Component-Based Synthesis for Complex APIs

Yu Feng
University of Texas at Austin, USA
yufeng@cs.utexas.edu

Ruben Martins
University of Texas at Austin, USA
rmartins@cs.utexas.edu

Yuepeng Wang
University of Texas at Austin, USA
ypwang@cs.utexas.edu

Isil Dillig
University of Texas at Austin, USA
isil@cs.utexas.edu

Thomas W. Reps
University of Wisconsin-Madison, USA
reps@cs.wisc.edu

Abstract

Component-based approaches to program synthesis assemble programs from a database of existing components, such as methods provided by an API. In this paper, we present a novel type-directed algorithm for component-based synthesis. The key novelty of our approach is the use of a compact Petri-net representation to model relationships between methods in an API. Given a target method signature $\mathcal{S}$, our approach performs reachability analysis on the underlying Petri-net model to identify sequences of method calls that could be used to synthesize an implementation of $\mathcal{S}$. The programs synthesized by our algorithm are guaranteed to type check and pass all test cases provided by the user.

We have implemented this approach in a tool called SyPet, and used it to successfully synthesize real-world programming tasks extracted from on-line forums and existing code repositories. We also compare SyPet with two state-of-the-art synthesis tools, namely InSynth and CodeHint, and demonstrate that SyPet can synthesize more programs in less time. Finally, we compare our approach with an alternative solution based on hypergraphs and demonstrate its advantages.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program Synthesis

Keywords Type-directed, Petri-net, Component-based, Program Synthesis

1. Introduction

The goal of *component-based synthesis* is to automatically generate loop-free programs from a collection of base components, such as methods provided by an API [15, 26]. Considering the explosion of software libraries over the last few decades, component-based synthesis promises to simplify programming by automatically composing the building blocks needed to achieve some implementation task. Hence, instead of spending precious time in learning how to use existing libraries, programmers can focus on challenging algorithmic tasks.

Despite significant advances in component-based synthesis over the last several years [15, 16, 26, 38], existing algorithms have two key shortcomings: First, they can only handle a small number of components, typically in the range of 5-20 methods; but real-world APIs typically involve thousands of procedures. Second, most existing tools require logical specifications for the underlying components; however, few APIs contain methods that are formally specified. As a result, the applicability of component-based synthesis remains limited to domain-specific applications, such as bit-vector, string, or data-structure manipulations [9, 26, 43].

In this paper, we propose a new algorithm for component-based synthesis that overcomes both of these difficulties. Similar to recent work on type-directed API-completion [19, 20, 30, 35], our algorithm uses types as a coarse proxy for logical specifications and can handle APIs with thousands of procedures. However, unlike API completion tools, our algorithm does not require a partial implementation, and can synthesize complete programs from method signatures and test cases. The programs synthesized by our approach are always guaranteed to type-check and pass all user-provided tests. Furthermore, our approach is oblivious to the underlying components, and can be used to synthesize Java code using any combination of APIs.

The workflow of our synthesis algorithm is illustrated in Figure 1. At a technical level, a key idea underlying our approach is to represent relationships between API components using a certain kind of *Petri net* where places (nodes) correspond to types, transitions represent methods, and tokens denote the number of program variables of a given type. For example, Figure 6 shows a Petri net that describes the relationships between a subset of the functions in the `java.awt.geometry` API. Given such a Petri net $\mathcal{N}$ and a target configuration defined by the method signature, our algorithm performs reachability analysis on $\mathcal{N}$ to identify a sequence of transitions (i.e., method calls) that "produce" the output type by "consuming" the input types.

In our approach, a reachable path in the Petri-net model corresponds to a program sketch rather than a complete executable program. In particular, to keep the underlying Petri net representation compact, our algorithm deliberately decomposes the synthesis task into two separate *sketch-generation* and *sketch-completion* phases. Hence, after we perform reachability analysis on the Petri net, we must still complete the sketch by determining what arguments to provide for each procedure. Toward this goal, our algorithm generates constraints that encode various syntactic and semantic requirements on the synthesized program, and uses a SAT solver to find a model. The satisfying assignment produced by the solver is then used to generate a candidate implementation that can be tested. If

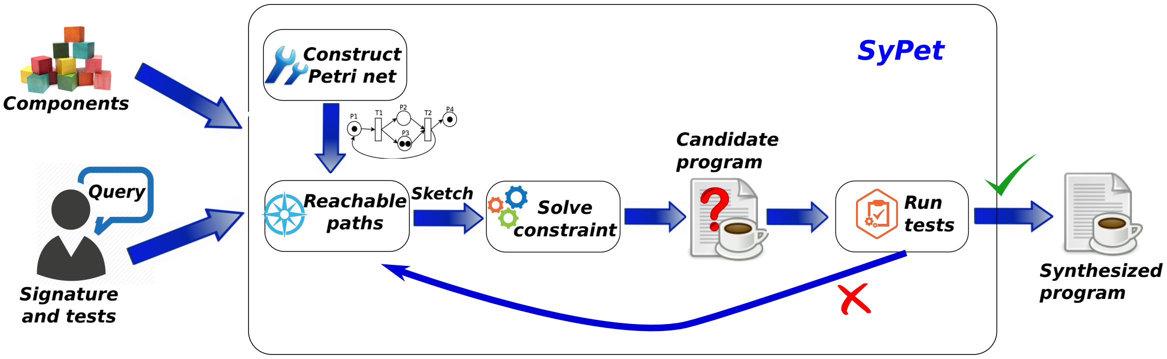


Figure 1. Workflow of the SYPET tool

the synthesized program fails any test case, our algorithm backtracks and generates a different implementation, either by finding another model of the SAT formula or by exploring a different reachable path in the Petri net.

At a very high level, our algorithm can be viewed as a generalization of techniques that use graph-reachability analysis for API completion. For example, standard graph reachability has been used to synthesize *jungloids*, which are sequences of *single argument* methods [30]. However, because our goal is to develop a *general solver* for component-based synthesis, we require a more expressive graph representation that can faithfully model relationships between multi-argument functions. In this work, we choose to use Petri nets as the underlying formalism because they have several advantages compared to other generalized graph representations, such as hypergraphs. As we show later in the paper, Petri nets allow us to synthesize a larger class of imperative programs, including those that call the same procedure multiple times or where components can have side effects.

Contributions. This paper makes the following contributions:

- We propose a novel type-directed algorithm for component-based program synthesis. Our algorithm can be instantiated with any set of APIs and only requires the user to specify a method signature and a few test cases.
- We show how Petri nets can be used for automatically generating program sketches from signatures of API components. We also propose a customized symbolic Petri-net-reachability solver that takes advantage of certain properties of the Petri nets constructed by our approach.
- We describe an implementation of our approach in a tool called SYPET and instantiate it with different Java APIs. We show that SYPET can successfully synthesize non-trivial programming tasks collected from online forums and Github projects.
- We compare SYPET against other state-of-the-art synthesis systems as well as variants of SYPET that use hypergraphs instead of Petri nets. The results demonstrate that our algorithm compares favorably with other tools and alternative solutions.

The rest of this paper is organized as follows: First, we start by presenting an example to motivate our approach (Section 2) and provide some necessary background on Petri nets (Section 3). After presenting an outline of the main synthesis algorithm in Section 4, we then elaborate on the core technical pieces in Sections 5, 6 and 7. In Sections 8 and 9, we describe implementation details and present our main experimental results. In Section 10, we compare our approach against an alternative solution based on hypergraphs and survey related work in Section 11.

```
public void test1() {
    Area a1 = new Area(new Rectangle(0, 0, 10, 2));
    Area a2 = new Area(new Rectangle(-2, 0, 2, 10));
    Point2D p = new Point2D.Double(0, 0);
    assertTrue(a2.equals(rotate(a1, p, Math.PI/2)));
}
```

Figure 2. Example test case for the `rotate` method

2. Motivating Example

Consider a programmer, Bob, who wants to implement functionality for rotating a 2-dimensional geometric object. Specifically, Bob has the following signature in mind:

```
Area rotate(Area obj, Point2D pt, double angle)
```

Here, the `rotate` method should take a 2-dimensional object called `obj` and return a new object that is the same as `obj` except that it has been rotated by the specified `angle` around the specified point `pt`. The types `Area` and `Point2D` are defined in the `java.awt.geom` library. Bob thinks that there is probably a way of implementing this functionality using the `java.awt.geom` package, but he cannot figure out how.

SYPET can help a programmer like Bob by automatically synthesizing the desired `rotate` method. To use SYPET, Bob only needs to provide (a) the method signature above, and (b) write one or more test cases. In this case, suppose Bob has written the unit test shown in Figure 2. This test creates a rectangle `a1` and its variant `a2` that has been rotated by 90°; it then asserts that invoking `rotate` on `a1` yields an object that is identical to `a2`.

Given this test case and method signature, SYPET automatically synthesizes the implementation of `rotate` shown in Figure 3 in 2.01 seconds. Observe that writing this code is non-trivial for a programmer like Bob for several reasons: First, Bob must know about the existence of a class called `AffineTransform` in the `java.awt.geom` library. Second, he must know about (and correctly use) the `setToRotation` method, which sets up a matrix representing the desired transformation. Finally, the call to `createTransformedArea` creates a new `Area` object that contains the same geometry as `obj`, but transformed by the specified transformation `at`. Hence, from the user's perspective, SYPET can significantly boost programmer productivity by automatically finding the relevant API methods and invoking them in the right manner.

From the synthesizer's perspective, automatically generating an implementation of `rotate` offers several challenges: First, the `java.awt.geom` library, which we use to synthesize this code, contains 725 methods. Hence, even though the implementation consists of just 6 lines of code, the number of components is quite large.

```
Area rotate(Area obj, Point2D pt, double angle) {
    AffineTransform at = new AffineTransform();
    double x = pt.getX();
    double y = pt.getY();
    at.setToRotation(angle, x, y);
    Area obj2 = obj.createTransformedArea(at);
    return obj2;
}
```

Figure 3. Implementation synthesized by SyPet

Second, even when we restrict ourselves to code snippets of length 3 (measured in terms of the number of API calls), there are already over 3.1 million implementations of `rotate` that type check. Because the search space is so large, finding the right implementation of `rotate` is akin to finding a needle in the proverbial hay stack.

3. Primer on Petri Nets

Because the remainder of this paper relies on basic knowledge about Petri nets, we first provide some background on this topic.

3.1 Petri Net Definition

A Petri net is a bipartite graph with two types of nodes: *places*, which are drawn as circles, and *transitions*, represented as solid bars (see Figure 4). Each place in a Petri net can contain a number of *tokens*, which are drawn as dots and typically represent resources. A *marking* (or *configuration*) of a Petri net is a mapping from each place p to the number of tokens at p. Transitions in the Petri net correspond to events that change the marking. In particular, incoming edges of a transition t represent necessary conditions for t to fire, and outgoing edges represent the outcome. For example, consider transition T_1 from Figure 4. A necessary condition for T_1 to fire is that there must be at least one token present at P_1, because the incoming edge to T_1 has weight 1. Because the precondition of this transition is met, we say that T_1 is *enabled*. If we fire transition T_1, we consume one token from place P_1 and produce one token at place P_2, because the outgoing edge of T_1 is also labeled with 1. Figure 5 shows the result of firing T_1 at the configuration shown in Figure 4. Observe that transition T_2 is *disabled* in both Figure 4 and Figure 5 because there are fewer than two tokens at place P_2.

Definition 1. *(***Petri net***) A Petri net $\mathcal{N}$ is a 5-tuple (P, T, E, W, M_0) where P is a set of places, T is a set of transitions, and $E \subseteq (P \times T) \cup (T \times P)$ is the set of edges (arcs). Finally, W is a mapping from each edge $e \in E$ to a weight, and M_0 is the initial marking of $\mathcal{N}$.*

Example 1. *Consider the Petri net shown in Figure 4. Here, we have $P = \{P_1, P_2, P_3\}$ and $T = \{T_1, T_2, T_3\}$. Let e^* be the edge $P_2 \to T_2$. We have $W(e^*) = 2$, and $W(e) = 1$ for all other edges e in E (e.g., $P_1 \to T_1$). The initial marking M_0 assigns P_1 to 2, and all other places to* 0.

