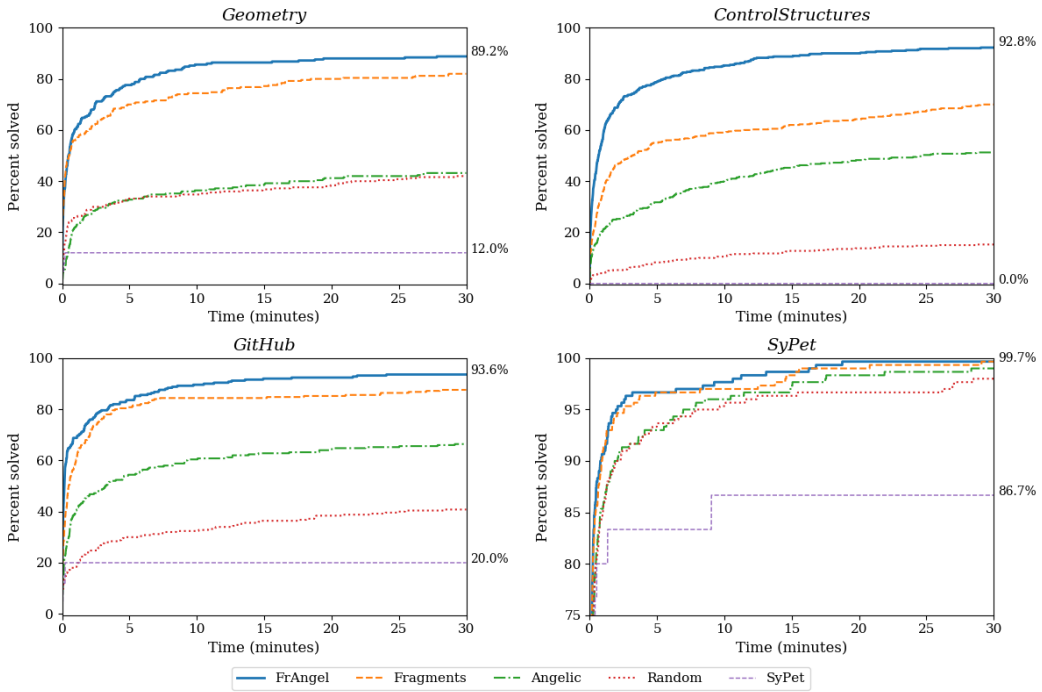

Fig. 7. Results for the four benchmarks. The success rates at 30 minutes for FrAngel and SyPet are listed on the right of each plot. (The plot for the *SyPet* benchmark uses a different *y*-axis scale.)

To illustrate FrAngel's abilities, in Figure 8 we list some of the most interesting programs generated (we removed unnecessary braces for space). In Figure 8a, FrAngel calculates ellipse eccentricity in a way simpler than the authors' best handwritten solution, implicitly taking advantage of the identity $\cos(\sin^{-1}(x)) = \sqrt{1-x^2}$. In Figure 8b, FrAngel implements a sorting algorithm in only 30 seconds when given a `swap` function.

Additionally, Figure 9 shows results for several tasks that can be solved in under 10 seconds, showing that FrAngel is fast enough to be used interactively for the easier but still interesting tasks in our benchmarks. Note that some of the implementations are not the most natural. For instance, in Figure 9's `longestString` implementation, `elem1.startsWith("", var1.length())` is more naturally written as `var1.length() <= elem1.length()`, although both have the same AST size. This is a fundamental limitation of the synthesis setting, where there might be many correct ways of obtaining the desired functionality, and a random search-based synthesizer is not at all guaranteed to find the "best" implementation.

Mining fragments, angelic conditions. To answer **(Q1)**, we compare FrAngel to Fragments, Angelic, and Random. Based on the relative performance of these variants, we conclude that mining fragments is an important strategy in all of our benchmarks. However, mining fragments is not as useful for the *SyPet* benchmark, because most of those tasks are not special-case similar or do not expose any special cases in the provided examples. Of the 30 tasks in the *SyPet* benchmark, 19 only provide a single example (test case). In this situation, mining fragments has no effect at all, since we only mine fragments from programs that pass a nonempty subset of test cases, and the search would terminate successfully as soon as the single test case is passed.

```
// Time: 280.0 sec (10/10), Size: 15, Components: 211, Examples: 8, Benchmark: Geometry
// Fragments: 285.6 sec (10/10), Angelic: not solved (0/10), Random: not solved (0/10)
static double ellipseEccentricity(Ellipse2D ellipse) {
    double var1 = 0.0;
    var1 = ellipse.getHeight() / ellipse.getWidth();
    return Math.cos(Math.asin(Math.min(var1, 1.0 / var1)));
}
```

