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 SY PET, and used it to successfully synthesize real-world programming tasks extracted from on-line forums and existing code repositories. We also compare SY PET with two state-of-the-art synthesis tools, namely INSYNTH and CODEHINT, and demonstrate that SY PET 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 [12, 22]. 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 [12, 13, 22, 34], 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 [7, 22, 35].

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 [16, 17, 20, 31], 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. A Petri net is a finite directed labeled graph 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
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. 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 been 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.

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:

\[
\text{Area rotate(Area obj, Point2D pt, double angle)}
\]

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.
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 \( N \) 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 \( N \).

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 \( P_2 \rightarrow T_2 \). We have \( W(e) = 2 \), and \( W(e) = 1 \) for all other edges \( e \in E \) (e.g., \( P_1 \rightarrow T_1 \)). The initial marking \( M_0 \) assigns \( P_1 \) to 1, \( P_2 \) to 0, and \( P_3 \) to 0.

A run (or trace) of a Petri net \( N \) is a sequence of transitions that are fired. For instance, some feasible runs of the Petri net shown in Figure 4 include \( T_1, T_1, T_2, T_2, T_3 \) and \( T_1, T_1, T_2, T_3, T_3 \). However, \( T_1, T_2, T_1, T_2, T_3 \) 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 \( N \) 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 \( M^* = [P_1 \rightarrow 0, P_2 \rightarrow 0, P_3 \rightarrow 1] \). 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 \( N \), denoted \( \mathcal{R}(N) \), is the set of all markings that are reachable from the initial state. Given Petri net \( N \) and target marking \( M^* \), a run of \( N \) is accepting if it ends in \( M^* \).

Another important concept about Petri nets is k-safety: A Petri net \( N \) is said to be k-safe if no place contains more than \( k \) tokens for any marking in \( \mathcal{R}(N) \). 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 \( \mathcal{R}(N) \) is bounded iff \( N \) 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 \( S \), a set of components \( A \), and test cases \( E \). Its output is either \( \perp \), meaning that the specification cannot be synthesized using components \( A \), or a well-typed program that passes all test cases \( E \).

Petri-net construction. The first step of our synthesis algorithm is to construct a Petri net using signatures of components in \( A \). In particular, the procedure CONSTRUCTPETRI in Algorithm 1 constructs a Petri net \( N \) where each transition is a component \( f \in A \) and each place corresponds to a type. If there is an edge in the Petri net from \( \tau \) to \( f \) with weight \( w \), component \( f \) takes \( w \) arguments of type \( \tau \). Similarly, an edge from \( f \) to \( \tau' \) indicates that \( f \)'s return value has type \( \tau' \).

Example 2. Figure 6 shows (a small part of) the Petri net generated by CONSTRUCTPETRI for the example from Section 2. The transition labeled \( \text{getX} \) has one incoming edge of weight 1 from Point2D because it takes a single argument of this type. There is also an edge from \( \text{getX} \) to \( \text{doubleValue} \) because \( \text{getX} \)'s return value is double. As another example, the weight of the edge from double to \( \text{setToRotation} \) is 3 because this method requires three arguments of type double. Note that Figure 6 also contains special clone transitions labeled \( \kappa \). Intuitively, these \( \kappa \) transitions allow us to duplicate tokens. As we will see in Section 3, the clone transitions allow us to reuse program variables in the synthesis context.
The initial and final markings on the Petri net are determined by the signature \( S \) provided by the user. For instance, the tokens on the Petri net \( N \) from Figure 6 indicate the initial marking \( M_0 \) of \( N \). In particular, because the desired \textit{rotate} method takes arguments of type \textit{Point2D}, \textit{double}, and \textit{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 \textit{void}. In contrast, \( M_0[\text{Area}] = 0 \) because \textit{rotate} does not take any arguments of type \textit{Shape}.

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

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

\[ KD, \text{getX}, \text{getY}, \text{new AffineTransform}, \]

\[ KT, \text{setToRotation}, \text{createTransformedArea} \]

Another accepting run \( r' \) can be obtained by replacing the transition \text{createTransformedArea} by \text{invert}. Observe that \( KD, \text{getX}, \text{getY} \) is not an accepting run because the marking obtained after this run assigns 3 tokens to \textit{double}.