A *run* (or *trace*) of a Petri net $\mathcal{N}$ is a sequence of transitions that are fired. For instance, some feasible runs of the Petri net shown in Figure 4 include T_1, T_1, T_2 and T_1, T_1, T_2, T_3. However, T_1, T_2 and T_1, T_2, T_3 are not feasible.

3.2 Reachability and k-safety in Petri Nets

A key decision problem about Petri nets is *reachability*: Given Petri net $\mathcal{N}$ with initial marking M_0 and *target marking* M^*, is it possible to reach M^* by starting at M_0 and firing a sequence of transitions? For instance, consider Figure 4 and target marking $M^* = [P_1 \mapsto 0, P_2 \mapsto 0, P_3 \mapsto 1]$. This marking is reachable because we can get to marking M^* by firing the sequence of transitions T_1, T_1, T_2. The *reachable state space* of a Petri net $\mathcal{N}$, denoted $\mathcal{R}(\mathcal{N})$, is the set of all markings that are reachable from the initial state. Given Petri net $\mathcal{N}$ and target marking M^*, a run of $\mathcal{N}$ is *accepting* if it ends in M^*.

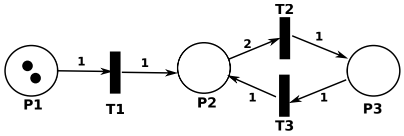


Figure 4. A simple Petri net

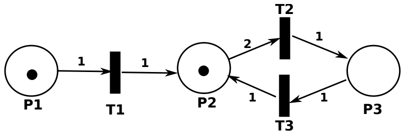


Figure 5. Result of firing T_1 in Figure 4

Another important concept about Petri nets is *k-safety*: A Petri net $\mathcal{N}$ is said to be *k-safe* if no place contains more than k tokens for any marking in $\mathcal{R}(\mathcal{N})$. For example, the Petri net of Figure 4 is 2-safe, because no place can contain more than 2 tokens in any configuration. However, if we modify this Petri net by adding a back edge from T_1 to P_1 (with an arc weight of 1), then the resulting Petri net is not k-safe for any k. As we will see later, the notion of k-safety plays an important role in the reachability analysis of Petri nets because the reachable state space $\mathcal{R}(\mathcal{N})$ is bounded iff $\mathcal{N}$ is k-safe.

4. Algorithm Overview

We now give an overview of SyPet's synthesis algorithm and illustrate how it works on the example from Section 2. As shown in Algorithm 1, the Synthesize procedure takes a method signature $\mathcal{S}$, a set of components Λ, and test cases $\mathcal{E}$. Its output is either $\perp$, meaning that the specification cannot be synthesized using components Λ, or a well-typed program that passes all test cases $\mathcal{E}$.

Petri-net construction. The first step of our synthesis algorithm is to construct a Petri net using *signatures* of components in Λ. In particular, the procedure ConstructPetri in Algorithm 1 constructs a Petri net $\mathcal{N}$ where each transition is a component $f \in \Lambda$ and each place correspond to a type. If there is an edge in the Petri net from τ to f with weight w, component f takes w arguments of type τ. Similarly, an edge from f to τ' indicates that f's return value has type τ'.

Example 2. *Figure 6 shows (a small part of) the Petri net generated by* ConstructPetri *for the example from Section 2. The transition labeled* `getX` *has one incoming edge of weight 1 from* `Point2D` *because it takes a single argument of this type. There is also an edge from* `getX` *to* `double` *because* `getX`*'s return value is* `double`*. As another example, the weight of the edge from* `double` *to* `setToRotation` *is 3 because this method requires three arguments of type* `double`*. Note that Figure 6 also contains special* ***clone transitions*** *labeled κ: Intuitively, these κ transitions allow us to duplicate tokens. As we will see in Section 5, the clone transitions allow us to reuse program variables in the synthesis context.*

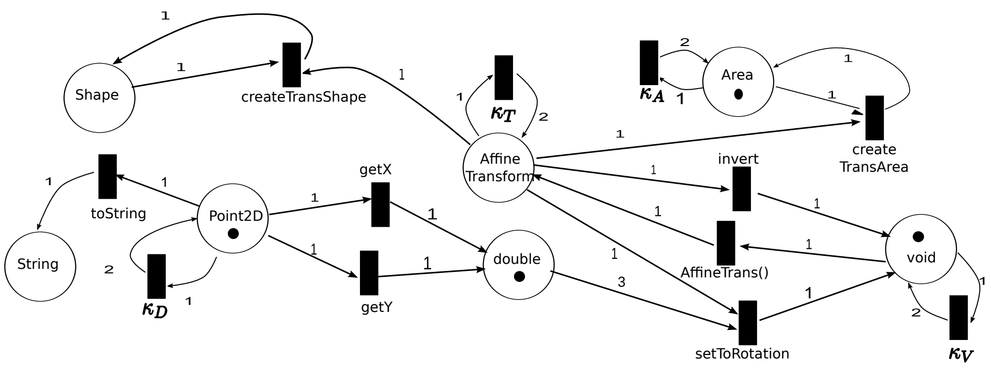


Figure 6. Petri net for motivating example

```
Algorithm 1 Synthesis Algorithm
1:  procedure SYNTHESIZE($\mathcal{S}$, $\Lambda$, $\mathcal{E}$)
2:      Input: Signature $\mathcal{S}$ of method to synthesize,
3:              components $\Lambda$, and tests $\mathcal{E}$
4:      Output: Synthesized program or $\bot$ for failure
5:      $(\mathcal{N}, M^*)$ := CONSTRUCTPETRI($\mathcal{S}$, $\Lambda$)
6:      while true do
7:          $\pi$ := GETNEXTPATH($\mathcal{N}$, $M^*$)
8:          $(\Sigma, \phi)$ := SKETCHGEN($\pi$)
9:          for all $\sigma \in$ MODELS($\phi$) do
10:             if RUNTESTS($\Sigma[\sigma], \mathcal{E}$) then
11:                 return $\Sigma[\sigma]$
12:     return $\bot$
```

The initial and final markings on the Petri net are determined by the signature $\mathcal{S}$ provided by the user. For instance, the tokens on the Petri net $\mathcal{N}$ from Figure 6 indicate the initial marking M_0 of $\mathcal{N}$. In particular, because the desired `rotate` method takes arguments of type `Area`, `Point2D`, and `double`, the initial marking assigns one token to each of these types. In addition, M_0 also assigns a single token to the special type `void`. In contrast, $M_0[\texttt{Shape}] = 0$ because `rotate` does not take any arguments of type `Shape`.

The target marking M^* of the Petri net is determined by the return type of $\mathcal{S}$. In our example, $M^*[\texttt{Area}] = 1$ because the return value of `rotate` is of type `Area`. However, for all other types τ (except for `void`), we require $M^*[\tau]$ to be 0, because this value effectively enforces that the synthesized implementation should not generate unused values. For instance, the target marking for the `rotate` example assigns `Point2D` to 0, thereby enforcing that the implementation uses argument `pt` and does not generate any other unused variables of type `Point2D`.

Reachability analysis. After constructing a Petri net $\mathcal{N}$ that models the relationships between components in Λ, we next perform reachability analysis to lazily find $\mathcal{N}$'s accepting runs (line 7 in Algorithm 1). For instance, an accepting run r for Figure 6 consists of the following sequence of transitions:

κ_D, `getX`, `getY`, `new AffineTransform`,
κ_T, `setToRotation`, `createTransformedArea`

Another accepting run r' can be obtained by replacing the transition `createTransformedArea` by `invert`. Observe that κ_D, `getX`, `getY` is *not* an accepting run because the marking obtained after this run assigns 3 tokens to `double`.

Sketch generation. Each accepting run of the Petri net $\mathcal{N}$ corresponds to a possible sequence of method calls with unknown arguments. Hence, the SKETCHGEN procedure used in line 8 of Algorithm 1 converts each reachable path π to a program sketch Σ which is then used to resolve unknown arguments. For example, consider the accepting run r of $\mathcal{N}$ that we considered earlier. This run r corresponds to the following code sketch:

```
x = #1.getX(); y = #2.getY();
t = new AffineTransform();
#3.setToRotation(#4, #5, #6);
a = #7.createTransformedArea(#8);
return #9;
```

In other words, we can convert an accepting run r to a program sketch Σ by ignoring the κ transitions and passing unknown arguments (denoted as `#i`) to each component. Furthermore, our construction guarantees that it is always possible to complete sketch Σ in a way that type-checks and satisfies certain well-formedness requirements. However, there may be multiple ways to instantiate the holes in Σ. For instance, we must assign `#1` and `#2` to `pt`, but we can assign `#4` to either `angle`, `x`, or `y`, because the only requirement is that `#4` is of type `double`.

Sketch completion. Similar to other sketching-based techniques (e.g., [45]), our technique uses a SAT solver to find possible completions of the generated program sketch. For this purpose, the SKETCHGEN procedure generates a propositional formula ϕ that encodes various semantic requirements on the generated program, including being well-typed, not containing unused variables, and having all holes filled. Specifically, our encoding introduces Boolean variables of the form $h_v^{\#i}$, which encode that hole `#i` is filled with program variable v. For example, for hole `#4`, our encoding generates the following constraint:

$$h_{angle}^{\#4} + h_x^{\#4} + h_y^{\#4} = 1.$$

This formula stipulates that hole `#4` must be filled with exactly one of `angle`, `x`, or `y` because those are the only program variables of type `double`. In addition, our encoding stipulates that each program variable must be used *at least once*. For instance, for variable angle, we generate the following constraint:

$$h_{angle}^{\#4} + h_{angle}^{\#5} + h_{angle}^{\#6} \geq 1.$$

This formula expresses that at least one of the holes #4, #5 and #6 must be instantiated with angle, because those are the only holes of type double.

After generating such a pseudo-boolean formula, we transform these constraints to CNF and use a SAT solver to find an assignment to each variable. For our running example, the following assignment σ is a model:

$$h_{pt}^{\#1} \wedge h_{pt}^{\#2} \wedge h_{t}^{\#3} \wedge h_{angle}^{\#4} \wedge \neg h_{x}^{\#4} \wedge \neg h_{y}^{\#4} \wedge \neg h_{angle}^{\#5} \wedge h_{x}^{\#5} \wedge \\ \neg h_{y}^{\#5} \wedge \neg h_{angle}^{\#6} \wedge \neg h_{x}^{\#6} \wedge h_{y}^{\#6} \wedge h_{obj}^{\#7} \wedge h_{t}^{\#8} \wedge \neg h_{obj}^{\#9} \wedge h_{a}^{\#9}$$

Observe that σ corresponds to instantiating holes #1-#9 in our code sketch with variables pt, pt, t, angle, x, y, obj, t, and a, respectively.

Validation and backtracking. Once we generate a complete program P, we then compile it and run P on the test cases provided by the user (line 10 in Algorithm 1). If all tests pass, we return P as a solution to the synthesis problem. If at least one test case fails, our algorithm backtracks and finds another satisfying assignment σ' to ϕ (if one exists) and generates a different completion of sketch Σ. If we have already considered all possible ways to fill the holes in Σ, our algorithm backtracks by finding a different accepting run of the Petri net $\mathcal{N}$ and generating a different sketch.

Discussion of design choices. A key design decision underlying our algorithm is to decompose the synthesis algorithm into two phases, namely *sketch generation* and *sketch completion*. In particular, an accepting run of the Petri net corresponds to a sequence of method calls, but there are, in general, multiple possible ways of choosing which variables to pass as arguments. We believe this decomposition between sketch generation and completion is beneficial because it allows us to perform reachability analysis on a more compact graph representation. We have considered an alternative Petri-net representation in which nodes represent parameters and return values instead of types. Under this representation, an accepting run of the Petri net can be directly translated into a code snippet rather than a sketch. However, because the corresponding Petri net is much larger, we found that the reachability problem becomes much harder, thereby making the algorithm less scalable.