(a) FrAngel's approach to computing ellipse eccentricity is actually simpler than our own handwritten solution, which used the formula $e = \sqrt{1 - b^2/a^2}$, where a and b are the lengths of the major and minor axes respectively. Mining fragments from special-case programs is a successful strategy for this task.

```
// Time: 30.1 sec (10/10), Size: 21, Comp.: 37, Examples: 5, Bench.: ControlStructures
// Fragments: not solved (0/10), Angelic: not solved (0/10), Random: not solved (0/10)
static void sortArrayGivenSwap(double[] arr) {
    for (int i1 = 0; i1 < arr.length; i1++)
        for (int i2 = 0; i2 < i1; i2++)
            if (arr[i1] < arr[i2])
                Swap.swap(arr, i2, i1);
}
```

(b) If given a `swap` function, FrAngel can use nested control structures to implement a variation of Insertion Sort in 30 seconds. In most other runs, FrAngel produces a Bubble Sort implementation instead.

```
// Time: 980.5 sec (7/10), Size: 17, Comp.: 36, Examples: 5, Bench.: ControlStructures
// Fragments: not solved (0/10), Angelic: 655.6 sec (10/10), Random: not solved (0/10)
static LinkedListNode reverseLinkedList(LinkedListNode node) {
    LinkedListNode var1 = null;
    while (!(null == node)) {
        var1 = new LinkedListNode(node.getValue(), var1);
        node = node.getNext();
    }
    return var1;
}
```

(c) FrAngel manipulates multiple variables to reverse a linked list. Angelic conditions are crucial for this task.

```
// Time: 432.1 sec (10/10), Size: 15, Components: 165, Examples: 2, Benchmark: GitHub
// Fragments: 196.8 sec (10/10), Angelic: not solved (0/10), Random: not solved (3/10)
static IndexResponse elasticsearch_fromXContent(XContentParser parser) {
    IndexResponse.Builder var1 = new IndexResponse.Builder();
    parser.nextToken();
    while (XContentParser.Token.FIELD_NAME.equals(parser.currentToken())) {
        IndexResponse.parseXContentFields(parser, var1);
        parser.nextToken();
    }
    return var1.build();
}
```

(d) FrAngel handles complex custom types in a domain-agnostic way. In this task, FrAngel automatically identifies all necessary methods and classes from the function signature without user assistance.

Fig. 8. Some of the most interesting programs synthesized by FrAngel. We display the solution produced by the 6th-fastest solve out of 10 runs. We also list the size of the displayed program, the number of components and examples for the task, and the 6th-fastest solve time and success rate for all FrAngel variants.

```
// Time: 3.7 sec, Size: 12, Components: 191, Examples: 4, Benchmark: Geometry
static Rectangle2D.Double rectangleUnion(Rectangle2D[] rects) {
    Rectangle2D.Double var1 = new Rectangle2D.Double();
    var1.setRect(rects[0]);
    for (Rectangle2D elem1 : rects)
        var1.add(elem1);
    return var1;
}

// Time: 9.1 sec, Size: 6, Components: 1421, Examples: 2, Benchmark: Geometry
static void rotatePointDegrees(Point2D point, double degrees) {
    AffineTransform.getRotateInstance(Math.toRadians(degrees)).transform(point, point);
}

// Time: 5.2 sec, Size: 13, Components: 137, Examples: 7, Benchmark: ControlStructures
static String longestString(List<String> list) {
    String var1 = "";
    for (String elem1 : list)
        if (elem1.startsWith("", var1.length()))
            var1 = elem1;
    return var1;
}

// Time: 9.2 sec, Size: 15, Components: 48, Examples: 6, Benchmark: ControlStructures
static void rotateQueue(Queue<Object> queue, int amount) {
    if (!queue.isEmpty())
        for (int i1 = 0; i1 < amount % queue.size(); i1++)
            queue.add(queue.poll());
}

// Time: 5.9 sec, Size: 14, Components: 119, Examples: 4, Benchmark: GitHub
static String guava_getPackageName(String classFullName) {
    if (!classFullName.contains("."))
        classFullName = ".";
    return classFullName.substring(0, classFullName.lastIndexOf("."));
}

// Time: 0.8 sec, Size: 11, Components: 213, Examples: 4, Benchmark: GitHub
static void zxing_maybeAppend(String[] values, StringBuilder result) {
    if (false == (values == null))
        for (String elem1 : values)
            ParsedResult.maybeAppend(elem1, result);
}

// Time: 8.0 sec, Size: 5, Components: 1093, Examples: 2, Benchmark: SyPet
static double[] sypet_06_solveLinear(double[][] mat, double[] vec) {
    return MatrixUtils.inverse(new BlockRealMatrix(mat)).operate(vec);
}

// Time: 0.3 sec, Size: 7, Components: 6072, Examples: 1, Benchmark: SyPet
static DocumentType sypet_26_getDoctypeByString(String xmlStr) {
    return DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
            new InputSource(new StringReader(xmlStr))).getDoctype();
}
```