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

\[ x = \#1.\text{getX}(); \ y = \#2.\text{getY}(); \]
\[ t = \text{new AffineTransform}(); \]
\[ \#3.\text{setToRotation}(#4, #5, #6); \]
\[ a = \#7.\text{createTransformedArea}(#8); \]
\[ \text{return } #9; \]

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

**Sketch completion.** Similar to other sketching-based techniques (e.g., [48]), our technique uses a SAT solver to find possible completions of the generated program sketch. For this purpose, the \textsc{SketchGen} procedure generates a propositional formula \( \phi \) 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 \( h^\Phi_i \), which encode that hole \#1 is filled with program variable \textit{v}. For example, for hole \#4, our encoding generates the following constraint:

\[ h^\Phi_{\text{angle}} + h^\Phi_x + h^\Phi_y = 1. \]

This formula stipulates that hole \#4 must be filled with exactly one of \textit{angle}, \textit{x}, or \textit{y}, because those are the only program variables of type \textit{double}. In addition, our encoding stipulates that each program variable must be used \textit{at least once}. For instance, for variable

 ![Figure 6. Petri net for motivating example](image-url)
angle, we generate the following constraint:

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

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 \( \sigma \) is a model:

\[
\begin{align*}
h_{pt}^1 &\land h_{pt}^2 &\land h_{\#3}^3 &\land h_{\#4}^4 &\land -h_{\#4}^4 &\land -h_{\#5}^5 &\land -h_{\#5}^5 &\land -h_{\#6}^6 &\land -h_{\#6}^6 &\land -h_{\#7}^7 &\land h_{\#8}^8 &\land -h_{\#8}^8 &\land -h_{\#9}^9 &\land h_{\#9}^9
\end{align*}
\]

Observe that \( \sigma \) 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 \( \sigma' \) to \( \phi \) (if one exists) and generates a different completion of sketch \( \Sigma \). If we have already considered all possible ways to fill the holes in \( \Sigma \), our algorithm backtracks by finding a different accepting run of the Petri net \( N' \) 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 \( N \) 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.). Given library components \( \Lambda \) and a desired method signature \( S \), the algorithm constructs \( N = (P, T, E, W, M_0) \) and a target marking \( M^* \) as follows:

- **Places** \( P \) correspond to types used in \( \Lambda \).
- **Transitions** \( T \) represent methods in \( \Lambda \). In addition, for every type \( \tau \in P \), there is a special transition called \( \kappa_{\tau} \).
- **Arcs** \((\tau, f)\) is in \( E \) and \( W[(\tau, f)] = k \) if component \( f \in \Lambda \) takes \( k \) inputs of type \( \tau \).
- **Arcs** \((f, \tau)\) is in \( E \) and \( W[(f, \tau)] = 1 \) if \( f \)’s return type is \( \tau \) for some component \( f \in \Lambda \).
- **Arcs** \((\tau, \kappa_{\tau})\) and \((\kappa_{\tau}, \tau)\) are both in \( E \). Furthermore, \( W[(\tau, \kappa_{\tau})] = 1 \) and \( W[(\kappa_{\tau}, \tau)] = 2 \).

1. **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 \( S \) has type \((\tau_1 \times \ldots \times \tau_n) \rightarrow \tau \), our goal is to produce a token at place \( \tau \) 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 \( \kappa_{\tau} \) (called clone transitions) that effectively allow us to “duplicate” objects of type \( \tau \). 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.

To illustrate the necessity of clone transitions, consider our motivating example from Section 3. 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 \( \kappa_{Point2D} \), 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 \( S \). 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.

### 6. Sketch Synthesis via Petri-Net Reachability

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

1. **Is \( M^* \in \mathcal{R}(N) \)?** 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. **Is \( M^* \in \mathcal{R}(N) \), to synthesize candidate program sketches, we must identify exactly those runs of \( N \) 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 \( \mathcal{R}(N) \) is unbounded. While there are existing methods for answering question (1) for unsafe Petri nets \([8][24]\), 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.

\[ M_0[\text{void}] = 1 \text{ and } M_0[\tau] = k \text{ if } S \text{ has } k \text{ inputs of type } \tau. \]

\[ \text{If the return type of } S \text{ is } \tau, \text{ then } M^*[\tau] = 1, \text{ and } M^*[\text{void}] \geq 0 \text{ and } M^*[\tau] = 0 \text{ for all other types } \tau. \]

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 \( S \) has type \((\tau_1 \times \ldots \times \tau_n) \rightarrow \tau \), our goal is to produce a token at place \( \tau \) 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 \( \kappa_{\tau} \) (called clone transitions) that effectively allow us to “duplicate” objects of type \( \tau \). 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.