5. Petri-Net Construction

We now explain in more detail how our algorithm constructs a Petri net $\mathcal{N}$ from type signatures of components. In the remainder of this paper, we assume a first-order language of type signatures with classes and built-in primitive types (string, int, etc.).[1] Given library components Λ and a desired method signature $\mathcal{S}$, the algorithm constructs $\mathcal{N} = (P, T, E, W, M_0)$ and a target marking M^* as follows:

- Places P correspond to types used in Λ.
- Transitions T represent methods in Λ. In addition, for every type $\tau \in P$, there is a special transition called κ_τ.
- Arc (τ, f) is in E and $W[(\tau, f)] = k$ if component $f \in \Lambda$ takes k inputs of type τ.
- Arc (f, τ) is in E and $W[(f, \tau)] = 1$ if f's return type is τ for some component $f \in \Lambda$.
- Arcs (τ, κ_τ) and (κ_τ, τ) are both in E. Furthermore, $W[(\tau, \kappa_\tau)] = 1$ and $W[(\kappa_\tau, \tau)] = 2$.
- $M_0[\texttt{void}] = 1$ and $M_0[\tau] = k$ if S has k inputs of type τ.
- If the return type of S is τ, then $M^*[\tau] = 1$, $M^*[\texttt{void}] \geq 0$ and $M^*[\tau'] = 0$ for all other types τ'.[2]

At a high level, the Petri-net construction outlined above views types as resources. In particular, a transition associated with component $f \in \Lambda$ "consumes" its input types and produces a token at its output type. Hence, if the desired signature $\mathcal{S}$ has type $(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau$, our goal is to produce a token at place τ by consuming the incoming tokens at places $\tau_1, \ldots, \tau_n$.

While this resource analogy fits very well with linear types, conventional types do not exactly behave as resources: In particular, invoking a component $f \in \Lambda$ on input x does not actually "consume" x; indeed, in a Java program, x can be used again. For this reason, the Petri-net construction outlined above introduces special transitions κ_τ (called *clone transitions*) that effectively allow us to "duplicate" objects of type τ. Intuitively, the number of clone transitions taken in a given run indicates the total number of times variables will be reused in the synthesized program.[3]

To illustrate the necessity of clone transitions, consider our motivating example from Section 2. Here, to synthesize the implementation of rotate, we must retrieve the x and y coordinates of point pt. However, because we initially only have one token at Point2D, we can only call getX or getY, but not both. By invoking the clone transition κ_D, we can generate two resources of type Point2D, allowing us to invoke both getX and getY on parameter pt.

Another interesting aspect of our construction is the choice of target marking M^*. First, observe that M^* assigns 0 tokens to all places other than void and the return type of $\mathcal{S}$. Intuitively, this requirement dictates that the synthesized method should use all of its inputs as well as any intermediate values that are produced. This property is desirable because a method implementation that takes x as an input but does not use x is unlikely to be correct. Furthermore, a method that produces unused variables necessarily performs redundant work and can be replaced by a simpler implementation.[4]

6. Sketch Synthesis via Petri-Net Reachability

Given a Petri net $\mathcal{N}$ with target marking M^*, we need to answer the following questions to generate a suitable code sketch:

(1) Is $M^* \in \mathcal{R}(\mathcal{N})$? If the answer to this question is negative, we know that it is not possible to synthesize well-typed code using the components we have available.

(2) If $M^* \in \mathcal{R}(\mathcal{N})$, to synthesize candidate program sketches, we must identify exactly those runs of $\mathcal{N}$ that end in M^*.

To answer these questions, we must overcome two difficulties: First, because our Petri nets are not k-safe, the state space $\mathcal{R}(\mathcal{N})$ is unbounded. While there are existing methods for answering question (1) for unsafe Petri nets [11, 28], they cannot be used for answering question (2). Second, because the number of available components may be very large, we must develop effective heuristics for pruning the search space. In the rest of this section, we describe a practical algorithm for finding reachable paths for the class of Petri nets described in Section 5.

[1] As described in Section 8, our approach also handles polymorphism, but using monomorphic instantiation.

[2] If the return type of S is void, then $M^*[\texttt{void}] \geq 0$.

[3] Our use of clone transitions is somewhat related to the use of read arcs in the Petri-net literature [51]. A read arc is a transition that does not consume tokens when fired. An alternative to having clone transitions is to use read arcs; however, this design choice would require us to use a different target marking that does not enforce the property that all inputs must be used.

[4] There are some methods, such as the add method of collections, that return a Boolean value that is often ignored. For such functions, we also consider a variant of the method that returns void.

Algorithm 2 Algorithm to construct reachability graph

```
1:  procedure ReachGraph($\mathcal{N}$, $\tau$)
2:      Input: Petri net $\mathcal{N}$, desired output type $\tau$
3:      Output: Reachability graph $\mathcal{R}^*$
4:      assume $\mathcal{N} = (P, T, E, W, M_0)$
5:      $\mathcal{R}^* := (\{M_0\}, \emptyset, M_0)$                        ▷ Initialize
6:      $\Phi := \{M_0\}$                                    ▷ Initialize worklist $\Phi$
7:      while $\Phi \neq \emptyset$ do
8:          choose $M \in \Phi$                              ▷ Process next in $\Phi$
9:          $\Phi := \Phi - \{M\}$
10:         for all $T \in$ enabled($M$) do
11:             $(M', p) :=$ fire($M, T$)                    ▷ Add successors
12:             if $\forall e \in out(p).\ M'[p] > W[e] + 1$ then
13:                 continue
14:             if $\neg$ PathExists($p, \tau, \alpha(\mathcal{N})$) then
15:                 continue
16:             if $M' \notin Nodes(\mathcal{R}^*)$ then
17:                 $Nodes(\mathcal{R}^*)$.insert($M'$)
18:                 $\Phi := \Phi \cup \{M'\}$
19:             $Edges(\mathcal{R}^*)$.insert($\langle M, T, M' \rangle$)
20:     return $\mathcal{R}^*$
```

At a high level, there are three key insights underlying our reachability algorithm. The first insight is that we can bound the search space without losing completeness in our context. That is, even though $\mathcal{R}(\mathcal{N})$ is unbounded, exploring a subset $\mathcal{R}^*(\mathcal{N})$ of $\mathcal{R}(\mathcal{N})$ is sufficient for identifying *all* accepting runs of $\mathcal{N}$ (see Section 6.2). The second key insight is to use an over-approximation $\alpha(\mathcal{N})$ of $\mathcal{N}$ to avoid exploring states that are irrelevant for reaching the target configuration M^* (see Section 6.3). Finally, rather than explicitly constructing $\mathcal{R}^*(\mathcal{N})$, we encode it symbolically and lazily enumerate the "most-promising" accepting runs of $\mathcal{N}$ by solving an optimization problem (see Section 6.4).

6.1 Basic Reachability Algorithm

Our algorithm for constructing the reachability graph $\mathcal{R}^*(\mathcal{N})$ is presented as pseudo-code in Algorithm 2. We first consider a basic version of the algorithm without lines 12–15, which is roughly equivalent to the standard algorithm for constructing $\mathcal{R}(\mathcal{N})$. The additional lines 12–15 correspond to our customization, and allow us to construct $\mathcal{R}^*(\mathcal{N})$ instead of $\mathcal{R}(\mathcal{N})$.

The procedure ReachGraph shown in Algorithm 2 takes as input a Petri net $\mathcal{N}$ with initial marking M_0 and the return type τ of the method we would like to synthesize, and returns a *reachability graph* $\mathcal{R}^*$. The nodes of $\mathcal{R}^*$ correspond to markings of $\mathcal{N}$, and a (directed) edge $\langle M, T, M' \rangle$ indicates that we can reach marking M' from M by firing transition T of $\mathcal{N}$. We denote nodes of $\mathcal{R}^*$ using labels of the form $\langle k_1, \ldots, k_n \rangle$, which indicates that there are k_i tokens at place P_i. For example, the marking of the Petri net from Figure 4 corresponds to the node label $\langle 2, 0, 0 \rangle$, whereas the marking from Figure 5 is given by $\langle 1, 1, 0 \rangle$.

The loop in lines 7–19 of Algorithm 2 iteratively constructs $\mathcal{R}^*$ starting from initial marking M_0. In particular, the worklist Φ contains all reachable markings that have not yet been processed. Initially, the only reachable marking is M_0; hence we initialize Φ to the singleton set $\{M_0\}$ at line 6. In each iteration of the loop, we compute the successor states of some marking M in Φ by firing its enabled transitions. Specifically, the procedure fire used at line 11 takes a marking M and a transition T and returns the resulting marking M', as well as the output place p of transition T.[5] Now, ignoring lines 12–15, we add the edge $\langle M, T, M' \rangle$ to our reachability graph $\mathcal{R}^*$ and insert M' into the worklist if it has not already been processed.

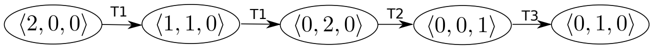


Figure 7. Reachability graph for Petri net from Figure 4

Example 3. *Figure 7 shows the reachability graph for the Petri net from Figure 4. Observe that feasible runs of $\mathcal{N}$ correspond to paths starting with M_0 in the reachability graph. Hence, using the reachability graph, we immediately see that T_1, T_1, T_2 is a feasible run, but T_1, T_2, T_3 is not.*

6.2 Ensuring Termination

As mentioned earlier, the construction outlined in Section 5 results in Petri nets that are not k-safe for any k. In particular, while the clone transitions κ_τ are necessary for synthesizing code that reuses the same variable multiple times, they also cause us to accumulate arbitrarily many tokens at a given place. For example, we can obtain an unbounded number of tokens at place `Point2D` of Figure 6 by taking the clone transition κ_D as many times as we want. As a result, the size of the reachability graph is unbounded, meaning that the basic reachability algorithm from Section 6.1 will not terminate.

Fortunately, it turns out that we can bound the size of the reachable state space without losing completeness. In particular, when constructing the reachability graph for Petri net $\mathcal{N}$, we can safely ignore markings that assign more than $k + 1$ tokens to a place p, where k denotes the maximum weight of any outgoing edge of p.[6] To see why we can ignore such markings, observe that no transition in $\mathcal{N}$ can be disabled due to p as long as we have at least k tokens at p. Furthermore, no matter what transition we take from the current marking, p will have at least 1 remaining token. Because our Petri nets contain clone transitions for every place, we can always produce k tokens at p by taking the clone transition sufficiently many times, as long as we have at least 1 token at p.

To formalize this intuition, let "$paths_{[M_0, M^*]}(G)$" denote the set of transition sequences in some reachability graph G that start at initial marking M_0, end at target M^*, and ignore all clone transitions. We can now state the following theorem:[7]

Theorem 1. *Let $\mathcal{R}(\mathcal{N})$ be the reachability graph constructed by the basic algorithm of Section 6.1, and let $\mathcal{R}^*(\mathcal{N})$ be the reachability graph constructed by employing lines 12–15 of Algorithm 2. If $p \in paths_{[M_0, M^*]}(\mathcal{R}(\mathcal{N}))$, then $p \in paths_{[M_0, M^*]}(\mathcal{R}^*(\mathcal{N}))$.*

Effectively, this theorem states we do not "lose" any valid code sketches by considering the paths of $\mathcal{R}^*(\mathcal{N})$ instead of $\mathcal{R}(\mathcal{N})$. Furthermore, because the size of $\mathcal{R}^*(\mathcal{N})$ is bounded by n^{k+1} where n is the number of places and k is the maximum edge weight in $\mathcal{N}$, Algorithm 2 is guaranteed to terminate. However, because places in $\mathcal{N}$ correspond to classes defined by a library, the reachability graph can still be very large. In the next subsection, we describe a pruning strategy to further reduce the size of the reachability graph.

[5] In our context, each transition has exactly one outgoing edge because every component has exactly one return type.

[6] For simplicity, we assume that the number of initial tokens at place p is less than or equal to $k + 1$. If this assumption is violated, the upper bound is given by the maximum of $k + 1$ and the number of initial tokens.

[7] Proofs of all theorems are given in the extended version of the paper [8].

6.3 Pruning using Graph Reachability

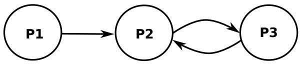
Another key idea of our algorithm is to use standard graph reachability to overapproximate Petri-net reachability. In particular, consider a place τ' in the Petri net that is not backwards reachable from our target type τ. Because there is no path from τ' to τ in $\mathcal{N}$, it is unnecessary to consider markings where τ' contains a non-zero number of tokens. Line 14 in Algorithm 2 exploits this observation to prune redundant nodes of $\mathcal{R}(\mathcal{N})$.

