Liu BB, Dong W, Liu JX *et al.* ProSy: API-based synthesis with probabilistic model. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1234–1257 Nov. 2020. DOI 10.1007/s11390-020-0520-4

ProSy: API-Based Synthesis with Probabilistic Model

Bin-Bin Liu[1], Wei Dong[2,*], *Member*, *CCF*, Jia-Xin Liu[1], Ya-Ting Zhang[1], and Dai-Yan Wang[1]

[1] *College of Computer Science, National University of Defense Technology, Changsha 410072, China*

[2] *Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha 410072, China*

E-mail: {liubinbin09, wdong, liujiaxin18, zhangyating18, wangdaiyan}@nudt.edu.cn

Received April 10, 2020; revised October 22, 2020.

Abstract Program synthesis is an exciting topic that desires to generate programs satisfying user intent automatically. But in most cases, only small programs for simple or domain-specific tasks can be synthesized. The major obstacle of synthesis lies in the huge search space. A common practice in addressing this problem is using a domain-specific language, while many approaches still wish to synthesize programs in general programming languages. With the rapid growth of reusable libraries, component-based synthesis provides a promising way, such as synthesizing Java programs which are only composed of APIs (application programming interfaces). However, the efficiency of searching for proper solutions for complex tasks is still a challenge. Given an unfamiliar programming task, programmers would search for API usage knowledge from various coding resources to reduce the search space. Considering this, we propose a novel approach named ProSy to synthesize API-based programs in Java. The key novelty is to retrieve related knowledge from Javadoc and Stack Overflow and then construct a probabilistic reachability graph. It assigns higher probabilities to APIs that are more likely to be used in implementing the given task. In the synthesis process, the program sketch with a higher probability will be considered first; thus, the number of explored reachable paths would be decreased. Some extension and optimization strategies are further studied in the paper. We implement our approach and conduct several experiments on it. We compare ProSy with SyPet and other state-of-the-art API-based synthesis approaches. The experimental results show that ProSy reduces the synthesis time of SyPet by up to 80%.

Keywords application programming interface (API)-based program, Petri net, probabilistic reachability graph, program synthesis

1 Introduction

Program synthesis, the task of automatically generating a program that satisfies user intent, has been regarded as the holy grail of computer science [1] and one of the most central problems in the theory of programming [2]. Many different techniques have been proposed for program synthesis, such as programming by examples [3], syntax-guided synthesis [4], component-based synthesis [5], program sketching [6], and neural programming [7, 8]. The key problem to be solved in most of these techniques is how to specify the user intent accurately and then investigate efficient search techniques in the program space. Due to the huge search space, many approaches define domain-specific languages (DSL) for specific tasks, such as bit-vector [5], string manipulation [9], and table transformation [10]. Although these techniques are efficient for problems in certain domains, they are not suitable for general-purpose program synthesis such as finding Java or C programs that meet various users' requirements.

With the rapid growth of reusable program libraries and components, component-based synthesis [11–13] is becoming an important research field of program synthesis, which assembles programs from a set of components. Although many of these studies are also limited

Regular Paper

Special Section on Software Systems 2020

This paper was supported by the National Natural Science Foundation of China under Grant No. 61690203, and the National Key Research and Development Program of China under Grant No. 2018YFB0204301.

*Corresponding Author

to DSLs [5, 11], there emerge some approaches to synthesizing programs in general programming languages. For example, SyPet [12] is a state-of-the-art component-based approach. It generates Java programs that are only composed of APIs (application programming interfaces), which makes it more practical. But since the program space constituted by all API sequences is still very large, finding the correct solution is time-consuming, and normally, only small programs can be found in practice.

To synthesize more complex API-based programs for real problems, we can recall what programmers do when faced with the same problem. If he/she has never solved such a problem and does not know which APIs to use, he/she will search for knowledge from different sources. He/she might refer to Javadoc① for APIs related to this problem. He/she can also search for programming posts in online forums, such as Stack Overflow②, to find out which APIs are used to solve similar problems. After acquiring the knowledge from different sources, he/she can restrict the APIs to a minimal scope and then try to write the program with these APIs. Obviously, more knowledge of APIs on specific problems can significantly reduce the solution space. In other words, the vast accumulation of source code and related documentation may bring a breakthrough to software automation such as program synthesis [14].