Fig. 9. FrAngel solves a variety of interesting tasks in under 10 seconds, making the tool suitable for interactive use. We display the results and times for the 6th-fastest solve out of 10 runs.

Table 2. Various statistics for FrAngel related to mining fragments (over all successful runs, excluding the *SyPet* benchmark). Roughly speaking, the last column measures the average amount of overlap between mined fragments and the final program.

	# Programs Remembered	# Fragments Mined	Avg. Fragment Usefulness
Mean	6.4	19.4	62.2%
Median	4	16	63.9%
Max	62	148	100.0%

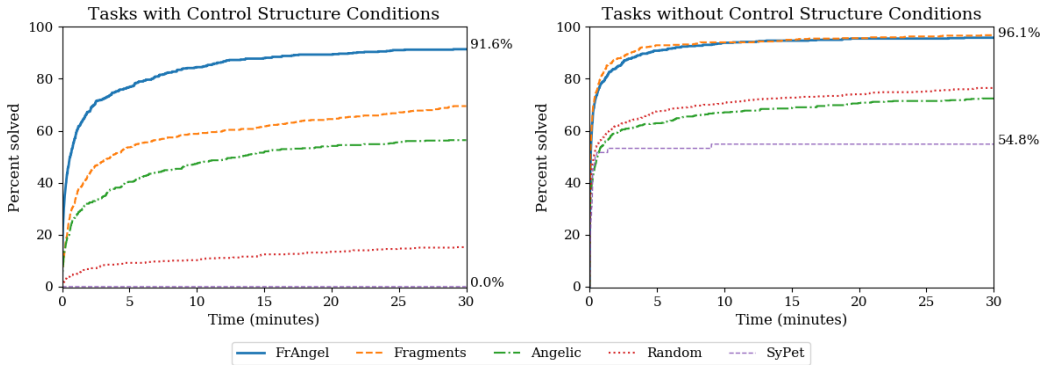

Fig. 10. Results separated by whether the task involves control structures with conditions (i.e., conditionals and loops excluding for-each loops). The success rate at 30 minutes for FrAngel and SyPet are listed on the right of each plot.

Table 2 lists various statistics about FrAngel related to mining fragments, considering all tasks except for the *SyPet* benchmark. For most tasks, FrAngel only needs to mine fragments from a handful of programs. Note that the number of remembered programs is much smaller than the theoretical maximum, i.e., one remembered program for each subset of test cases. Our tasks have up to 12 examples, yet the number of programs remembered is always relatively small.

For a given mined fragment, its "usefulness" measures the maximal amount of overlap between that fragment and some fragment in the final program. For example, if the final program is `a + b.foo(c)`, then the fragment `b.foo(1)` has a usefulness of 67%, since 2 AST nodes (i.e., `b` and `foo`), out of 3 total, match the fragment `b.foo(c)` from the final program.

For each successful run of FrAngel, we average the usefulness of all mined fragments to obtain the data points summarized in the table. Hence, in the average run of FrAngel, the average mined fragment has at most a 62.2% overlap with some fragment in the final program. This provides empirical evidence that mined fragments are useful for synthesis.

Angelic conditions are also generally helpful, but their effect is most prominent in the *Control-Structures* benchmark, where all tasks require control structures. This is expected, since angelic conditions are only relevant when control structures with conditions are involved (i.e., not for-each loops). This distinction is more apparent in Figure 10, which shows that angelic conditions are very helpful when control structures with conditions are involved, while the strategy causes a slight slowdown for tasks without control structure conditions.