To make this discussion more precise, let us define $\alpha(\mathcal{N})$ to be the graph *induced by* Petri net $\mathcal{N}$ as follows:

Definition 2. (Induced graph) *Let $\mathcal{N} = (P, T, E, W, M_0)$ be a Petri net. The graph induced by $\mathcal{N}$, denoted $\alpha(\mathcal{N})$, is a directed graph (V, E') where $V = P$ and $(P, P') \in E'$ iff there is a transition $f \in T$ such that $(P, f) \in E$ and $(f, P') \in E$.*

In other words, $\alpha(\mathcal{N})$ includes an edge between two places P, P' if it is possible to reach P' from P by firing a single transition.

Example 4. *The graph induced by the Petri net of Figure 4 is shown below:*



Theorem 2. *Let $\mathcal{N}$ be a Petri net with no path from τ' to τ in $\alpha(\mathcal{N})$. Let M^* be the target marking that assigns one token to target type τ, and let M be a marking such that $M(\tau') > 0$. Then, there is no path from M to M^* in $\mathcal{R}(\mathcal{N})$.*

According to this theorem, if a marking M assigns a non-zero value to any place τ' that is not backwards-reachable from τ in $\alpha(\mathcal{N})$, then there is no path from M to M^* in $\mathcal{R}(\mathcal{N})$. Hence, we can prune such a marking M without affecting completeness. Line 14 in Algorithm 2 takes advantage of this fact by only adding M' to $\mathcal{R}^*(\mathcal{N})$ if p is backwards reachable from τ.

6.4 Symbolic Encoding using ILP

So far, our algorithm explicitly constructs $\mathcal{R}^*(\mathcal{N})$ and enumerates all paths of $\mathcal{R}^*(\mathcal{N})$. However, because $\mathcal{R}^*(\mathcal{N})$ can have many accepting paths, this strategy is suboptimal. Instead, a better alternative is to encode this problem symbolically and *lazily* generate accepting runs of $\mathcal{N}$ in order of increasing cost. Toward this goal, we formulate the problem of finding an accepting run of $\mathcal{N}$ as a 0-1 Integer Linear Programming (ILP) problem and obtain the "most-promising" path by minimizing a heuristic objective function.

Our lazy symbolic path-enumeration algorithm is presented in Algorithm 3. We consider accepting runs of $\mathcal{N}$ in increasing order of length, starting from the minimum bound k (line 6). In particular, if τ_i is one of the input types and τ is the desired output type, then any accepting run of $\mathcal{N}$ must contain at least as many transitions as the shortest path between τ_i and τ in $\alpha(\mathcal{N})$; hence, we do not need to look for accepting runs below this threshold.

Now, given a target length k, we symbolically encode the k-reachability problem of $\mathcal{N}$ as a propositional formula ϕ. In particular, formula ϕ from line 8 is satisfiable if and only if there exists an accepting run of $\mathcal{N}$ of length k. Our symbolic encoding is similar to previous SAT-based encodings of Petri nets [22, 31, 34], but we make use of the observations from Sections 6.2 and 6.3. While a full discussion of our symbolic encoding is beyond the scope of this paper, we refer the interested reader to the extended version of the paper [8].

Algorithm 3 Lazy symbolic path enumeration

```
 1: procedure LazyPathGen($\mathcal{N}$, $\tau_1, \ldots, \tau_n$, $\tau$)
 2:     Input: Petri net $\mathcal{N}$, input types $\tau_1, \ldots, \tau_n$,
 3:            output type $\tau$
 4:     Output: An accepting run $t$ of $\mathcal{N}$ if one exists
 5:     $\pi_i$ := ShortestPath($\alpha(\mathcal{N}), \tau_i, \tau$)        ▷ Lower bound
 6:     $k$ := max(length($\pi_1$), ..., length($\pi_n$))
 7:     while true do
 8:         $\phi$ := Encode($\mathcal{N}, k$)             ▷ Unfolding of length $k$
 9:         $\psi$ := true
10:         while true do
11:             $\sigma$ := Minimize($\Sigma_i c_i x_i,\ \phi \land \psi$)
12:             if $\sigma = \bot$ then
13:                 break
14:             if Check($\sigma$) then
15:                 return Trace($\sigma$)
16:             $\psi$ := $\psi \land$ Block($\sigma$)
17:         $k := k + 1$
18:     return $\bot$
```

The inner loop in lines 10–16 of Algorithm 3 lazily enumerates paths of length k in order of increasing cost, where the cost is determined by some heuristic evaluation function. To generate the "most-promising" path, we solve an ILP problem with objective function $\Sigma_i c_i x_i$ (line 11). Here, x_i is a variable that is assigned to 1 by our encoding if and only if component T_i is used in the accepting run and to 0 otherwise. The costs c_i used in the objective function reflect the likelihood of component T_i being used in the synthesized code—i.e., the smaller the c_i, the more likely it is that component T_i is useful. While there are many possible heuristics for assigning costs to components, our current implementation uses a similarity metric between the name of the desired method and the documentation and name of each library component.[8] Going back to our running example from Section 2, this methodology assigns a lower cost to a component called `setToRotate` compared to another component called `invert` because the former component is likely to be more "similar" to the desired `rotate` method.

Once we obtain a satisfying assignment σ of ϕ that minimizes our heuristic objective function, we ask an "oracle" to confirm or refute it (lines 14–15). In this context, the oracle completes the code sketch given by σ (see Section 7) and runs the test cases. If σ does not correspond to a satisfactory code sketch, we need to "block" this assignment in future iterations by adding a blocking clause ψ. In the simplest case, a blocking clause can be obtained as the negation of σ; however, our algorithm generates a stronger blocking clause by performing a particular form of partial-order reduction [3, 36] on the current path p. In particular, if p contains two consecutive calls to methods f and g that cannot be called with the same arguments, then our algorithm also blocks variants of this path where calls to f and g have been re-ordered.

7. Code Synthesis from Paths

Given an accepting run r of the Petri net described in Sections 5 and 6, to synthesize a suitable program from r, we still need to perform the following tasks:

(a) Use the transitions in r to create a code sketch Σ

(b) Fill the holes in Σ with program variables

[8] We refer the interested reader to the extended version of the paper [8] for a more detailed discussion of our similarity metrics.

Each transition in r corresponds to either an invocation of a method foo from an API or a special κ transition. When synthesizing code, we ignore clone transitions and only consider API calls. In particular, if some API method foo used in r has n input parameters, the code sketch for foo's invocation looks like the following:

```
// if m is a virtual method
T_o out = #1.foo(#2, #3, #4, ..., #n+1)

// if m is a static method or constructor
T_o out = foo(#1, #2, #3, ..., #n)
```

In general, if trace r is of length l and contains k clone transitions, the corresponding synthesized program contains $l - k + 1$ lines, where the first $l - k$ lines correspond to API calls and the last line is a return statement of the form `return #m` (when the program does not return `void`).

Now, given sketch Σ, we need to instantiate each hole with a program variable. To achieve this goal, we generate a propositional formula ϕ that encodes well-formedness requirements. In particular, our encoding introduces Boolean variables $h_v^{\#i}$ that are true when program variable v is used to fill hole #i. To ensure type compatibility, we only introduce Boolean variable $h_v^{\#i}$ if the type of program variable v matches the type of hole #i. Furthermore, because a program variable cannot be used before it is defined, we only introduce $h_v^{\#i}$ if v is a parameter or the result of an invocation that appears before hole #i.

While our construction of the Boolean variables guarantees that the holes will be filled in a type-compatible way, we still have to ensure that no hole remains empty and that all variables are used. Let V be the set of all program variables and H the set of all holes in Σ. Let $getV$ be a function that receives V and a hole h and returns $V' \subseteq V$, where V' corresponds to all program variables that can be placed in hole h. Similarly, let $getH$ be a function that receives H and a variable $v \in V$ and returns $H' \subseteq H$, where H' corresponds to all holes where v can be placed. Using these definitions, we generate a formula ϕ as follows:

(1) Each hole is filled with one program variable:
$$\forall_{\#i \in H} \forall_{v \in getV(V,\#i)} \sum h_v^{\#i} = 1$$
(2) Each program variable is used at least once:
$$\forall_{v \in V} \forall_{\#i \in getH(H,v)} \sum h_v^{\#i} \geq 1$$

Example 5. *Consider the code sketch in Section 4. From requirement (1), we generate the following constraints:*

$$h_{pt}^{\#1} = 1 \; ; \; h_{pt}^{\#2} = 1 \; ; \; h_{t}^{\#3} = 1 \; ; \; h_{angle}^{\#4} + h_{x}^{\#4} + h_{y}^{\#4} = 1$$
$$h_{angle}^{\#5} + h_{x}^{\#5} + h_{y}^{\#5} = 1 \; ; \; h_{angle}^{\#6} + h_{x}^{\#6} + h_{y}^{\#6} = 1$$
$$h_{obj}^{\#7} = 1 ; h_{t}^{\#8} = 1 \; ; \; h_{obj}^{\#9} + h_{a}^{\#9} = 1$$

Similarly, from requirement (2), we generate the constraints:

$$h_{pt}^{\#1} \geq 1 \; ; \; h_{pt}^{\#2} \geq 1 \; ; \; h_{t}^{\#3} \geq 1$$
$$h_{angle}^{\#4} + h_{angle}^{\#5} + h_{angle}^{\#6} \geq 1 \; ; \; h_{x}^{\#4} + h_{x}^{\#5} + h_{x}^{\#6} \geq 1$$
$$h_{y}^{\#4} + h_{y}^{\#5} + h_{y}^{\#6} \geq 1 \; ; \; h_{obj}^{\#7} + h_{obj}^{\#9} \geq 1 \; ; \; h_{a}^{\#9} \geq 1$$

Because each satisfying assignment σ to ϕ corresponds to a well-typed completion of sketch Σ, we can now run the user-provided test cases on $\Sigma[\sigma]$. If any test fails, we then obtain a different instantiation of the sketch by obtaining a model of $\phi \wedge \neg\sigma$ in the next iteration.

8. Implementation

We have implemented our synthesis algorithm as a new tool called SyPet, which consists of approximately 10,000 lines of Java code. SyPet uses the Sat4j [5] tool for solving SAT problem, and can be instantiated with any Java API (or combinations of APIs) to synthesize straight-line Java code. Soot [50] is used to parse the .jar files of the libraries and extract the signatures of classes and methods, which will be converted to places and transitions in the Petri-net, respectively.

Because many Java libraries use parametric polymorphism, our implementation also supports generic types. Our handling of polymorphism is similar to template instantiation in C++. For instance, given a polymorphic type of the form `Foo<? extends A>` and subclasses `B`, `C` of `A`, we generate three different copies of type `Foo`, namely `FooA`, `FooB`, and `FooC`, each of which corresponds to a different place in the Petri net. We also handle polymorphic methods in a similar way and create different transitions for each instantiation of a polymorphic API component.

As mentioned in Section 6, SyPet uses a symbolic encoding of the Petri-net-reachability problem, but our implementation differs from Algorithm 3 in one small way. Given a Petri net $\mathcal{N}$, recall that Algorithm 3 explores all reachable paths of length k before moving on to paths of length $k + 1$. While this approach simplifies our presentation, it is not a very good implementation strategy: Because there can be *many* paths of length k , we have found that a better strategy is to explore different path lengths in a round-robin fashion. In particular, our search strategy is parametrized by two integers n, m: Given a starting path length k, we first explore m paths of size k, and then move on to paths of length $k + 1$. After exploring m paths each of length $k, \ldots, k + n$, we go back to exploring paths of length k. In our current implementation, we use the values 2 and 100 for n and m, respectively.

9. Evaluation

To evaluate SyPet, we performed experiments that were designed to answer the following questions:

1. How well does SyPet perform on component-based synthesis tasks that involve Java APIs?
2. How many test cases does the user typically need to supply for SyPet to succeed?
3. How complex are the programs synthesized by SyPet?
4. How does SyPet's success rate compare with other tools for component-based synthesis?

To answer these questions, we collected six widely-used Java APIs: a math library (`apache.commons.math`), a geometry library (`java.awt.geom`), a time/date library (`joda-time`), and text and XML-related libraries (`jsoup`, `w3c.dom` and `javax.xml`). In addition to being widely used, these libraries are reasonably large, containing 50–1215 classes and 751–9578 methods. The average number of classes and components in each library is 528 and 4721, respectively.

