Synthesizing Specifications for Synthesis

VIVIAN DING, Cornell University, USA
ALEX DRAKE, Cornell University, USA

We present a framework to generate programs in a user-specified target language while featuring a lightweight interface. A key insight behind our approach is that properties of a language which are useful for synthesis can themselves be synthesized with a programming-by-example approach, resulting in a tool which is both flexible and user-friendly. We decompose the synthesis goal into two subproblems—the synthesis of specifications on the target language, and the synthesis of programs guided by those specifications. We provide a partial implementation of our framework in a tool called SpyrAL.

CCS Concepts: • **Software and its engineering → Programming by example**.

Additional Key Words and Phrases: Program Specifications, Program Synthesis

1 INTRODUCTION

One limiting factor in program synthesis is the domain-specific knowledge required to write a synthesizer for a particular target language (hereafter referred to as "target"). Specifications of the target are often used to accelerate search through the program space. For instance, deduction rules about types and sizes of operator arguments can be applied to prune a search space and infer examples for subexpressions of a partially synthesized program. This approach yields significant speedup during top-down enumeration [Feser et al. 2015]. However, specifications in this form can be difficult to write, especially for users inexperienced with formal semantics, and they must be written separately for each desired synthesis target. As a result, a novice user is mostly restricted to existing synthesizers built for a particular target.

Existing work on synthesizing specifications requires the user to provide formal semantics of the target language [Park et al. 2023] or design and specify the property language themselves [Astorga et al. 2021; Park et al. 2023]. As a result, the application of such work is well-suited only to experienced users.

We present a framework that generates *approximate* specifications for input to another program synthesizer, with little burden on the user to supply sophisticated information about the target language. It is general enough for application to many targets, and produces nontrivial specifications with the potential to significantly accelerate synthesis in the target languages.

2 OVERVIEW

Our approach decomposes the goal into two synthesis subproblems: generating specifications on the target, and generating programs in the target using those specifications.

Step 1 (synthesizing specifications). The input to this step is an *initial specification* of the target, such as black-box access to operators, natural language documentation, or example input/output pairs. This initial specification ought to be simple to supply, even for inexperienced users. In this work, we focus on synthesizing specifications from positive examples of invoking operators in the target language. The property synthesizer then generates information useful for synthesizing programs in the target, the *synthesized properties*.

Our approach to the first step is centered around an algorithm for the following problem: *Given black-box access to a function f and expressive positive examples of input/output to f, attempt to find a precise formula—expressed in a unifying language of properties $\mathcal{U}$—that holds under all calls to f.*

Authors' addresses: Vivian Ding, vyd2@cornell.edu, Cornell University, USA; Alex Drake, amd459@cornell.edu, Cornell University, USA.

Step 2 (synthesizing programs). The input to this step is the result of Step 1, as well as examples specifying desired behavior of the target program. The result is a program in the target language which satisfies the provided examples.

Our work makes the following contributions:

- A framework and algorithm for the synthesis of language properties (§3).
- A proposed algorithm for the synthesis of programs using these properties (§4).
- A (work in progress) tool that we implemented to support our framework (§5), SpyRAL.
- An evaluation of SpyRAL's Step 1 on selected benchmarks (§6).

§7 proposes directions for future work. §8 concludes.

3 SYNTHESIZING SPECIFICATIONS

In this section, we describe our approach to generating specifications, step 1 of our framework.

3.1 Problem Statement

We first define the problem of synthesizing properties of a function application $x_o = f(x_1, \ldots, x_n)$. We are interested in properties which are consistent with provided positive examples and expressible in a unifying language of properties $\mathcal{U}$:

$$\begin{aligned} p &\rightarrow e\ cmp\ e \\ cmp &\rightarrow = \mid \leq \mid < \mid \geq \mid > \\ e &\rightarrow n \mid \texttt{len}(x) \mid e\ op\ e \\ op &\rightarrow\ +\ \mid\ -\ \mid\ \times \end{aligned}$$

Where x is any input or output variable, n any integer. For instance, below are some examples of properties we seek to synthesize, expressed in the form *function invocation* $\implies$ *property*. Note that although they look similar, the first two examples use a list-like library as the target, while the latter pertain to strings.