SyPet comparison. We address **(Q2)** by comparing FrAngel to the previous work SyPet in Figure 7. It might seem surprising that a naïve random search (i.e., Random) outperforms SyPet. One contributing factor is that our grammar is more expressive than SyPet's grammar. For instance,

Random can use control structures but SyPet cannot, which explains why SyPet is not able to solve any tasks in our *ControlStructures* benchmark.

We also note that SyPet is designed for tasks that involve many datatype conversions. However, most of our new benchmarks perform many operations on the same types, e.g., `String` operations that return another `String`. We observed that SyPet struggles with such tasks. In particular, suppose `a`, `b`, `c`, and `d` are `String`s. SyPet takes 0.29 seconds to synthesize `a + b`, 28.2 seconds to synthesize `a + b + c`, and over 30 minutes (i.e., timeout) to synthesize `a + b + c + d`. In contrast, Random takes 0.003, 0.107, and 11.4 seconds respectively to synthesize those expressions.

To explain why Random outperforms SyPet even on the *SyPet* benchmark, we point out that our implementation achieves a very high throughput, defined as the number of programs tried per unit time. On the *SyPet* benchmark, SyPet executes on average 37 programs per second (PPS) on successful runs, not counting the time used to compile programs. In contrast, Random's average throughput exceeds 14,000 PPS. This difference is partially because SyPet spends a large portion of time constructing Petri nets and solving ILP and SAT problems, while Random does nothing beyond generating and executing candidate programs. On the *SyPet* benchmark, the other FrAngel variations have a similar throughput. Note that angelic conditions have no effect because, for consistency in our comparison to SyPet, we do not use control structures for the *SyPet* benchmark.

On our new benchmarks (i.e., excluding the *SyPet* benchmark), Random achieves a higher average throughput of 31,400 PPS because many tasks involve simpler types, so candidate programs can be evaluated much more quickly. FrAngel spends more time with each program—angelic programs involve executing many code paths, resolving conditions takes time, and mining fragments requires checking all test cases (whereas Random can break early as soon as one test case fails). Thus, FrAngel only tries on average 5,300 PPS. Fragments and Angelic have throughputs of 17,700 and 5,600 PPS respectively.

In our opinion, one takeaway from FrAngel's comparison to SyPet is that lightweight heuristic approaches (e.g., FrAngel or even Random) can be surprisingly effective in domains where execution of candidate programs is cheap, compared to other heavyweight approaches (e.g., using symbolic reasoning or deep learning) that can be fundamentally limited in throughput.

Task complexity. To address **(Q3)**, we use program size (number of AST nodes) as a measure of the underlying task's difficulty. More precisely, for each task, we consider all programs solving the task (produced by any FrAngel variant among all 10 runs), and we define that task's "size" to be the minimum size of such a program. It is possible that a task could be solved with a program of even smaller size; our notion of task size is only an approximate measure of the task's difficulty.

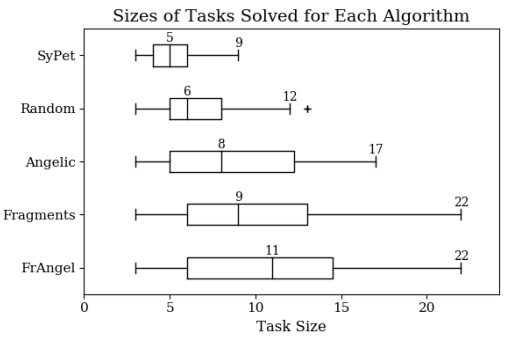

Fig. 11. Sizes of the tasks solved for each algorithm. A FrAngel variant "solves" a task if it succeeds at least 6 times out of the 10 runs.