For each of these APIs, we collected a set of programming tasks that require non-trivial interaction between different classes. Our programming tasks come from two sources—namely, online forums like *stackoverflow* and existing Github repositories. For the former category, we manually curated common questions that programmers typically ask about the relevant API. For the latter category, we wrote a script to crawl over Github projects and filter straight-line methods that use one of the aforementioned APIs. A brief summary of each programming task is provided under the "Description" column in Figure 8.

Lib	ID	Description	Synthesis Time (s)	#Paths	#Progs	#Tests	#Comps	#Holes
apache math	1	Compute the pseudo-inverse of a matrix	6.78	255	509	1	3	4
	2	Compute the inner product between two vectors	0.25	1	1	1	3	5
	3	Determine the roots of a polynomial equation	0.64	7	13	1	3	5
	4	Compute the singular value decomposition of a matrix	0.16	1	1	1	3	4
	5	Invert a square matrix	0.63	16	31	1	3	4
	6	Solve a system of linear equations	28.25	790	1,605	1	6	8
	7	Compute the outer product between two vectors	2.12	14	48	1	4	6
	8	Predict a value from a sample by linear regression	2.56	25	51	2	5	5
	9	Compute the ith eigenvalue of a matrix	164.60	3,197	7,636	2	6	8
geometry	10	Scale a rectangle by a given ratio	1.37	78	271	1	4	7
	11	Shear a rectangle and get its tight rectangular bounds	1.76	79	280	1	4	7
	12	Rotate a rectangle about the origin by the specified number of quadrants	0.32	9	21	1	4	6
	13	Rotate two dimensional geometry object by the specified angle about a point	2.01	67	226	2	5	8
	14	Perform a translation on a given rectangle	0.72	41	150	1	4	7
	15	Compute the intersection of a rectangle and the rectangular bounds of an ellipse	0.08	1	1	1	3	5
joda	16	Compute number of days since the specified date	4.55	78	156	2	3	4
	17	Compute the number of days between two dates considering timezone	174.16	774	4,736	3	4	6
	18	Determine if a given year is a leap year	35.32	306	613	3	4	5
	19	Return the day of a date string	0.74	1	1	2	3	5
	20	Find the number of days of a month in a date string	35.23	175	531	2	4	6
	21	Find the day of the week of a date string	47.27	126	376	2	4	6
	22	Compute age given date of birth	7.90	142	288	3	3	4
jsoup, dom, text	23	Compute the offset for a specified line in a document	0.31	3	5	1	3	5
	24	Get a paragraph element given its offset in the a document	1.14	33	65	1	4	6
	25	Obtain the title of a webpage specified by a URL	10.29	277	553	1	3	4
	26	Return doctype of XML document generated by string	0.87	9	17	1	6	7
	27	Generate an XML element from a string	0.89	26	51	1	6	7
	28	Read XML document from a file	0.11	1	1	1	3	4
	29	Generate an XML from file and query it using XPath	16.33	20	44	1	7	10
	30	Read XML document from a file and get the value of root attribute specified by a string	0.29	3	5	1	5	7

Figure 8. Summary of experimental results

9.1 SyPet Performance

Setup. To evaluate SyPet on these programming tasks, we provided a signature of the desired method as well as one or more test cases. We also specify which libraries are used for each programming task, e.g., `joda.time`, `apache.commons.math`, etc. However, it is easy to configure the tool to use any set of libraries. For the benchmarks taken from Github, we used the existing method signature (and test cases if available). For most *stackoverflow* benchmarks, method signature and test cases were not available in the forum discussion, so we wrote them ourselves. For all benchmarks, we initially provided a *single* test case and used SyPet to synthesize an implementation that works on that test case. We then manually inspected the synthesized code and provided an additional test case if the synthesized code did not perform the desired functionality. We then repeated this process until the code produced by SyPet met our expectations.

The results of our evaluation are summarized in Figure 8 (For more detailed results, please refer to the extended version of the paper [8]). All experiments are conducted using Oracle HotSpot JVM 1.7.0_75 on an Intel Xeon(R) computer with an E5-2640 v3 CPU and 32G of memory, running Ubuntu 14.04.

Performance and statistics. As shown in the "Synthesis Time" column of Figure 8, SyPet can successfully synthesize all benchmarks in an average of 2.33 seconds.[9] Note that the synthesis time neither includes compilation time nor the overhead of parsing the .jar files with Soot. Compilation has an average overhead of 53% on the running time and Soot takes an average of 7.00 seconds to parse the Java libraries. The "#Paths" column indicates the total number of code sketches generated by our tool. Note that this number is equivalent to the number of explored paths (accepting runs) of the Petri net. On average, SyPet explores 29 different code sketches before it identifies the correct sequence of method calls. Furthermore, each iteration of the tool is quite fast; SyPet finds an accepting run of the Petri net in 0.08 seconds on average. The column labeled "#Progs" indicates the total number of programs generated by SyPet before finding the correct program. On average, SyPet explores 61 programs before generating an implementation that performs the desired functionality.

While SyPet synthesizes 73% of the benchmarks in $<$ 10 seconds and 93% in $<$ 60 seconds, a few benchmarks (e.g., 9 and 17) take longer. We have manually inspected these outliers and found that the user-provided signatures for these examples match the signature of many API components. Hence, SyPet ends

[9] If there are multiple rounds of user interaction to create additional test cases, we report statistics for the last one. We calculate averages using geometric mean.

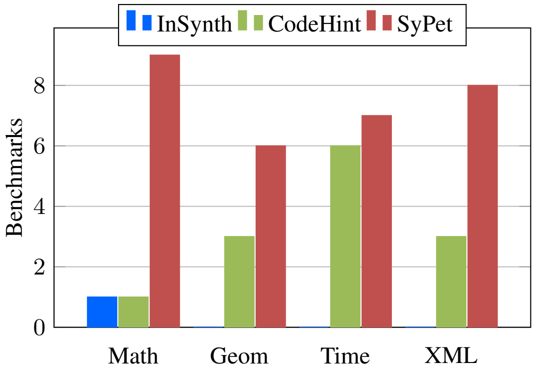


Figure 9. Comparison with other tools

up exploring hundreds of code sketches before it synthesizes the intended one.

Usability. In addition to successfully synthesizing the desired code in a reasonable amount of time, we also see that SyPet does not require many test cases from the user. In particular, as shown under the "#Tests" column in Figure 8, SyPet requires 1 test case on average, with the maximum number of test cases being 3.

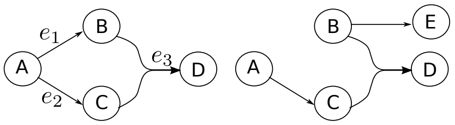
Synthesized programs. The "#Comps" and "#Holes" columns in Figure 8 provide information about the synthesized programs. In particular, "#Comps" reports the number of components in the code sketch (in terms of the length of the accepting run), and "#Holes" indicates the number of holes. The average synthesized program contains 4 components and 6 holes. These statistics reinforce our earlier claim that SyPet combines the practicality of API completion tools with the power of synthesis tools: While programs synthesized by SyPet are moderately sized, straight-line code fragments, SyPet can handle two orders of magnitude more components than previous synthesis tools [15, 16, 26, 38]. On the other hand, while API-completion tools [12, 19, 20, 30] can handle thousands of components, they can typically only suggest very small (single-line) code snippets.[10]

9.2 Comparison with Other Tools

To validate our claim that SyPet compares favorably with existing synthesis tools that do not require logical specifications, we also compare SyPet with CodeHint and InSynth. CodeHint is a state-of-the-art type-based synthesis tool, and, similar to SyPet, it takes as input a method signature and test case. In contrast, InSynth is a type-directed API-completion tool that can synthesize expressions of a given type.

The results of our comparison are provided in Figure 8, which shows how many benchmarks were synthesized by each tool within a 30-minute time limit. For both CodeHint and InSynth, we consider the synthesis task to be successful if the correct implementation is among any of the suggested code snippets. While SyPet is able to synthesize all 30 benchmarks, CodeHint synthesizes 13 benchmarks and InSynth can synthesize just one of them.

Because InSynth is mainly intended to be used as a single-line code-completion tool, we also performed a second (simpler) experiment using InSynth. Specifically, given the full implementation of each benchmark except a single line of code, we tried to use InSynth to complete the right-hand-side of each assignment one at a time. We considered InSynth to be successful if it was able to complete the right-hand-side of all assignments used in the implementation. However, even for this easier task, InSynth was only able to solve 14 out of the 30 benchmarks.



(a) A B-path (b) Not a B-path

Figure 10. Hypergraph examples

10. Design Choices and Comparison with Hypergraph-Based Solutions

As mentioned in Section 1, the synthesis algorithm underlying SyPet can be seen as a generalization of the algorithm used in Prospector, which employs standard graph-reachability analysis to perform API completion [30]. Specifically, given a source type τ_{in} and a target type τ_{out}, Prospector constructs a graph in which nodes represent types and an edge labeled f from τ to τ' indicates that f is a unary function of type $\tau \rightarrow \tau'$. Hence, a path from τ_{in} to τ_{out} corresponds to a sequence of method calls that can be used to solve the synthesis problem defined by (τ_{in}, τ_{out}).

SyPet solves a more general synthesis problem than Prospector because the underlying components do not have to be unary functions. Moreover, to tackle the complexities that arise from this generalization, SyPet uses a more powerful graph representation, namely Petri nets. However, because Petri-net-reachability analysis is a hard problem (PSPACE complete), the reader may wonder whether Petri nets are overkill and whether some other internal representation might be more suitable. While it should be clear that standard graphs are not sufficient for faithfully representing multi-argument functions, one obvious alternative is to use *directed hypergraphs* instead of Petri nets. We have carefully considered this alternative, and, in this section, we explain why we believe Petri nets are a better match for this problem than hypergraphs, both in terms of expressiveness as well as overall scalability of synthesis.

Background on hypergraphs. *Hypergraphs* generalize graphs by allowing edges that can connect any number of vertices. Specifically, a *directed hypergraph* G is a pair (V, E) where V is a set of vertices, and E is a set of *hyperedges*. A hyperedge is a pair (T, H) where *tail* T and *head* H are subsets of V. A *B-hyperedge* is a special kind of edge where the head H is a singleton. Hypergraphs that only contain B-hyperedges are called *B-hypergraphs*.

Example 6. *Figure 10 shows two B-hypergraphs. In Figure 10(a), e_3 is a hyperedge with tail $\{B, C\}$ and head $\{D\}$. Intuitively, to reach node D, nodes B and C must both be reachable.*

Definition 3. (Simple path) *A simple path $v_0 \rightsquigarrow v_n$ is a sequence $v_0, e_1, v_1 \ldots e_n, v_n$ such that $v_i \in \mathsf{head}(e_i)$, $v_i \in \mathsf{tail}(e_{i+1})$ and each hyperedge e_i is distinct.*

For example, in Figure 10(a), A, e_1, B, e_3, D is a simple path.

Definition 4. (B-path) *Given B-hypergraph G, a B-path P from node s to node t is a minimal subgraph (V_P, E_P) such that $s, t \in V_P$ and $\forall v \in V_P - \{s\}$, there exists a simple path $s \rightsquigarrow v$ in P.*

In this definition, if an edge e is chosen to be in E_P, then *head*(e) and *tail*(e) must also be part of V_P.

Example 7. *The graph in Figure 10(a) is a B-path. In contrast, Figure 10(b) is not a B-path for two reasons: First, there is no*

[10] For instance, 94% of the benchmarks used in evaluating InSynth [19, 20] (a state-of-the-art completion tool) involve a single API call.

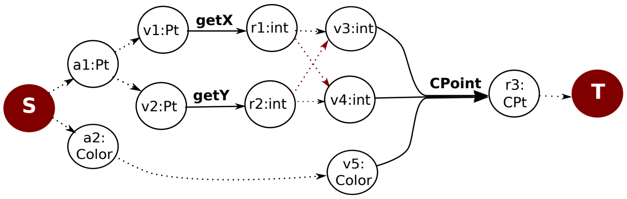


Figure 11. Hypergraph for Example 8

simple path from A to B. However, even if we add an edge e with tail A and head B, this graph would still not be a B-path because it is not minimal (there is an extra edge from B to E).

10.1 Using Hypergraphs for Synthesis

In this section, we describe an alternative solution based on hypergraphs for solving the component-based synthesis problem.