In this paper, we propose a novel probabilistic-model based approach named ProSy to synthesize Java programs which are only composed of APIs. The key idea is to reduce the search space of program synthesis with the knowledge acquired from various coding resources such as Javadoc and Stack Overflow. In our approach, based on the Petri net model used in SyPet [12], we propose the probabilistic reachability graph (PRG) model which assigns a probability to each API. The probability can measure the likelihood of an API to be used in implementing the given programming task. In the synthesis process, the reachable path (program sketch) with a higher probability will be enumerated first, which can improve the efficiency of program synthesis. After that, some extension and optimization strategies are further studied. The evaluation on a set of real programming tasks shows that ProSy is more efficient than existing work.

The main contributions of the paper are listed as below.

- A probabilistic reachability graph (PRG) is proposed for API-based synthesis. It assigns higher probabilities to APIs which are more likely to be used in implementing the given programming task. Considering the API sequences with a higher probability first will speed up the search process of program synthesis.
- The method of calculating API probabilities in PRG based on the data acquired from Javadoc and Stack Overflow is proposed. It considers the similarity between the given programming task and the API usage description as well as the occurrence frequency of general-purpose APIs.
- An extension of handling field accesses and two optimization strategies of pruning model and handling inheritance relationships are studied. They can extend the task types that the synthesizer can handle and can further reduce the search space.
- A set of experiments are conducted to evaluate our approach. The experimental results show that our approach can synthesize more tasks in less time, which demonstrates ProSy outperforms existing work.

The rest of this paper is organized as follows. Section 2 introduces some preliminary concepts. Section 3 presents the overview of our approach through an example. In Section 4, we illustrate the construction of the probabilistic model and the extension of handling field accesses. Section 5 describes the optimization strategies. Section 6 shows the effectiveness of our approach with experiments. Finally, Section 7 discusses related work, and Section 8 concludes this paper.

2 Preliminaries

SyPet takes a component library of APIs and a signature of the desired method together with test cases as input. It decomposes the synthesis process into two separate phases of sketch-generation and sketch-completion. The first phase performs a reachability analysis on the Petri net to generate a reachable path, which corresponds to a program sketch represented as a sequence of APIs. The second phase completes the sketch with parameters and variables to generate a candidate program, which is then verified by test cases.

As a start, we give a brief introduction about the Petri net which is used to model API relations in SyPet [12]. It is also used as the basis of our approach in this paper. The Petri net is a mathematical modeling language first invented by Carl Adam Petri [15]. Fig.1

① https://docs.oracle.com/javase/8/docs/api/index.html, Oct. 2020.
② https://stackoverflow.com, Oct. 2020.

gives an example of a Petri net. A Petri net contains places (represented by circles), transitions (represented by bars), and arcs (represented by a directed edge between a place and a transition). Each place contains a discrete number of tokens (represented by black dots). The weight on an arc from a place to a transition indicates how many tokens will be consumed in the input place to fire this transition, and the weight on an arc from a transition to a place indicates how many tokens will be created in the output place by the transition. A transition is enabled if all of its input places contain sufficient tokens to be consumed. Only one transition can be fired at one time. The mapping from each place to the number of tokens is called a marking, which indicates a configuration of the Petri net.

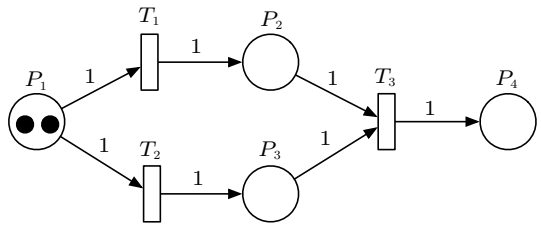

Fig.1. Example of Petri net.

Definition 1 (Petri Net [12]). *A Petri net is a* 5*-tuple* $N = (P, T, E, W, M_0)$*, where* P *and* T *are disjoint sets of places and transitions respectively.* $E \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs.* $W : E \to \mathbb{N}$ *assigns a non-negative integer (weight) to each arc.* M_0 *is the initial marking of* N*.*

Reachability graph of a Petri net denoted as $R(N)$ is constructed by connecting its reachable markings with the firing transitions. The reachability graph of the Petri net in Fig.1 is shown in Fig.2. For instance, transition T_1 from marking $[2, 0, 0, 0]$ to $[1, 1, 0, 0]$ represents how the marking will change when T_1 is fired in configuration $[2, 0, 0, 0]$. The definition of the reachability graph is given as follows.