Figure 11 shows the sizes of the tasks solved by each algorithm, where we consider a FrAngel variant to "solve" a task if it succeeds at least 6 times out of the 10 runs. Compared to the other methods, FrAngel solves the most difficult tasks, with a median task size of 11 and maximum of 22. SyPet is only able to solve tasks with size less than 10, most of which are from the *SyPet* benchmark. This suggests that the tasks in our new benchmarks are much more difficult than the tasks in *SyPet*.

```
static int sypet_23_getOffsetForLine(Document doc, int line) {
    int var1 = 0;
    line = doc.getDefaultRootElement().getElement(var1).getEndOffset();
    return line;
}

static double distanceInCircle(Point2D point, Ellipse2D circle) {
    if (!circle.contains(point))
        point = new Point2D.Double(Double.MAX_VALUE, 0.0);
    return point.distance(circle.getCenterX(), circle.getCenterY());
}
```

Fig. 12. Two representative false positives produced by FrAngel. (1) The first program returns the end offset of the element at index `0`, but it should return the start offset of the element at index `line`. SyPet's original benchmark only provides one example for this task, where `line` is 1. Further examples with different values of `line` would rule out this false positive. (2) `distanceInCircle` is supposed to return the distance between `point` and the center of `circle` if `point` is contained inside `circle`, or `Double.POSITIVE_INFINITY` otherwise; this implementation is incorrect in the latter case when `circle.getCenterX()` is `Double.MAX_VALUE`.

Note that Figure 11 does not reflect programs that are larger than necessary. On average, a program produced by FrAngel has a size 1.16× the task's size, and 64% of the time, the two sizes are equal. We thus conclude that FrAngel is reasonably consistent in the sizes of its solutions.

False positives. To answer **(Q4)**, we manually verified (by inspection and further testing) a random selection of 100 programs out of 1127 produced by FrAngel.[10] We found that 7 out of the 100 programs were *false positives*, i.e., they pass all given test cases but are not completely correct. Two representative false positives with explanations are shown in Figure 12.

The first false positive is for a *SyPet* task that only has one example. Hence, the task is somewhat ambiguously specified, so it is expected that random search approaches like FrAngel may produce false positives.[11] Further examples, if chosen reasonably, would eliminate that false positive. Of the 7 false positives, 3 were for *SyPet* tasks with only one example.

The second false positive is more difficult to guard against, since it is correct except for a narrow class of inputs. Furthermore, there are multiple variations of the false positive, and ruling them out via examples would require a different specially-crafted input for each variation.

Due to the possibility of false positives, we stress that the human users should independently verify the correctness of programs produced by FrAngel. Note that the possibility of false positives is common to most example-based synthesis approaches.

7 RELATED WORK

Past efforts in program synthesis have been quite diverse. Some approaches search through a space of programs using brute force [Bar-David and Taubenfeld 2003; Katayama 2005], MCMC search [Heule et al. 2015; Schkufza et al. 2016], genetic programming [Weimer et al. 2009], or neural networks [Balog et al. 2017; Devlin et al. 2017]. Others use constraint-solving [Feng et al. 2017a,b; Gulwani et al. 2011a; Jha et al. 2010; Srivastava et al. 2010] or version space algebras [Gulwani 2011; Lau et al. 2000]. FrAngel's random search is most similar to the MCMC and genetic programming approaches, and FrAngel enjoys the simplicity and generality offered by random search. However, unlike many solver-based approaches, FrAngel cannot guarantee program correctness.

[10] The programs and our verdicts can be found at https://www.github.com/kensens/FrAngel [Shi et al. 2018].
[11] SyPet is guaranteed to solve the task correctly because it is deterministic and because the SyPet authors added examples until SyPet produced the expected result.

FrAngel's focus on finding distinct program behaviors is one quality that sets it apart from approaches using other similar synthesis paradigms. For instance, standard enumerative approaches and MCMC search do not attempt to combine functionality drawn from previous partially-successful programs, which FrAngel does naturally by using mined code fragments. Furthermore, the candidate programs produced by enumeration or MCMC are often highly correlated, which can lead to a loss in diversity. On the other hand, FrAngel's use of fragments helps to balance diverse samples in the program space with learning from experience.

Genetic programming is based on the idea of combining previous successful programs to create better ones. However, the approach typically optimizes for a metric such as the number of test cases passed. This objective can lead to remembering many equally-high-scoring programs with nearly the same behavior, while programs containing desirable edge-case functionality might be discarded if they pass fewer tests. In contrast, FrAngel explicitly remembers the simplest program for every behavior distinguishable using the test cases, leading to a greater diversity in remembered program behaviors. FrAngel's strategy cannot be emulated within the standard genetic programming framework, as an equivalent "fitness function" must depend on the whole population: whether a program remains in the population depends on the presence of a simpler program passing the same tests or a superset of them. Furthermore, mining fragments can be viewed as a generalization of crossover and mutation from genetic programming, since FrAngel can combine and modify fragments from several programs simultaneously.