$$\begin{aligned} o = \texttt{append}(l, x) &\implies \texttt{len}(o) = \texttt{len}(l) + 1 \\ o = \texttt{replace}(l, i, x) &\implies \texttt{len}(o) = \texttt{len}(l) \end{aligned}$$

$$\begin{aligned} o = \texttt{substring}(s, i_1, i_2) &\implies \texttt{len}(o) \leq \texttt{len}(s) \\ o = \texttt{concat}(s_1, s_2) &\implies \texttt{len}(o) = \texttt{len}(s_1) + \texttt{len}(s_2) \end{aligned}$$

The goal of our framework is to synthesize sound, precise properties. For instance, the property $o = \texttt{concat}(s_1, s_2) \implies \texttt{len}(o) \geq \texttt{len}(s_1) \wedge \texttt{len}(o) \geq \texttt{len}(s_2)$ is sound, but not as precise as $\texttt{len}(o) = \texttt{len}(s_1) + \texttt{len}(s_2)$.

A user of this tool must provide the following inputs for the first step:
Queried function. The function f for which properties are to be synthesized.
Positive examples. A list of examples, each of the form $(i_1, \ldots, i_n, o)$, such that o is the result of calling f on inputs $i_1, \ldots, i_n$.
Implementation of $\mathcal{U}$. The implementations of basic primitives used in $\mathcal{U}$ (currrently, only `len`).

This interface is designed to be accessible to the novice user. Our framework does not require the user to supply any formal semantics; expressive positive examples of function application may even be taken from documentation. The usage of a unifying property language $\mathcal{U}$ alleviates the burden of choosing and describing the semantics of a suffiently expressive language from the user.

3.2 Algorithm

The algorithm synthesizes properties of one operator f in the target language at a time. For each f there exists a set of valid models over the arguments and return value, the semantics of f. Our algorithm uses provided positive examples as an underapproximation of the semeantics of f.

Our algorithm takes a modified counterexample guided inductive synthesis (CEGIS) approach. Given a set of positive examples E^+, it maintains a set of negative examples E^- and a hypothesized property h such that h evaluates to `true` on examples in E^+ and `false` on all those in E^-. Let $\overline{E^+} = \{(i, o) \mid \exists o' \neq o : \ (i, o') \in E^+\}$; that is, $\overline{E^+}$ denotes the set of assignments to input and output of f such that the assignments represent certainly invalid invocations of f.

First, the algorithm constructs an initial hypothesized property which satisfies all positive examples. Then, it iteratively produces new properties and accumulates negative examples through queries to a procedure called CheckPrecision. It ends when E^- reaches a fixpoint.

CheckPrecision attempts to return a new property which accepts all examples in E^+, rejects all examples in E^-, and rejects an additional negative example $e^- \in \overline{E^+} \setminus E^-$, whereas the previous property accepts e^-. The new property is a witness that the negative example e^- can be rejected by some property which otherwise agrees with the previous one, that is, the new property is a more precise fit to the input examples. After each call to CheckPrecision, e^- is added to E^-.

The algorithm is guaranteed to converge, since $\overline{E^+}$ is finite and $E^- \subseteq \overline{E^+}$.

3.3 (Un)Soundness and Precision

A property φ is sound with respect to f when $[\![f]\!] \subseteq [\![\varphi]\!]$, where $[\![\varphi]\!]$ is the set of models of the synthesized property φ and $[\![f]\!]$ is the set of valid models over the parameters to f.

Because our algorithm underapproximates $[\![f]\!]$ using E^+, our system does not guarantee soundness with respect to f. Instead, it only guarantees that $E^+ \subseteq [\![\varphi]\!]$, that is, the synthesized property is sound with respect to the positive examples.

Finding a sound property for a function f in the target is trivial; `true` is always one. On the other hand, a *most-precise* property is likely to overfit to the input examples, and therefore be unsound with respect to f even if it is sound with respect to the examples. Synthesizing a property which sacrifices soundness for excessive precision is an example of a phenomenon known as *overfitting*. Without a full description of the semantics of f, it is impossible to synthesize a sound, most-precise property for f. As a result, we sacrifice soundness for the sake of accessibility. In §4 we discuss how this limitation affects the second synthesis step. In §7 we describe how the underapproximation of $[\![f]\!]$ may be improved.