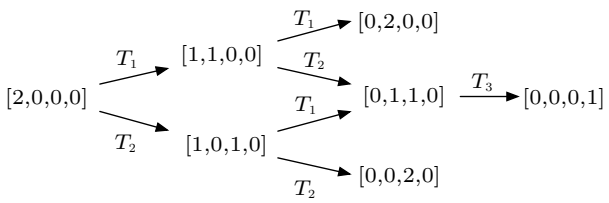

Fig.2. Reachability graph of the Petri net in Fig.1.

Definition 2 (Reachability Graph). *Given a Petri net* $N = (P, T, E, W, M_0)$*, a reachability graph* $R(N)$ *is a directed graph* (V, E')*, where* V *is the set of reachable markings of the Petri net from the initial marking* M_0*. Each directed edge* e *in* E' *is a* 3*-tuple* (M, t, M')*, which means we can reach marking* M' *from* M *by firing transition* t*.*

The reachability problem of the Petri net is to decide, given a Petri net N and a marking M, whether $M \in R(N)$. An important property of the Petri net is boundedness. A place in the Petri net is called k-bounded if it does not contain more than k tokens in all the reachable markings. A Petri net is called k-bounded if all of its places are k-bounded. A Petri net is bounded if and only if its reachability graph is finite.

A Petri net can be used to model the relations between APIs and data types. In SyPet, a place in the Petri net corresponds to a data type, and a transition corresponds to an API. An arc from place P_1 to transition T_1 means the API T_1 needs the arguments of type P_1. An arc from transition T_1 to place P_2 means the return type of API T_1 is P_2. We also create a place for a special type `void`. There is a special transition named "clone transition". After we have created places for data types, we create a clone transition for each place. Each place is connected to the clone transition with an arc weighted 1 and a reversed arc weighted 2. Clone transitions allow us to duplicate tokens, which ensures the program variables can be reused in the synthesis context.

Example 1. Fig.3 shows a part of the Petri net constructed for library `org.joda.time`. Places `DateTime`, `LocalDate`, and `Days` represent class types. Place `int` represents the primitive data type `int`. Transitions `toLocalDate`, `daysBetween`, and `getDays` represent APIs. Each place is connected with a clone transition. `toLocalDate` has an incoming arc weighted 1, which means it needs an argument of type `DateTime`. The outgoing arc of `toLocalDate` means its return type is `LocalDate`. The marking of the current configuration

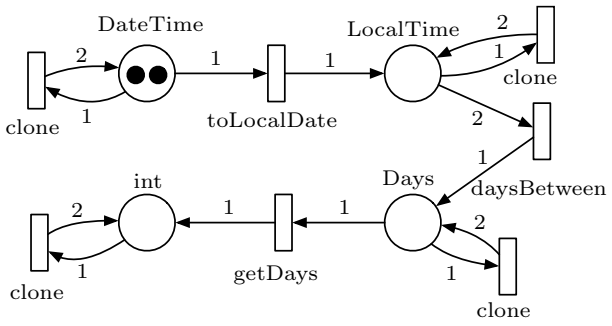

Fig.3. Example of Petri net for modeling API relations.

is $M = [\texttt{DateTime} \rightarrow 2, \texttt{LocalDate} \rightarrow 0, \texttt{Days} \rightarrow 0, \texttt{int} \rightarrow 0]$ (written as $[2, 0, 0, 0]$ for short).

Suppose that a user desires to implement a method `daysBetween` with two input arguments (`arg0`, `arg1`) of type `DateTime` and a return type `int`. In example 1, the corresponding initial marking is $M_0 = [2, 0, 0, 0]$ and the target marking is $M^* = [0, 0, 0, 1]$. Especially, the initial marking assigns one token to the place of type `void`, which is omitted in the figure. A firing sequence of transitions in the reachability graph is shown in Fig.4.

```
toLocalDate; toLocalDate; daysBetween; getDays;
```

Fig.4. Firing sequence of transitions.

The firing sequence of transitions corresponds to the program sketch shown in Fig.5.

```
LocalDate v1 = #1.toLocalDate();
LocalDate v2 = #2.toLocalDate();
Days v3 = Days.daysBetween(#3, #4);
int v4 = #5.getDays();
return #6;
```

Fig.5. Corresponding program sketch.