To illustrate the necessity of clone transitions, consider our motivating example from Section 3. 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 \( \kappa_{Point2D} \), 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 \( S \). 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.

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

3. **Our use of clone transitions is somewhat related to the use of read arcs in the Petri-net literature \([40]\). 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 \( void \).**
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 $R^*(N)$ is unbounded, exploring a subset $R^*(N)$ of $R^*(N)$ is sufficient for identifying all accepting runs of $N$ (see Section 6.2). The second key insight is to use an over-approximation $\alpha(N)$ of $N$ to avoid exploring states that are irrelevant for reaching the target configuration $M^*$ (see Section 6.3). Finally, rather than explicitly constructing $R^*(N)$, we encode it symbolically and lazily enumerate the most-promising accepting runs of $N$ by solving an optimization problem (see Section 6.4).

### 6.1 Basic Reachability Algorithm

Our algorithm for constructing the reachability graph $R^*(N)$ 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 $R(N)$. The additional lines 12–15 correspond to our customization, and allow us to construct $R^*(N)$ instead of $R(N).

The procedure $\text{REACHGRAPH}$ shown in Algorithm 2 takes as input a Petri net $N$ with initial marking $M_0$ and the return type $\tau$ of the method we would like to synthesize, and returns a reachability graph $R^*$. The nodes of $R^*$ correspond to markings of $N$, and a (directed) edge $\langle M, T, M' \rangle$ indicates that we can reach marking $M'$ from $M$ by firing transition $T$ of $N$. We denote nodes of $R^*$ using labels of the form $(k_1, \ldots, k_n)$, 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 $(2, 0, 0)$, whereas the marking from Figure 5 is given by $(1, 1, 0)$. The loop in lines 7–19 of Algorithm 2 iteratively constructs $R^*$ starting from initial marking $M_0$. In particular, the worklist $\Phi$ contains all reachable markings that have not yet been processed. Initially, the only reachable marking is $M_0$; hence we initialize $\Phi$ 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 $\Phi$ 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 $T$. In our context, each transition has exactly one outgoing edge because every component has exactly one return type.

### 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 $\kappa_j$ 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 $p_0$ in Figure 6 by taking the clone transition $\kappa_0$ 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 5 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 $N$, we can safely ignore markings that assign more than $k$ tokens to a place $p$, where $k$ denotes the maximum weight of any outgoing edge of $p$. To see why we can ignore such markings, observe that no transition in $N$ 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 $k$ remaining tokens. 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 $\text{paths}_{M_0, \kappa_j}(G)$ 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:

**Theorem 1.** Let $R(N)$ be the reachability graph constructed by the basic algorithm of Section 5 and let $R^*(N)$ be the reachability graph constructed by employing lines 12–15 of Algorithm 2. If $p \in \text{paths}_{M_0, \kappa_j}(R(N))$, then $p \in \text{paths}_{M_0, M^*}(R^*(N))$.

Effectively, this theorem states we do not “lose” any valid code sketches by considering the paths of $R^*(N)$ instead of $R(N)$. Furthermore, because the size of $R^*(N)$ is bounded by $n^{k+1}$ where $n$ is the number of places and $k$ is the maximum edge weight in $N$, Algorithm 2 is guaranteed to terminate. However, because places in $N$ 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.

---

1. In our context, each transition has exactly one outgoing edge because every component has exactly one return type.
2. 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.
3. Proofs of all theorems are given in the extended version of the paper. [4]
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 $\tau'$ in the Petri net that is not backwards reachable from our target type $\tau$. Because there is no path from $\tau'$ to $\tau$ in $\mathcal{N}$, it is unnecessary to consider markings where $\tau'$ 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 $(P, P') \in E'$ if 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:

![Graph Induced by Petri Net](image)

**Theorem 2.** Let $\mathcal{N}$ be a Petri net with no path from $\tau'$ to $\tau$ in $\alpha(\mathcal{N})$. Let $M^*$ be the target marking that assigns one token to target type $\tau$, 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 $\tau'$ that is not backwards-reachable from $\tau$ 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 $R^*(\mathcal{N})$ if $p$ is backwards reachable from $\tau$.