7.1 Synthesis Setting

Component-based synthesis. Component-based synthesis approaches are interesting because they generalize to different user-provided components. Such approaches have been applied to many areas, including string manipulation [Menon et al. 2013], bitvector algorithms [Gulwani et al. 2011a], deobfuscation [Jha et al. 2010], geometric constructions [Gulwani et al. 2011b], and transformations of data [Feng et al. 2017a; Feser et al. 2015; Harris and Gulwani 2011; Yaghmazadeh et al. 2016]. Some works focus on code completion, finding compositions of components that typecheck given the surrounding code context [Galenson et al. 2014; Gvero et al. 2013, 2011; Mandelin et al. 2005; Raychev et al. 2014].

From a user-interface perspective, our work is most similar to SyPet [Feng et al. 2017b]. Like FrAngel, SyPet is a component-based synthesizer that synthesizes Java programs from examples using arbitrary libraries. SyPet uses a Petri net datastructure to find sequences of function calls that produce the desired output type. However, SyPet cannot synthesize control structures and struggles with many operations on the same types, while FrAngel excels in these areas. Our experimental results show that FrAngel outperforms SyPet even on SyPet's benchmarks.

CodeHint [Galenson et al. 2014] is another component-based synthesis tool that produces a ranking of Java expressions given the surrounding code context. It performs synthesis at run-time, i.e., inside a debugging session paused at a breakpoint, using the actual execution context. CodeHint only produces one-line expressions, while most of our tasks require the use of multiple statements.

Programming by example (PBE). Many synthesizers including FrAngel use input-output examples to specify the desired behavior [Feng et al. 2017a,b; Feser et al. 2015; Gulwani 2011; Harris and Gulwani 2011; Menon et al. 2013; Osera and Zdancewic 2015; Perelman et al. 2014; Yaghmazadeh et al. 2016]. Most of these focus on a particular domain and require domain-specific knowledge in the form of a DSL or carefully-chosen components. In contrast, SyPet and FrAngel do not require domain-specific knowledge, only a list of relevant libraries.

A PBE approach by Menon et al. [2013] uses examples to extract "features" for string-processing tasks. For instance, if the input is a list with duplicate strings, and the output is a list without

duplicates, this feature suggests using a `dedup` operator. This is similar to FrAngel's technique of mining fragments, but FrAngel obtains fragments from previous programs, not by extracting features from examples.

7.2 Synthesis Techniques

Learning from previous programs. LaSy [Perelman et al. 2014] is a PBE approach inspired by test-driven development. From a sequence of increasingly-complex test cases, LaSy synthesizes a sequence of programs, each a modification of the previous program, where program i passes the first i test cases. Our idea of mining fragments (i.e., by remembering the simplest program that passes each subset of tests) could be seen as a generalization of the LaSy approach. In particular, we do not assume that test cases are ordered by complexity; instead, we allow remembered programs and fragments to build upon each other in a nonlinear fashion—new programs can use fragments extracted from any combination of previously-remembered programs.

Strata [Heule et al. 2016] synthesizes formal semantics of x86-64 instructions and uses previously-learned semantics to bootstrap the learning process of unknown semantics. Note that the problem addressed by Strata is very different from ours; FrAngel synthesizes the source code of Java functions instead of assembly instruction semantics.

Program sketching. A common synthesis technique, popularized by Sketch [Solar-Lezama 2008], is to provide the overall code context (a program *sketch*) while leaving "holes" to be filled by the synthesizer [Bodík et al. 2010; Feng et al. 2017a,b; Feser et al. 2015; Galenson et al. 2014; Raychev et al. 2014]. This technique is frequently used by synthesizers including FrAngel to perform synthesis in two steps: first identify promising program sketches and then complete the sketches. In FrAngel, angelic conditions are holes in place of control structure conditions, so angelic programs are a kind of program sketch.

FrAngel's angelic conditions are related to a programming model called *angelic nondeterminism* [Bodík et al. 2010]. In this paradigm, the programmer first writes a program sketch using a `choose` operator, which nondeterministically evaluates to a constant such that execution passes all `assert` statements (if possible). Thus, programs with `choose` operators are similar to our angelic programs. The programmer can then iteratively refine the program, slowly removing `choose` operators while maintaining the program's validity. Eventually, when all `choose` operators are removed, the programmer arrives at a deterministic program. This is similar to our approach of resolving angelic conditions one at a time. While the ideas in FrAngel overlap with those in angelic nondeterminism, they have different purposes: FrAngel uses angelic conditions to decompose a random search by evaluating the quality of partial programs, while angelic nondeterminism uses the `choose` operator to represent a class of programs that is gradually narrowed down by the user.