The key idea is to construct a B-hypergraph where nodes represent parameters and return values, and B-edges represent function calls (or assignments). Specifically, each function f corresponds to a B-hyperedge, where the tail includes f's parameters and the head is the singleton representing f's return value. In addition, there is an edge from every return node of type τ to all parameter nodes of type τ. The latter class of edges allow us to express that the return value of one procedure may feed as input to another procedure.

Example 8. *To illustrate this construction, let us consider the following very simple API with classes* `Point` *and* `CPoint`*:*

```
class Point { int getX(); int getY(); }
class CPoint { CPoint(int x, int y, Color c); }
```

Figure 11 shows the hypergraph we construct for this synthesis problem (for now, ignore the red nodes labeled S and T, and the initial argument nodes $a1$ and $a2$). Here, the constructor for `CPoint` *corresponds to a hyperedge whose tail has three elements, namely* `v3`, `v4` *and* `v5`*, representing its arguments. The dashed edges represent possible flows from return values of one function to the arguments of another. For example, there is an edge from* `r1` *to* `v3` *and* `v4` *because the return value of* `getX` *has the same type as the first two argument of the* `CPoint` *constructor.*

Now, given such a hypergraph G and the signature for target function f, we obtain a final graph G' by adding two special source and target nodes, namely S and T, to G. Additionally, for each argument of type τ in f, we create a node that corresponds to that argument and add an edge from that node to all parameter nodes of type τ (as well as parameters of type void). Similarly, if f has return value τ', we then add an edge from all other return nodes of type τ' to T. Finally, to solve the synthesis problem defined by hypergraph G', we find a B-path from S to T and translate this path into a sequence of method calls.

Example 9. *Suppose that we want to synthesize a function* `makeCPoint`*, which takes an argument of type* `Point` *and another argument of type* `Color`*. Figure 11 shows the corresponding hypergraph for this synthesis problem. Note that this graph contains a B-path from S to T, which we can obtain by deleting the red dashed edges (between* `r1`, `v4` *and* `r2`, `v3`*).*

10.2 Problems with the Hypergraph Approach

At first glance, the strategy outlined in Section 10.1 may seem appealing for multiple reasons: First, there exist polynomial algorithms for finding a B-path in a hypergraph [13, 29, 33]; hence, it is tempting to conclude that the hypergraph approach is more scalable compared to our Petri-net-based algorithm. Second, the hypergraph solution does not require two separate sketch generation and completion phases; thus, the algorithm seems conceptually simpler. Despite these apparent advantages of hypergraphs, we now point out some serious drawbacks of this approach.

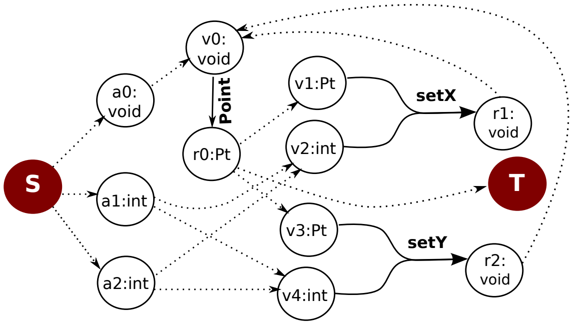


Figure 12. Hypergraph for Example 10

Advantages of Petri nets in theory. First, it turns out that the strategy discussed in Section 10.1 is significantly less expressive compared to the algorithm based on Petri nets. In particular, there are many programs that can be synthesized by our approach, but not using the hypergraph-based algorithm.

Example 10. *Consider a* `Point` *API, which has an empty constructor, as well as two setter methods,* `setX` *and* `setY`*. Suppose that we want to implement a method called* `makePoint`*, which takes two integers,* `x` *and* `y`*, and returns a* `Point`*. Clearly, we can implement this method using this API as follows:*

```
Point makePoint(int x, int y) {
  Point p = new Point();
  p.setX(x); p.setY(y);
  return p; }
```

However, this simple program cannot be synthesized using the approach described in Section 10.1. To understand why, consider the hypergraph from Figure 12, which shows the hypergraph associated with this synthesis problem. Note that there is no B-path from S to T that involves the `setX` *and* `setY` *methods.*

As this example illustrates, the hypergraph approach outlined in Section 10.1 does not work well when the underlying components have side effects. The reader may be tempted to work around this problem by pretending that setter methods return the receiver object. Unfortunately, this work-around solution creates additional difficulties: First, one would need to statically analyze the underlying components to determine which parameters are modified. However, since the implementation of the API may be quite complex, we believe this strategy is unrealistic. Second, even if this kind of information was available, the hypergraph representation would no longer be a B-hypergraph (since functions can now have multiple "return values"). As a result, the corresponding reachability problem would now become much harder.

In addition to facing difficulties in the presence of impure components, the hypergraph-based solution also has other limitations. For example, the solution outlined in Section 10.1 can also *not* be used to synthesize methods that call the same procedure twice.

Example 11. *Consider an API with a* `Point` *constructor, which takes two integers, and a* `distance` *method, which computes the distance between two points. The following implementation of* `computeDist` *cannot be synthesized by the hypergraph approach, because it requires calling the* `Point` *constructor twice:*

```
int computeDist(int x1, int y1, int x2, int y2) {
  Point p1 = new Point(x1, y1);
  Point p2 = new Point(x2, y2);
  return p1.distance(p2);
}
```

Finally, in addition to *not* being able to generate many *valid* programs, the hypergraph approach also generates many *redundant* programs that have little chance of being correct. Because the hypergraph approach does not enforce that *all* inputs are used, many redundant programs must be compiled and checked against the provided test cases. [11]

Advantages of Petri nets in practice. So far, we argued that the Petri-net approach has significant advantages over the hypergraph approach in theory. Naturally, the reader may wonder if these limitations actually matter in practice. To answer this question, we also implemented the hypergraph-based algorithm, and provide an empirical comparison between the two algorithms on the benchmarks from Section 9. For our hypergraph-based implementation, we use the *halp* package [4], which can be used to enumerate k shortest hyperpaths. Specifically, the *halp* package implements a polynomial-time algorithm [33] for finding the k shortest B-paths in a hypergraph G. The complexity of this algorithm is known to be $O(kn(m\,log\,n + size(G)))$.[12]

Figure 13 compares the number of benchmarks that can be solved within 30 minutes by the hypergraph approach with those that can be solved within the same time limit by SyPet. As shown in Figure 13, SyPet can synthesize all 30 benchmarks, while the hypergraph approach can only synthesize 8 benchmarks. Furthermore, even when we restrict ourselves to the 8 benchmarks that can be solved by both approaches, SyPet's average synthesis time is 2.1 seconds, while the algorithm based on hypergraphs requires 355.9 seconds.

Further discussion. The reader may have noticed the discrepancy between our Petri-net representation, where nodes correspond to types, and the hypergraph formalism, where nodes represent parameters and return values. A natural question to consider is whether it is possible to consider a more compact hypergraph representation where nodes represent types. While this is possible, the alternate, more compact representation would be a kind of *"hyper-multigraph"* where we have multiple edges between a pair of nodes. For example consider a function f that takes two integers and returns a string. In this case, we would have a hyperedge whose tail is the multi-set { `int, int` }. Furthermore, if there is another function g with the same signature as f, then there would be multiple hyperedges between the node `int` and `string`. We chose to present the representation from Section 10 because we believe it is simpler and easier to understand. Nevertheless, the more compact hyper-multigraph representation still suffers from the same issues that we discussed in Section 10.2 in addition to new challenges (e.g., a path in this representation no longer corresponds to a unique sketch, but a set of possible sketches because a path does not necessarily impose a total order on the sequence of calls).

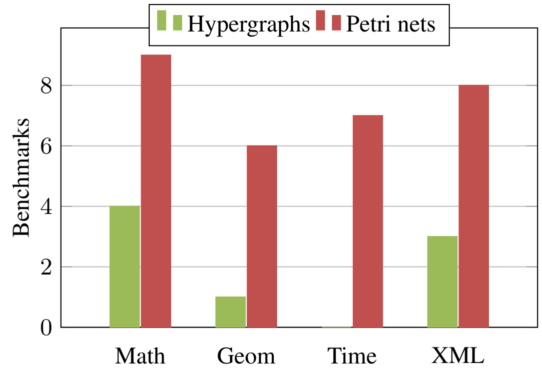


Figure 13. Comparison between hypergraphs and Petri nets

11. Related Work

SyPet is related to a long line of work on program synthesis and API completion. Here, we survey approaches that are most closely related to ours.

Component-based synthesis. Component-based synthesis typically refers to the generation of loop-free code from a database of available "components." Such techniques have been used in a variety of applications, including bit-vector algorithms [15], deobfuscators [26], geometry constructions [16], and string and data-structure transformations [9, 38]. Most of these approaches require logical specifications of the underlying components, which are often not available for real-world Java APIs. While some of these systems (e.g., DBS [38]) can synthesize more complex programs than SyPet, our approach has the advantage of being able to handle orders of magnitude more components.

Among synthesis tools, SyPet is most closely related to CodeHint [12]. Similar to SyPet, CodeHint can also handle real-world Java APIs and utilizes user-provided test cases. However, unlike SyPet and most other synthesis tools, CodeHint synthesizes and evaluates code at *run time* and uses a probabilistic model to guide the search towards expressions that are more often used in practice. Because of CodeHint's similarity to SyPet in terms of its user-facing interface, we were able to empirically compare CodeHint with SyPet. Our evaluation in Section 9 shows that SyPet can synthesize a larger set of programs than CodeHint.

API completion. Code completion refers to the generation of small code snippets involving API calls [18–20, 24, 30, 37, 39, 41, 49, 53]. While the line between component-based synthesis and API completion is rather blurry, code-completion tools typically expect a partial program and provide a ranked list of (single-line) completions. Hence, code snippets generated by API completion tools are typically much simpler compared to synthesis tools.

InSynth is a recent API-completion tool that uses theorem proving to compute type inhabitants [19, 20]. While InSynth handles higher-order functions and polymorphism quite elegantly, it cannot synthesize multi-statement code snippets that involve impure functions. As discussed in Section 9, InSynth can only synthesize one example out of the 30 benchmarks used in our evaluation.

Another recent code-completion tool is SLANG [39] which predicts probabilities of API calls using statistical methods. Because SLANG is based on machine learning, it requires training data and is therefore only applicable when the target API has a significant number of clients. However, we believe that the SLANG approach is complementary to ours. In particular, we could use a SLANG-like approach to prioritize some reachable paths in the Petri net over others.

[11] There is a way to enforce this property on the hypergraph representation, but the problem then reduces to solving the Subtree Constrained Hyperpath problem, which is known to be NP-hard [33].

[12] k is the number of B-paths; n and m are the number of vertices and hyperedges in H, respectively; and $size(G)$ is the size of G given by the sum of the cardinalities of its hyperedges.

Our approach is also related to type-directed completion, in which users issue queries using partial expressions [37]. An example of such a partial expression is `?(img, size)`, which queries for API components that are likely to use variables `img` and `size`. While extremely useful in IDEs, this approach can only synthesize single-line code snippets rather than entire methods.

Another tool that is related to automated API completion is MATCHMAKER, which synthesizes "glue code" to allow framework classes to interact with each other [53]. Unlike SYPET where the query is a method signature, MATCHMAKER queries are of the form "How can I get type `A` and type `B` to interact with each other?" Because MATCHMAKER uses dynamic traces, the techniques underlying this tool are very different from SYPET.

Programming by Example. Similar to many *programming-by-example* (PBE) approaches, SYPET requires users to provide partial specifications as input-output examples [2, 9, 14, 17, 21, 35, 44]. While most PBE approaches target end-users who cannot program, SYPET is intended for programmers. In contrast to most PBE approaches that target a specific domain (like string or list manipulations), SYPET can be used for any API, although it can only synthesize straight-line programs. Similar to SYPET, the DBS tool [38] is domain-agnostic and can be viewed as a meta-synthesis tool for generating example-guided synthesizers. While DBS can synthesize more complex programs with loops and conditionals, its scalability depends on a small set of components chosen by a domain expert.

Program Sketching. In sketch-based synthesis [25, 45–48], the programmer writes a draft program containing missing expressions. The pioneer of this approach is the SKETCH system [45], which uses counterexample-guided inductive synthesis (CEGIS) to complete the holes. Unlike SKETCH, which expects the programmer to write the program sketch, SYPET automatically generates sketches. However, the program sketches generated by SYPET are always straight-line programs, where the holes are unknown function arguments. Furthermore, while the holes in SKETCH always correspond to constants, unknown expressions in SYPET are variables.