6.4 Symbolic Encoding using ILP

So far, our algorithm explicitly constructs $R^*(\mathcal{N})$ and enumerates all paths of $R^*(\mathcal{N})$. However, because $R^*(\mathcal{N})$ 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 $\tau_i$ is one of the input types and $\tau$ is the desired output type, then any accepting run of $\mathcal{N}$ must contain at least as many transitions as the shortest path between $\tau_i$ and $\tau$ 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 $\phi$. In particular, formula $\phi$ 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 \[19\] \[27\] \[30\], 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 [6].

**Algorithm 3 Lazy symbolic path enumeration**

1: procedure LAYZYPATHGEN($\mathcal{N}$, $\tau_1, \ldots, \tau_n$, $\tau$)
2: Input: Petri net $\mathcal{N}$, input types $\tau_1, \ldots, \tau_n$, output type $\tau$
3: Output: An accepting run $t$ of $\mathcal{N}$ if one exists
4: $\pi_1 := $ ShortestPath($\alpha(\mathcal{N}), \tau_i, \tau$) \> Lower bound
5: $k := $ max(length($\pi_1$), ..., length($\pi_n$))
6: while true do
7: $\phi := $ ENCODE($\mathcal{N}, k$) \> Unfolding of length $k$
8: $\psi := $ true
9: while true do
10: $\sigma := $ MINIMIZE($\sum c_i x_i, \phi \land \psi$)
11: if $\sigma = \bot$ then
12: break
13: if CHECK($\sigma$) then
14: return Trace($\sigma$)
15: $\psi := $ $\psi \land $ BLOCK($\sigma$)
16: $k := k + 1$
17: 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 $\sum c_i x_i$ (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. 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 $\sigma$ of $\phi$ 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 $\sigma$ (see Section 4) and runs the test cases. If $\sigma$ does not correspond to a satisfactory code sketch, we need to “block” this assignment in future iterations by adding a blocking clause $\psi$. In the simplest case, a blocking clause can be obtained as the negation of $\sigma$; however, our algorithm generates a stronger blocking clause by performing a particular form of partial-order reduction \[13\] \[32\] 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 $\Sigma$
(b) Fill the holes in $\Sigma$ with program variables

\[8\] We refer the interested reader to the extended version of the paper \[6\] for a more detailed discussion of our similarity metrics.
Each transition in \( \tau \) corresponds to either an invocation of a method \( \text{foo} \) from an API or a special \( \kappa \) transition. When synthesizing code, we ignore clone transitions and only consider API calls. In particular, if some API method \( \text{foo} \) used in \( \tau \) has \( n \) input parameters, the code sketch for \( \text{foo}'s \) invocation looks like the following:

\[
// \text{if } m \text{ is a virtual method} \\
T_0 \text{ out} = \text{foo}(\#2, \#3, \#4, \ldots, \#n+1)
\]

\[
// \text{if } m \text{ is a static method or constructor} \\
T_0 \text{ out} = \text{foo}(\#1, \#2, \#3, \ldots, \#n)
\]

In general, if trace \( \tau \) 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 \( \text{return } \#m \) (when the program does not return \( \text{void} \)).

Now, given sketch \( \Sigma \), we need to instantiate each hole with a program variable. To achieve this goal, we generate a propositional formula \( \phi \) 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 \( \Sigma \). 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 \( \phi \) 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:

- \( h_{\#1} = 1 \)
- \( h_{\#2} = 1 \)
- \( h_{\#3} = 1 \)
- \( h_{\#4} \land h_{\#5} \land h_{\#6} = 1 \)
- \( h_{\#7} = 1 \)
- \( h_{\#8} = 1 \)
- \( h_{\#9} \land h_{\#a} = 1 \)

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

- \( h_{\#1} \geq 1 \)
- \( h_{\#2} \geq 1 \)
- \( h_{\#3} \geq 1 \)
- \( h_{\#4} \land h_{\#5} \land h_{\#6} \geq 1 \)
- \( h_{\#7} \land h_{\#8} \land h_{\#9} \land h_{\#a} \geq 1 \)

Because each satisfying assignment \( \sigma \) to \( \phi \) corresponds to a well-typed completion of sketch \( \Sigma \), 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 \( \phi \land \neg \sigma \) 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 45 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 \( \text{Foo}? \) extends \( \text{A} \) and sub-classes \( \text{B}, \text{C} \) of \( \text{A} \), we generate three different copies of type \( \text{Foo} \), namely \( \text{FooB}, \text{FooC}, \text{FooC} \), 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 \( m, n \): 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 path 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 \( m \) and \( n \), 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 (java.time), and text and XML-related libraries (javax.xml, scala.xml and java.util). 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.
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). 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. 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 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

![Figure 8. Summary of experimental results](image-url)
For instance, 94% of the benchmarks used in evaluating InSynth [16, 17] (single-line) code snippets.