Note that we cannot directly set $E^- = \overline{E^+}$, since $\mathcal{U}$ may not be expressive enough to produce such a precise property. This necessitates the CEGIS-style CheckPrecision loop instead.

4 APPLYING SPECIFICATIONS TO SYNTHESIS

This portion of our framework is a work in progress. The second synthesis step takes as input examples specifying desired behavior of the target program, and properties of the target language generated by Step 1. Existing work details how top-down enumerative search can be accelerated by using deduction rules on the target language to prune the search space [Feser et al. 2015]. We propose the synthesized properties be used to guide a top-down search through the program space. However, pruning based on potentially unsound properties will make the search algorithm incomplete. Since properties synthesized in Step 1 are not guaranteed to be sound, they cannot be used to directly narrow the search space in Step 2.

Instead, we take inspiration from BUSTLE [Odena et al. 2021], which reweights partial programs after assessing their likelihood, affecting the order in which they are explored. We suggest that the

order of exploration of the program space be determined by properties in Step 1. If a synthesized property implies that a function cannot be applied at a particular point in program search, rather than pruning the branch entirely, the Step 2 search algorithm can simply decrease the weight of the function, exploring the space of programs using it later. This guarantees that the search is complete, despite being guided by unsound deduction rules.

5 IMPLEMENTATION

We (partially) implemented our framework in a tool called SpyrAL (Spyro-inspired Analysis of Libraries), which produces specifications by generating calls to the Sketch program synthesizer [Solar-Lezama et al. 2006]. In this section, we describe how synthesis tasks are encoded into Sketch queries. This includes a representation of positive and negative examples, specification of the property language $\mathcal{U}$, a representation of the user-defined key primitive `len`, and the CheckPrecision query.

Since SpyrAL has no access to the formal semantics of f nor the values in the input examples, all synthesis actions must treat them as black-box values. As such, it is without loss of generality that each unique value in the input examples is translated to a dummy integer value in the resulting Sketch query.

The encoding of positive and negative examples is as follows. For each positive or negative example, SpyrAL generates an assertion that the property being synthesized satisfies that example. This is seen in Figure 1.

```
// for each positive example o = f(i_1, ..., i_n)
boolean out;
property(i_1, ..., i_n, o, out);
assert out;
// for each negative example o ≠ f(i_1, ..., i_n)
boolean out;
property(i_1, ..., i_n, o, out);
assert !out;
```

Fig. 1. Sketch code for encoding satisfaction of examples in E^+, E^-.

```
generator int Int_gen(int x1, int x2, int o) {
    int t = ??;
    if (t == 0) { return 0; }
    if (t == 1) { return 1; }
    if (t == 2) { return len(x0); }
    if (t == 3) { return len(x1); }
    if (t == 4) { return len(o); }
    int e1 = Int_gen(x1, x2, o);
    int e2 = Int_gen(x1, x2, o);
    return op(e1, e2);
}
```

Fig. 2. Sketch code for encoding generation of integer values in $\mathcal{U}$.

The specification of $\mathcal{U}$ was encoded using Sketch generators. One nontrivial portion of this was that the integer generator includes a variable number of switches, which correspond to lengths of function parameters. For instance, if f takes two arguments $x1, x2$ and produces one output o, we produce the integer generator in Figure 2.

Notice that `Int_gen` takes three dummy integer values which uniquely represent the true values of $x1, x2, o$. Also, only 0 and 1 are provided as constants. Our implementation asks Sketch to minimize the size of generated expressions in $\mathcal{U}$. The intended consequence of this minimization and the restriction on constants is that arbitrary constants will rarely occur in generated properties, reducing the risk of overfitting (although this cause-and-effect has not actually been verified).

The function `len` was implemented as a concrete function which compares its input to each possible dummy value, returning the length of the value in the corresponding example if it exists. It rejects all inputs which do not correspond to a value in an example (asserts `false`). This suffices because `len` is only evaluated in order to check the correctness of synthesized properties using existing examples.