Synthesizing control structures. As noted in Section 1, many component-based synthesizers cannot handle control structures, especially loops [Feng et al. 2017b; Gulwani et al. 2011a; Jha et al. 2010; Menon et al. 2013]. Beyond component-based synthesis, some approaches in the wider program synthesis literature do handle loops [Bar-David and Taubenfeld 2003; Barthe et al. 2013; Heule et al. 2015; Qi et al. 2012; Srivastava et al. 2010], but many do not [Alur et al. 2017; Balog et al. 2017; Devlin et al. 2017; Lau et al. 2000; Schkufza et al. 2016]. Hence, handling combinations of control structures is a distinguishing aspect of FrAngel.

FrAngel uses the idea of generating and evaluating the body of a control structure before learning the condition. This provides a decomposition of the synthesis task, ultimately allowing FrAngel to synthesize programs using combinations of loops and conditionals. The concept of leaving a control structure condition unspecified has been applied to `if` and `switch` statements by several other works [Alur et al. 2017; Gulwani 2011; Heule et al. 2015; Yaghmazadeh et al. 2016].

This is often presented as learning several programs and then learning the classifier that chooses which program to use for a given input. Applying this to loops, as done by FrAngel, requires nontrivial extensions such as the partial enumeration procedure in Section 5.2.

The Mimic algorithm [Heule et al. 2015] infers a loop structure by analyzing execution traces (i.e., sequences of memory reads and writes). Mimic uses MCMC search and requires an executable that can produce traces on new inputs. While Mimic and FrAngel both try to learn control structures, their settings differ since FrAngel does not require execution traces.

Some synthesizers handle loops with techniques from program verification [Bar-David and Taubenfeld 2003; Barthe et al. 2013; Srivastava et al. 2010], but such approaches have only been demonstrated in low-level code (i.e., only using primitive operations and not library functions). Other approaches call components that loop internally [Feser et al. 2015; Harris and Gulwani 2011; Qi et al. 2012; Yaghmazadeh et al. 2016] or use a DSL for specialized loops [Gulwani 2011; Perelman et al. 2014], but these are only applicable in specific domains where the loops can be categorized into a few common types. FrAngel is the first approach to our knowledge that combines synthesis of control structures and domain-agnostic library function calls.

8 CONCLUSION

FrAngel is a domain-agnostic approach to component-based program synthesis from examples that mines code fragments from partial successes and uses angelic conditions to evaluate control structure sketches. These techniques help FrAngel discover programs with new and complex behaviors related to the synthesis task. Our experiments show that FrAngel can generate interesting programs using various libraries and combinations of control structures within several seconds.

For future work, we note that most of FrAngel's randomly-generated candidate programs would seem unnatural to human programmers, with symptoms including code with no effect (e.g., `str.length();`), excessive complexity (e.g., `rect.getBounds2D().getBounds2D()` when `rect` suffices), and improper use of variables (e.g., assigning to a variable that is never used again). While FrAngel can afford to process unnatural candidate programs due to its high throughput, the quality of the candidate programs could be greatly improved. This could allow FrAngel to use local variables in more complex ways, which is an aspect of programming that FrAngel currently struggles with. Furthermore, when generating code with fragments, we uniformly sample from all mined fragments. One could imagine a more intelligent scheme that takes into account information such as the fragment's size and the test cases passed by remembered programs containing it.

As another perspective on our work, FrAngel demonstrates that there is rich information in well-chosen test cases that can substantially aid program synthesis. In particular, FrAngel uses simple and varied test cases to give "partial credit" to simplifications of the desired task and to efficiently evaluate the feasibility of angelic programs. More generally, one could imagine other partial credit schemes, possibly with higher granularity, that convey even richer information. This could be a key component in scaling synthesis to more complex programs.

ACKNOWLEDGMENTS

The authors would like to thank Rishabh Singh and Osbert Bastani for their insightful discussions and assistance; Mina Lee, Cynthia Yin, and the anonymous reviewers for their valuable feedback on the paper; and Cynthia, Evan Liu, and Eva Zhang for their help in testing FrAngel's user interface.