SYPet involves a single API call. For example, YNTH is a type-directed API-completion tool that can synthesize just one of them.

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 [12, 13, 22, 34]. On the other hand, while API-completion tools [9, 16, 17, 26] can handle thousands of components, they can typically only suggest very small (single-line) code snippets.

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 at one 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.

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 [25]. Specifically, given a source type \( \tau_0 \) and a target type \( \tau_{\text{out}} \), PROSPECTOR constructs a graph in which nodes represent types and an edge labeled \( f \) from \( \tau \) to \( \tau' \) indicates that \( f \) is a unary function of type \( \tau \rightarrow \tau' \). Hence, a path from \( \tau_0 \) to \( \tau_{\text{out}} \) corresponds to a sequence of method calls that can be used to solve the synthesis problem defined by \( (\tau_0, \tau_{\text{out}}) \).

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 \( T \) and \( H \) 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 \( v_0 \leadsto v_n \) is a sequence \( v_0, e_1, v_1 \ldots e_n, v_n \) such that \( e_i \in \text{head}(e_{i+1}) \), \( v_i \in \text{tail}(e_i+1) \) 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 \( s \leadsto v \) in \( P \).

In this definition, if an edge \( e \) is chosen to be in \( E_P \), then \( \text{head}(e) \) and \( \text{tail}(e) \) 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...
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 hyper-
graphs for solving the component-based synthesis problem.

The key idea is to construct a B-hypergraph where nodes repre-
sent 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

```java
class CPoint { CPoint(int x, int y, Color c); }
```

In this section, we describe an alternative solution based on hyper-
graphs for this synthesis problem. Note that this graph contains a
hyperedge, where the tail includes arguments of type Point
and the head

```java
t1:Pt r1:int
v2:Pt r2:int
CPt
v3:int
v5:
Color
S Tv4:int T
a1:Pt
a2:
Color
```

The key idea is to construct a

```latex
G
```

-hypergraph where nodes repre-
sent 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 9. Suppose that we want to implement a function makeC-
Point, which takes an argument of type Point and another
argument of type Color. Figure 17 shows the corresponding hyper-
graph 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 algo-
rithms for finding a B-path in a hypergraph [10,23,29]; 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 com-
pletion phases; thus, the algorithm seems conceptually simpler. De-
spite these apparent advantages of hypergraphs, we now point out
some serious drawbacks of this approach.

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:

```java
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 ap-
proach described in Section 10.1. To understand why, consider the
hypergraph from Figure 12, which shows the hypergraph associ-
ated 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 compo-
ents 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 underly-
ing components to determine which parameters are modified. How-
ever, 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 prob-
lem would now become much harder.

In addition to facing difficulties in the presence of impure com-
ponents, 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.  

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 11. For our hypergraph-based implementation, we use the hulp package [4], which can be used to enumerate k shortest hyperpaths. Specifically, the hulp package implements a polynomial-time algorithm [29] 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))).

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 11 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 11.2 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).

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 [12], deobfuscators [22], geometry constructions [13], and string and data-structure transformations [7, 54]. 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 [53]) 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 [9]. 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 11.2 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 [15, 17, 26, 33, 35, 44, 48]. 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 [16, 17]. 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 12, INSYNTH can only synthesize one example out of the 30 benchmarks used in our evaluation.

Another recent code-completion tool is SLANG [35] 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.
Our approach is also related to type-directed completion, in which users issue queries using partial expressions. An example of such a partial expression is \( \text{img, size} \). 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. 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. 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 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, the programmer writes a draft program containing missing expressions. The pioneer of this approach is the SKETCH system, 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. To the best of our knowledge, the only previous application of Petri nets in program synthesis is for deadlock avoidance in concurrent C programs.

SYPET is closely related to the PROSPECTOR tool for synthesizing “jungloid code snippets”. 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. 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. 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 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