Finally, precision queries were constructed as seen in Figure 3. The function `negative_example_generator` toggles between various assignments to input parameters and outputs seen from existing examples, and asserts that the selected output is not the true output for the selected arguments.

```
harness void checkPrecision() {
    // synthesize a new negative example and store its assignments in
    //   x1, ..., xn, o
    negative_example_generator(x1, ..., xn, o);
    // negative example accepted by previous property
    assert previous_property(x1, ..., xn, o);
    // negative example rejected by new property
    assert !new_property(x1, ..., xn, o);
}
```

Fig. 3. Sketch code for generating properties with increased precision.

6 EVALUATION

We tested our implementation of Step 1 on a variety of functions in List and String libraries. Similar benchmarks have been used for evaluating existing work on synthesizing specifications [Park et al. 2023], and string functions are of interest due to the prevalence of program synthesizers for string programs. Figure 4 contains some of our results of running SpyrAL on string functions, and Figure 5 contains results for list functions.

Function	Examples	Property
`o = substring(x0)`	`["cat", 0, 1] -> "c";` `[compute, 0, 7] ->"compute"`	`length(o) <= length(x0)`
`o = concat(x0, x1)`	`["sub", "string"] ->"substring";` `["", concat] ->"concat"`	`length(x1) ==` `(length(o) - length(x0))`

`o = replaceFirst(x0, x1, x2)`	`["1 str, 2 str, 3 str", "str", !] -> "1 !, 2 str, 3 str";`
	`["1 str, 2 str, 3 str", "str", "!!!!!!!!!"]`
	`->"1 !!!!!!!!, 2 str, 3 str";`
	`["a", "b", "c"]->"a"`
		`length(x1) <= length(o)`

Fig. 4. Output of SpyrAL on string functions.

Most generated properties were reasonably informative and sound with respect to the queried function, indicating that only a few positive examples suffice for producing nontrivial specifications.

Function	Examples	Property
`o = add(x0, elem)`	`[[1, 2], 3] -> [1, 2, 3];` `[[], 1] -> [1]`	`(length(o) - length(x0))` `== 1`

`o = addAll(x0,x1)`	`[[1, 2], []] -> [1, 2];`
	`[[], [3]] -> [3];`
	`[[1, 2, 3], [5, 6, 7, 8]] ->`
	`[1, 2, 3, 5, 6, 7, 8]`
		`(length(o) - length(x1))`
		`== length(x0)`

`o = duplicate(x0)`	`[[]] -> [];`
	`[[1]] -> [1, 1];`
	`[[1, 2, 3]] -> [1, 1, 2, 2, 3, 3]`
		`(length(o) - length(x0))`
		`== length(x0)`

`o = removeAll(x0)`	`[[1, 2]] -> [];` `[[]] -> []`	`1 != length(o)`
`o = removeFirst(x0)`	`[[1, 2]] -> [2];` `[[1, 2, 3]] -> [2, 3]`	`(length(x0) - length(o))` `== 1`

`o = max(x1, x2)`	`[[1, 2], [1]] -> [1, 2];`
	`[[], [0]] -> [0];`
	`[[1], [1, 2, 4]] -> [1, 2, 4]`
		`(length(o) - length(l0))`
		`>=`
		`(length(l1) - length(o))`

`o = filterBy0(x0)`	`[[1, 2]] -> [1, 2];` `[[1, -1, 0, 2, 4]] -> [1, 2, 4]`	`length(x0) >= length(o)`

Fig. 5. Output of SpyrAL on list functions.

However, some synthesized properties were indeed not sound, as expected. Notice that the property for `replaceFirst` is unsound; the argument `x1` is the substring to search for in `x0`, and can quite possibly be longer than the string `o`. Since the input examples do not represent this situation, it is erroneously rejected by the synthesized property.

On the other hand, notice that the property generated for `removeAll` is not as precise as possible, as the property that `length(o) == 0` would be stronger. This is a result of our procedure for generating negative examples. Since outputs for negative examples are only sampled from outputs of existing positive examples, invariants involving constants that hold across all function calls are not found. An easy solution to this is to broaden the pool of negative examples by allowing outputs to be chosen from any known value of the correct type[1].