Graph reachability for synthesis. The main novelty of our approach is the use of Petri nets in the context of type-directed synthesis. Petri nets are a widely-used modeling tool in the context of concurrent and distributed systems, and much existing work focuses on their properties and analysis [32]. To the best of our knowledge, the only previous application of Petri nets in program synthesis is for deadlock avoidance in concurrent C programs [52].

SYPET is closely related to the PROSPECTOR tool for synthesizing "jungloid code snippets" [30]. A *jungloid* is a composition of API calls, where each method takes a single argument and returns a non-void value. Similar to our technique, PROSPECTOR constructs a graph from method signatures and looks for a reachable path between the source and target. As mentioned earlier, our Petri-net formulation can be viewed as a generalization of the jungloid graph.

The DENALI tool for super-optimization performs graph reachability analysis to generate more efficient, but semantically equivalent code [27]. DENALI uses *E-graphs* to represent all possible ways of computing a term and uses a SAT solver to find the most efficient execution strategy. Similar to DENALI, SYPET also uses a SAT-based approach to solve the graph-reachability problem. However, both the application domains as well as the underlying graph representations are different.

Reinking et al. [40] have recently proposed an approach that uses graph reachability for API synthesis and repair. Similar to INSYNTH, this approach cannot synthesize multi-statement code snippets involving impure methods. While we tried to empirically compare SYPET against the implementation of Reinking et al., we were not able to synthesize any of our benchmarks using their tool.

12. Conclusion

We have proposed a new type-directed approach to component-based program synthesis. Our approach constructs a Petri net from the signatures of API components and generates a code sketch by identifying accepting runs of the resulting Petri net. The code sketches are then completed using SAT-based reasoning and tested on the user-provided examples.

We evaluated SYPET on a collection of programming tasks involving six widely-used APIs. Our evaluation shows that SYPET can synthesize the desired program in a practical manner using few test cases. Our tool is publicly available [1] and can be easily used by programmers to synthesize complex APIs from test cases.

Acknowledgments

We thank Thomas Dillig, Navid Yaghmazadeh, Arati Kaushik, Osbert Bastani, Zhao Song and David Melski for their insightful comments. We would also like to thank the anonymous reviewers for their helpful feedback.

This work was supported in part by NSF Award #1453386 and AFRL Awards #8750-14-2-0270. The views, opinions, and findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] SyPet. `http://fredfeng.github.io/sypet/`.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950. Springer-Verlag, 2013.

[3] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV*, pages 340–351. Springer, 1997.

[4] B. Avent, A. Ritz, and T. Murali. halp: Hypergraph Algorithms Package. `http://murali-group.github.io/halp/`.

[5] D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, pages 59–6, 2010.

[6] J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Springer, 2008.

[7] P. Fabiani and Y. Meiller. Planning with tokens: an approach between satisfaction and optimisation. In *PuK*, pages 26–35, 2000.

[8] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-Based Synthesis for Complex APIs. Technical report, University of Texas at Austin, 2016.

[9] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[10] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[11] A. Finkel. *The minimal coverability graph for Petri nets*. Springer, 1993.

[12] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663. ACM, 2014.

[13] G. Gallo, G. Longo, and S. Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

[14] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[16] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61. ACM, 2011.

[17] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[18] T. Gvero and V. Kuncak. Synthesizing Java expressions from free-form queries. In *OOPSLA*, pages 416–432, 2015.

[19] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[20] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[21] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[22] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. *Fundamenta Informaticae*, pages 247–268, 1999.

[23] S. L. Hickmott, J. Rintanen, S. Thiébaux, and L. B. White. Planning via petri net unfolding. In *IJCAI*, pages 1904–1911. AAAI Press, 2007.

[24] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125. ACM, 2005.

[25] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. Jsketch: Sketching for Java. In *ESEC/FSE*, pages 934–937. ACM, 2015.

[26] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. IEEE, 2010.

[27] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314. ACM, 2002.

[28] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, pages 147–195, 1969.

[29] D. E. Knuth. A generalization of dijkstra's algorithm. *Inf. Process. Lett.*, 6(1):1–5, 1977.

[30] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61. ACM, 2005.

[31] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV*, pages 164–177. Springer, 1993.

[32] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[33] L. R. Nielsen, K. A. Andersen, and D. Pretolani. Finding the *K* shortest hyperpaths. *Computers & OR*, 32:1477–1497, 2005.

[34] S. Ogata, T. Tsuchiya, and T. Kikuno. SAT-based verification of safe petri nets. In *ATVA*, pages 79–92. Springer, 2004.

[35] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[36] D. Peled. Ten years of partial order reduction. In *CAV*, pages 17–28. Springer, 1998.

[37] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286. ACM, 2012.

[38] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, page 43. ACM, 2014.

[39] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, page 44. ACM, 2014.

[40] A. Reinking and R. Piskac. A type-directed approach to program repair. In *CAV*, pages 511–517, 2015.

[41] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM, 2006.

[42] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, pages 613–620, 1975.

[43] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, pages 740–751, 2012.

[44] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651. ACM, 2012.

[45] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[46] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.

[47] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[48] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178. ACM, 2007.

[49] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213. ACM, 2007.

[50] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[51] W. Vogler. Efficiency of asynchronous systems and read arcs in petri nets. In *ICALP*, pages 538–548. Springer, 1997.

[52] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, pages 252–263. ACM, 2009.

[53] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82. ACM, 2011.

Appendix A: Proofs of Theorems

Proof of Theorem 1.

Proof. Let M' be a configuration (marking) such that $M'(p) > k + 1$ for some place p, and suppose that $t' = T_1, T_2, \ldots, T_n$ is a sequence of transitions that can be fired starting from M'. Now, let M be another configuration such that $M(p) = k + 1$ and for all other p', $M'(p) = M(p)$. We will show that trace $t = T_1, \kappa^{c_1}, T_2, \ldots, \kappa^{c_n}, T_n$ can be fired from M, where each c_i is the number of p tokens consumed by T_i and κ denotes a generic clone transition. Because we consider these two paths t, t' to be equivalent, this property implies any trace that can be generated from M' can also be generated from M.

We will prove this claim using induction, using the following (strengthened) inductive hypothesis. If $T_1, \ldots, T_i$ is reachable from M' then (i) $T_1, \kappa^{c_1}, \ldots, T_i, \kappa^{c_i}$ is reachable from M and (ii) $M_i(p) = k + 1$ and $M_i(p') = M'_i(p')$ for all $p' \neq p$. (Here, we use M_i to denote the marking right before transition T_{i+1}.)

For the base case, we have $i = 1$. Because T_1 is reachable from M' in one step, we can fire T_1 in M'. Now let $p_1, \ldots, p_m$ be the predecessors of T_1 with edge weights $w_1, \ldots, w_m$. For any $p_i \neq p$, M has the same number of tokens as M'. Furthermore, if $p_i = p$, then $w_i \leq k$. Hence, T_1 is also enabled at M. Furthermore, we have at least 1 token left at p after taking transition T_1, so the clone transition remains enabled after T_1. Because the clone transition does not decrease the number of tokens, it remains enabled, so we can execute it as many times as we want. Hence if T_1 is reachable from M', then T_1, κ^{c_1} is reachable from M.

Now, we'll prove property (ii) for the base case. Suppose transition t consumed c_1 number of p tokens. Right before T_2, we still have $k + 1$ tokens at p because we fired c_1 clone transitions. Furthermore, for all other places p', the number of tokens remains the same because they were the same in M, M' and we took the same transition T_1 in both traces.

For the inductive step, we show the property for $i + 1$. Suppose we take transition T_{i+1} in t'. By the inductive hypothesis, we know:

1. $T_1, \kappa^{c_1}, \ldots, T_i, \kappa^{c_i}$ is a prefix of t and
2. $M'_i(p') = M_i(p')$ for $p' \neq p$ and $M_i(p) = k + 1$

Observe that if T_{i+1} is enabled at M'_i, then it must also be enabled at M_i using (2) and the same reasoning as in the base case. Furthermore, we will have at least one p token left after executing T_{i+1}, so the clone transition is again enabled. Now, we execute as many clones as T_{i+1} consumed p tokens, so $M_{i+1}(p)$ will remain $k + 1$. For all other places p', we still have $M_{i+1}(p') = M'_{i+1}(p')$ because they were initially the same, and T_{i+1} consumed an equal number of tokens. □

Proof of Theorem 2.

Proof. Let p be any path that starts at marking M and ends at M' in $\mathcal{R}(\mathcal{N})$. We will prove that $M'(\tau^*) > 0$ for some place $\tau^* \neq$ `void` that is reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. Because τ^* is reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$, we have $\tau' \neq \tau$. Furthermore, because M^* must assign 0 to τ^*, this property implies that no path starting at M can end in M^*.

The proof is by induction on the length of path p. For the base case, we have *length*$(p) = 0$ (i.e., $M' = M$). Because $M(\tau') > 0$ and because τ' is reachable from itself, the property holds in the base case.

For the inductive step, let us consider a path p of length $k + 1$ that ends in M'', and let p' be the prefix of p of length k. By the inductive hypothesis, p' ends in a marking such that $M'(\tau^*) > 0$ for some place τ^* reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. There are two possibilities: We either fire a transition f that (i) has τ^* as its predecessor or (ii) does not have τ^* as its predecessor. In the latter case, $M''(\tau') > 0$ because we did not consume any tokens of τ^*, so the property holds. For case (i), f consumes at least one token of τ^* but produces at least one token at some other place τ'', so we have $M'(\tau'') > 0$. Because τ'' is reachable from τ^*, it is also reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. Furthermore, τ'' cannot be `void`; otherwise, this would imply that τ is reachable from τ' in $\alpha(\mathcal{N})$ because every type is reachable from `void`. Because we have shown that $M'(\tau'') > 0$ for some $\tau'' \neq$ `void`, the property also holds in the inductive step. □

Appendix B: Symbolic Encoding of Petri Net

Given a Petri net $\mathcal{N}$, there are several ways to encode the reachability problem as a propositional formula [6, 22, 31, 34]. One approach is to view the reachability problem as a planning problem [7, 23]. A *planning problem* using the standard STRIPS propositional formalism [10] is defined as a four-tuple (A, O, I, G) where A is a set of atoms, O is a set of operators or actions, and I and G correspond to the initial and goal states, respectively. An operator $o \in O$ is defined as triple $o = (pre, del, add)$ where pre is the set of preconditions for an action o to be fired, del are the negative effects of firing o and add are the positive effects of firing o. A *state* of the planning problem is an element of 2^A. If o can be fired, then this will lead to a new state $o(s) = (s \setminus del) \cup add$. A solution of length n to a planning problem corresponds to finding a sequence of actions $o_1, \ldots, o_n$ such that I enables o_1, o_i enables o_{i+1} with $1 \leq i < n$ and o_n enables G.

The reachability problem of a Petri net $\mathcal{N} = (P, T, E, W, M_0)$ can be seen as finding a plan of the STRIPS instance (P, T, M_0, M^*). Given a predefined plan length of l, our goal is to build a 0-1 ILP formula ϕ that encodes all possible plans of that length. As Boolean variables, we consider the state of each place $\{{}^{w}s_p^t \mid p \in P\}$ and each transition $\{s_f^t \mid f \in T\}$ for each time step $t \in L$, where L is the set of all time steps between 0 and l. For each place p, we also consider the possible number of tokens w that can be in p at each time step t. Note that by Theorem 1, the number of tokens in each place p is bounded by the maximum weight of the outgoing edges from p. Let $max(p)$ denote the maximum number of tokens at place p.

For $x \in \{P \cup T\}$, let ${}^{\bullet}x = \{y \in P \cup T \mid (y, x) \in E\}$ be the *preset* of x and $x^{\bullet} = \{y \in P \cup T \mid (x, y) \in E\}$ the *postset* of x. A transition $f \in T$ is defined as $f = (pre, del, add)$, where pre corresponds to the preconditions on the markings of $p \in {}^{\bullet}f$ at time step t to fire f, del corresponds to the decrease of the markings of $p \in {}^{\bullet}f$, and add corresponds to the increase of the marking of $p \in f^{\bullet}$ at time step $t + 1$ after firing f. A function $m : P \times L \rightarrow \mathbb{N}$ denotes the marking of a place $p \in P$ at time step $t \in L$. This notion can be extended to consider the marking of p after firing a transition f. A function $m' : P \times T \times L \rightarrow \mathbb{N}$ defined as $m'(p, f, t) = m(p, t) - |\{p\} \cap {}^{\bullet}f| + |\{p\} \cap f^{\bullet}|$ denotes the marking of a place p at time step t after firing a transition f.