REFERENCES

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations (ICLR)*.

Yoah Bar-David and Gadi Taubenfeld. 2003. Automatic Discovery of Mutual Exclusion Algorithms. In *International Symposium on Distributed Computing (DISC)*.

Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMDLoop Synthesis. In *Principles and Practice of Parallel Programming (PPoPP)*.

Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Principles of Programming Languages (POPL)*.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *International Conference on Machine Learning (ICML)*.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Programming Language Design and Implementation (PLDI)*.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-Based Synthesis for Complex APIs. In *Principles of Programming Languages (POPL)*.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Programming Language Design and Implementation (PLDI)*.

Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and Interactive Synthesis of Code Snippets. In *International Conference on Software Engineering (ICSE)*.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Principles of Programming Languages (POPL)*.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011a. Synthesis of Loop-Free Programs. In *Programming Language Design and Implementation (PLDI)*.

Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011b. Synthesizing Geometry Constructions. In *Programming Language Design and Implementation (PLDI)*.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Programming Language Design and Implementation (PLDI)*.

Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. In *Computer Aided Verification (CAV)*.

Theodore E Harris. 2002. *The Theory of Branching Processes*. Courier Corporation.

William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. In *Programming Language Design and Implementation (PLDI)*.

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Programming Language Design and Implementation (PLDI)*.

Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing Models for Opaque Code. In *Foundations of Software Engineering (FSE)*.

Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *International Conference on Software Engineering (ICSE)*.

Susumu Katayama. 2005. Systematic Search for Lambda Expressions. *Trends in Functional Programming* 6 (2005), 111–126.

Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *International Conference on Machine Learning (ICML)*.

Chris Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *International Conference on Machine Learning (ICML)*.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the APIJungle. In *Programming Language Design and Implementation (PLDI)*.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. 2013. A Machine Learning Framework for Programming by Example. In *International Conference on Machine Learning (ICML)*.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Programming Language Design and Implementation (PLDI)*.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-Driven Synthesis. In *Programming Language Design and Implementation (PLDI)*.

Dawei Qi, William N. Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. 2012. Modeling Software Execution Environment. In *Working Conference on Reverse Engineering (WCRE)*.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Programming Language Design and Implementation (PLDI)*.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (2016), 114–122.

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2018. FrAngel Source Code and Experimental Results. https://www.github.com/kensens/FrAngel.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. University of California, Berkeley.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Principles of Programming Languages (POPL).*

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE).*

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Programming Language Design and Implementation (PLDI).*

APPENDIX

Some minor details of our FrAngel implementation are described below.

Program Interpreter

Compiling Java code is quite expensive. To evaluate candidate programs as quickly as possible, FrAngel instead interprets programs using Java reflection. The interpreter accepts a Java program (as an AST, restricted to the grammar in Figure 4), inputs, and expected outputs, and returns whether or not the program behaves as expected. If the program uses angelic conditions, the interpreter also accepts a code path and returns the actual code path (as described in Section 5). The interpreter terminates execution early if the run consumes too much time or memory, implemented using limits on the number of loop iterations and sizes of objects.

In rare scenarios, FrAngel's interpreter will invoke a method with inputs that cause it to loop indefinitely. The interpreter is currently not equipped to deal with such cases (requiring the FrAngel process to be restarted), although further engineering effort using multithreading or multiprocessing should allow the interpreter to forcibly terminate execution of methods based on elapsed time.

Skipping and Pruning Programs

If FrAngel generates a non-angelic candidate program that it has already seen, evaluation of the program is skipped since it cannot provide new information. If FrAngel generates a duplicate angelic program, its evaluation is skipped with $\frac{3}{4}$ probability (resolving conditions is a random process that might or might not succeed, so FrAngel re-evaluates with probability $\frac{1}{4}$ for robustness). These checks are performed efficiently with low memory overhead by compressing every program into a compact and unique String encoding and storing the encodings in hashtables. One million encodings can be stored in about 200 MB of memory. We bound the overall memory usage of FrAngel by limiting the number of stored encodings. In our experiments, we store at most 5 million encodings each for angelic and non-angelic programs.

In addition, FrAngel prunes away control structures with non-angelic conditions that do not involve any variables (i.e., the condition is effectively constant). Furthermore, if any line in the program is too large, FrAngel skips the program entirely. This prevents FrAngel from synthesizing code that would be too convoluted for a human to easily parse.