7 FUTURE WORK

We are interested in many directions for future work. First, we would like to complete an implementation of our suggested algorithm for synthesizing programs using potentially unsound deduction rules (Step 2), and evaluate its efficacy.

Second, our results suggest that the quality and number of available positive examples heavily influence the quality of generated properties. We hope to investigate improving the underapproximation of operators in the target language during Step 1 by automatically generating additional positive examples. This could be done by taking values from input examples and re-combining them to produce all possible well-typed arguments to the operators, and could involve values from examples across multiple operators in the target language. Given black-box access to the target language, these assignments to arguments can then be used to produce new positive examples by simply invoking the associated functions.

A natural generalization of this idea is to simply request the user to input various values for types involved with the operators. Then, our tool would be free to mix and match these values, invoke the operators on them, and produce as many positive examples as possible.

[1]Not included in this submission due to timing constraints.

Another approach to generating more positive examples is requesting that the user provide functions which modify values. These need not be actually implemented by the user; simply passing in an existing function, for example `List.append()` or `String.substring()`, would suffice. An example generator could then simply randomly modify existing values, producing new values from which additional examples can be produced.

Third, designing a property language which is universal enough to be defined for all targets but expressive enough to accelerate synthesis is quite challenging. Our approach could be made far more powerful by allowing a more expressive language of properties. However, nontrivial features of values often vary across target languages. We propose allowing the user to opt-in to providing additional definitions of feature extraction, in addition to `length`. For instance, adding a notion of matrix shape to $\mathcal{U}$ might enable synthesis of programs employing popular libraries such as NumPy or TensorFlow. It is important that this feature extraction be opt-in to reduce burden on the user; however, having access to an extensible language of properties will allow this framework to apply practically to a far greater range of target languages.

Fourth, we hope to formalize the notion of a reasonable balance between soundness and precision. Prior work on this topic either defines the balance in the functional synthesis setting [Park et al. 2023], or relies on syntactic constraints for the inductive synthesis setting [Astorga et al. 2021].

Intuitively, positive examples impose a notion of soundness, while negative examples guide improvements in precision of the hypothesized property. Since our algorithm only synthesizes counterexamples which are *certainly* invalid invocations of f (formally, $E^- \subseteq \overline{E^+}$), it seems that the way in which properties become more precise[2] is heavily intertwined with how representative the input examples are with respect to states expressible by $\mathcal{U}$.

Another interesting measure of precision or "insight" is whether a property *distinguishes* an operator from other functions of the same type. One future approach to explore might be randomly enumerating many functions of the same type as a target operator f, and searching for properties whose behavior is different on f compared to the randomly generated functions.

Finally, we are interested in extending this framework beyond a programming-by-example approach. We would like to investigate whether specifications useful for synthesis can be generated from other widely available sources of information, such as natural language documentation.

8 CONCLUSION

This paper presents a framework for synthesizing programs in a user-specified target language while keeping the interface lightweight. We present an algorithm for synthesizing properties of a target using examples of valid operator invocations, and propose an approach to using these properties in a program generation step.

The ultimate goal of this work is to step in the direction of program synthesizers that work for any desired language or library, without need for extensive user understanding of the target or formal languages nor training a large machine learning model[3]. In this world, a program synthesizer could learn to generate code without any need for information beyond what programmers would use to learn the same target language: documentation.

[2]Intentionally handwavy, as this concept is not well understood

[3]For instance, a user may hope to synthesize programs in a target only accessible to their company, or a target which was only recently released. In these cases, little training data is available, but documentation might be.

REFERENCES

Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing Contracts Correct modulo a Test Generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (oct 2021), 27 pages. https://doi.org/10.1145/3485481

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2021. BUSTLE: Bottom-Up Program Synthesis Through Learning-Guided Exploration. arXiv:2007.14381 [cs.PL]

Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023. Synthesizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 285 (oct 2023), 30 pages. https://doi.org/10.1145/3622861

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (oct 2006), 404–415. https://doi.org/10.1145/1168919.1168907