For a transition f to be fired at time step t, we need to know which places $p \in {}^{\bullet}f$ have enough resources. This set is given by a function $e : T \times L \rightarrow P$ defined as $e(f, t) = \bigwedge_{p \in {}^{\bullet}f} {}^{m(p)}s_p^t$.

Using the previous definitions, we can now present the constraints that encode the reachability problem as a propositional formula ϕ:

(1) Exactly one transition f is fired at each time step t:

$$\forall_{t=0}^{l-1} \sum_{f \in T} s_f^t = 1$$

(2) If a transition f is fired at time step t then it implies that all places $p \in f^{\bullet}$ will have their markings increased at time step $t+1$:

$$\forall_{t=0}^{l-1} \forall_{f \in T}\ (s_f^t \land e(f,t)) \implies \forall_{p \in f^{\bullet}}\ {}^{m'(p,f,t)}s_p^{t+1}$$

(3) If a transition f is fired at time step t then it implies that all places $p \in {}^{\bullet}f$ will have their markings decreased at time step $t+1$:

$$\forall_{t=0}^{l-1} \forall_{f \in T}\ (s_f^t \land e(f,t)) \implies \forall_{p \in {}^{\bullet}f}\ {}^{m'(p,f,t)}s_p^{t+1}$$

(4) If all outgoing transitions f from a given place p are not fired at time step t then the marking in p does not change between time step t and $t+1$:

$$\forall_{t=0}^{l-1} \forall_{p \in P}\ (\bigwedge_{f \in {}^{\bullet}p} \neg s_f^t) \implies \forall_{w=0}^{max(p)} ({}^{w}s_p^t \implies {}^{w}s_p^{t+1})$$

(5) The initial state is defined by imposing the initial markings in M_0 at time step $t = 0$ as follows:

$$\bigwedge_{p \in M_0} {}^{m(p,0)}s_p^0, \qquad \bigwedge_{p \in P \setminus M_0} \neg({}^{m(p,0)}s_p^0)$$

(6) Similarly, the goal state is defined by imposing the final marking in M^* at time step $t = l$ as follows:

$$\bigwedge_{p \in M^*} {}^{m(p,l)}s_p^l, \qquad \bigwedge_{p \in P \setminus M^*} \neg({}^{m(p,l)}s_p^l)$$

The above constraints can be encoded as pseudo-Boolean constraints and given to a 0-1 ILP solver. A satisfying assignment σ to ϕ will assign one transition f to *true* at each time step. The sequence of transitions between time steps 0 and $l-1$ will give a trace r that can be converted to a program sketch Σ by ignoring the clone transitions.

A Petri net $\mathcal{N}$ may have many feasible paths, and enumerating all paths until the correct program sketch is found may take a large number of iterations. We extend our 0-1 ILP encoding to generate "most-promising" paths first by incorporating an objective function $\sum_f c_f x_f$. Here, x_f is a variable that is assigned to 1 by our encoding iff transition f is used in the accepting run and to 0 otherwise. The relationship between the x_f variables and the state variables s_f^t that describe the transitions is encoded by the following additional constraints:

(7) If x_f occurs then transition f must occur at some time step:

$$\forall_{f \in T}\ x_f \implies \bigvee_{t=0}^{l-1} s_f^t$$

(8) If transition f occurs at some time step then x_f must be assigned to 1:

$$\forall_{t=0}^{l-1}\ s_f^t \implies x_f$$

The costs c_f used in the objective function express the likelihood of a transition f being used in the synthesized code. While there are many heuristics for assigning costs to transitions, our current implementation uses a similarity metric between the name of the desired method and the documentation of each library component. For further details on how these costs are computed, we refer the interested reader to Appendix C.

Appendix C: Similarity Metrics

In this appendix, we describe how to compute a similarity metric between the desired method and each component in the library. These similarity metrics are used to generate a relevance score for each component, which in turn corresponds to a coefficient of the objective function associated with the ILP encoding of the Petri net (recall Section 6.4).

Suppose that the desired method is represented as a pair $Q = (M, D)$, where M denotes the name literal, and D denotes a sequence of words that are extracted from the method name. For instance, for a method called `computeAverage`, we have $M =$`computeAverage`, and $D = \{$ `compute, average` $\}$. Similarly, we also represent each component P_i in the library using a pair (m_i, c_i) where m_i denotes the component name, and c_i denotes its corresponding Javadoc.

Given a method Q and a component P_i, the similarity between Q and P_i, denoted $\delta(Q, P_i)$, is a real-valued number in the range $[0, 1]$. Intuitively, the larger the value of $\delta(Q, P_i)$, the more relevant component P_i is to the desired method Q. SYPET prioritizes components that are most relevant to Q according to our similarity metric δ. Specifically, given a desired method $Q = (M, D)$ and a component $P_i = (m_i, c_i)$ we define $\delta(Q, P_i)$ as follows:

$$\delta(Q, P_i) = w_1 \cdot \delta_m(M, m_i) + w_2 \cdot \delta_d(D, c_i)$$

In other words, $\delta(Q, P_i)$ is a weighted average of two different similarity metrics, taking into account P_i's name and Javadoc, respectively. Each weight w_i is a real number in the range $[0, 1]$ satisfying the constraint $w_1 + w_2 = 1$.

The first similarity metric δ_m for names is straightforward and defined as follows:

$$\delta_m(M, m_i) = 1 - \frac{\textit{edit}(M, m_i)}{\textit{len}(M) + \textit{len}(m_i)}$$

In this definition, the numerator $\textit{edit}(M, m_i)$ is the standard *edit distance* between the names M and m_i, and the denominator is a normalizer to ensure that $\delta(M, m_i)$ is a real number in the range $[0, 1]$.

To compute the second similarity metric δ_d, we represent both D and c_i as text documents and measures their similarity using the vector-space model. More specifically, our approach first represents both the word-sequence part D of the desired method as well as the Javadoc c_i of component P_i as a set of words $\{W_1, \ldots, W_n\}$. When converting D and c_i to its constituent words, our method filters stopwords in the English language and performs standard token-normalization tricks, such as removing plurals and verb tenses.

Now, let Σ_Q and Σ_P represent the set of words extracted from D and c_i respectively, and let $\Sigma = \Sigma_Q \cup \Sigma_P$ be an ordered set, referred to as the *dictionary*. The next step towards defining our distance metric δ_d is to represent both D and c_i as vectors in a $|\Sigma|$-dimensional space. We view D and c_i as text documents and convert them to vectors using *tf-idf weighting* [42]. Specifically, given text T, let $\alpha(T)$ denote a vector $\vec{v}$ such that the i'th element v_i of $\vec{v}$ is assigned as follows:

$$v_i = \textit{count}(\Sigma_i, T) \cdot \log \frac{N}{n_{\Sigma_i}}$$

Here, the term $\textit{count}(\Sigma_i, T)$ represents the number of occurrences of the i'th dictionary word Σ_i in text T. In contrast, N denotes the total number of components in the library, and n_{Σ_i} is the number of components in which word Σ_i appears. Intuitively, v_i represents the frequency of word Σ_i in text T weighted by its importance.

Using this machinery, we now define δ_d as the standard cosine distance between vectors $\alpha(D)$ and $\alpha(c_i)$:

$$\delta_d(D, c_i) = \cos(\alpha(D), \alpha(c_i))$$

Finally, by scaling $\delta(Q, P_i)$ with a factor -100 and rounding to the nearest integer, we convert the similarity to an integer-valued cost in the range $[-100, 0]$.

Lib	ID	Description	SyPet Time(s)	No Pruning	No Obj	Only ILP
apache math	1	Compute the pseudo-inverse of a matrix	5	6.17	4.72	3.47
	2	Compute the inner product between two vectors	0.45	0.7	0.95	0.61
	3	Determine the roots of a polynomial equation	0.12	0.2	0.1	0.23
	4	Compute the singular value decomposition of a matrix	0.2	0.31	0.21	0.28
	5	Invert a square matrix	0.7	5.37	1.42	5.45
	6	Solve a system of linear equations	33.75	42.49	T/O	T/O
	7	Compute the outer product between two vectors	2.48	118.29	6.41	705.97
	8	Predict a value from a sample by linear regression	2.79	0.68	397.88	18.41
	9	Compute the ith eigenvalue of a matrix	155.56	247.23	196.49	T/O
geometry	10	Scale a rectangle by a given ratio	0.74	1.55	0.56	1.34
	11	Shear a rectangle and get its tight rectangular bounds	1.61	2.93	0.54	1.24
	12	Rotate a rectangle about the origin by the specified number of quadrants	0.38	1.22	0.27	1.18
	13	Rotate a two dimensional geometry object by the specified angle about a point	2.29	4.16	3.75	5.77
	14	Perform a translation on a given rectangle	0.77	1.52	0.53	1.28
	15	Compute the intersection of a rectangle and the rectangular bounds of an ellipse	0.09	0.13	0.12	0.12
joda	16	Compute number of days since the specified date	4.11	5.85	225.99	48.07
	17	Compute the number of days between two dates considering timezone	152.13	111.94	T/O	87.31
	18	Determine if a given year is a leap year	19.06	75.02	58.82	T/O
	19	Return the day of a date string	0.88	7.42	1.12	11.29
	20	Find the number of days of a month in a date string	51.61	272.54	367.06	T/O
	21	Find the day of the week of a date string	52.41	267.33	141.13	486.11
	22	Compute age given date of birth	7.14	18.51	121.04	13.09
jsoup, dom, text	23	Compute the offset for a specified line in a document	0.27	0.43	0.4	0.69
	24	Get a paragraph element given its offset in the document	1.42	5.6	1.08	3.91
	25	Obtain the title of a webpage specified by a URL	9.34	15.53	3.91	11.78
	26	Return the doctype of an XML document generated by a string	1.06	1.36	0.7	1.42
	27	Generate an XML element from a string	0.89	1.52	1.48	1.45
	28	Read XML document from a file	0.16	0.14	0.12	0.13
	29	Generate an XML from file and query it using XPath	1.07	5.12	1.12	204.34
	30	Read XML document from a file and get the value of root attribute specified by a string	0.37	0.42	0.3	0.49

Table 1. Summary of extended experimental results

Appendix D: Extended Results

The extended results of our evaluation are summarized in Table 1, where we compare SyPet with its three different variants. In this experiment, we set a timeout of 30 minutes.

The first column labeled "SyPet Times" shows the total running time of the SyPet tool, while the next three columns show the running time of its three variants. In particular, the column labeled "No pruning" shows synthesis time when we do not employ the pruning optimization described in Section 6.3. As we can see from Table 1, synthesis time with pruning is on average two times faster than without pruning. This difference in time is closely related to the size of the ILP encoding of the Petri net because without pruning the ILP encoding can become much larger. For instance, while benchmark 7 can be synthesized in 2.48 seconds by SyPet, it takes 118.29 seconds when we do not use pruning.

The column labeled "No Obj" shows synthesis time when we have a symbolic encoding of the Petri net, but we do not use an objective function to optimize (i.e., all paths of length k are deemed equally desirable). The results in Table 1 demonstrate that using a heuristic cost function to guide the search has a significant positive impact on overall synthesis time: For example, while SyPet can synthesize 93% of the benchmarks in under 60 seconds, the "No Obj" variant of the tool can only synthesize 73% of the benchmarks in under 60 seconds and reports a timeout on 2 out of 30 benchmarks.

The last column, labeled "Only ILP", shows the synthesis time when we do not employ any of the afore mentioned optimizations. That is, we still symbolically encode the Petri net as an ILP problem, but without an objective function to optimize. Furthermore, we also do not use the pruning optimization based on graph reachability. As expected, the "Only ILP" variant performs significantly worse on average and reports a timeout on 4 out of 30 benchmarks. For instance, while benchmark 29 can be synthesized in 1.07 seconds by SyPet, it takes 204.34 seconds when we do not use any of the optimizations.