According to the algorithm in SyPet, after completing the sketch with a set of satisfiable assignments (by assigning `arg0` to `#1`, `arg1` to `#2`, and `v1`, `v2`, `v3`, `v4` to `#3`, `#4`, `#5`, `#6` respectively), a valid candidate program is obtained as shown in Fig.6.

```
int daysBetween(DateTime arg0, DateTime arg1) {
    LocalDate v1 = arg0.toLocalDate();
    LocalDate v2 = arg1.toLocalDate();
    Days v3 = Days.daysBetween(v1, v2);
    int v4 = v3.getDays();
    return v4;
}
```

Fig.6. Obtained candidate program.

From the example above we can see, a Petri net can be used to model the relations between APIs. The problem of program synthesis can be decomposed into two phases of finding a reachable path in the reachability graph and completing the program sketch.

3 Motivation and Approach Overview

3.1 Motivation

Consider a scenario below. A programmer is asked to implement a method named `daysBetween` which intents to compute the number of days between two dates. He/she seeks help from program synthesizers such as SyPet to generate the desired method. He/she needs to provide a method signature, which contains two input arguments of type `DateTime` and a return type `int`. The name of Java library `org.joda.time` should also be designated to tell SyPet the scope of the APIs to be used. Besides, one or more test cases should be provided to verify the correctness of candidate programs. After about three minutes on our platform, the Java program in Section 2 (see Fig.6) will be returned.

Now let us see the search space for this task. The library `org.joda.time` contains about 200 classes and 3 000 methods. The Petri net constructed contains as many as 250 places and 3 500 transitions including transitions of these methods and clone transitions of different places. A more serious problem is that the number of markings in the reachability graph grows exponentially with the number of places and transitions. Even though the desired program only contains four APIs, finding a reachable path from such a large search space is not an easy job. Most of the synthesis time is wasted on the transitions that are not related to this task.

If a programmer is asked to implement this task manually and he/she is not clear which APIs should be used, what would he/she do? Normally, he/she would look for the support from various coding resources based on the description of this task. He/she might look up and read the explanations of related classes and APIs in Javadoc documentation. He/she might turn to programming forums like Stack Overflow to search related posts. He/she might also search code repositories like GitHub to find related implementations via code annotations. Under normal conditions, an API described similarly to a programming task might be more helpful. Similarly, APIs that are rarely used in practice are less likely to be used in the programming tasks.

By these means, he/she can restrict the APIs that might be used to a small scope. The description of the desired method is "compute the number of days between two dates". He/she finds that class `Days` has a method `daysBetween`, which is described similarly to the desired method. It computes the number of days between two objects of `LocalDate`. However, the input of this method has two arguments of type `DateTime`. He/she finds that class `LocalDate` has the method `toLocalDate`, which converts the `DateTime` object to the `LocalDate` object. But the output type of method `daysBetween` is `Days`, while the desired output type is `int`. Further, he/she finds that method `getDays` in class `Days` can convert the `Days` object into `int`. With

the knowledge, he/she can implement a similar program manually.

As we can see, the knowledge from various coding resources can help us reduce the search space. The knowledge may tell us which APIs are more related to certain functionalities. A more relevant API should have a higher probability to be used in the search space. Thus, given a task, looking for the knowledge and constructing a probabilistic model for the search space may significantly improve the efficiency of synthesis.

3.2 Approach Overview

Based on the discussion above, we propose the probabilistic reachability graph (PRG). PRG is constructed by assigning a real number to each edge in the reachability graph. The real number represents the probability of firing the corresponding transition at a specific marking.

Definition 3 (Probabilistic Reachability Graph, PRG). *A PRG is a 4-tuple $PR = (M, E, F, Pr)$, where M is the set of reachable markings of the Petri net, and $E \subseteq M \times M$ is the set of directed edges. $F : E \to T$ is a mapping labeling each edge with a transition (i.e., method) of the Petri net. $Pr : E \to [0, 1]$ assigns a real number to each edge satisfying*:

$$\sum_{(m,m')\in E} Pr(m, m') = 1, \ for\ each\ m \in M.$$

Now we give an overview of our approach as shown in Fig.7. Given a task, the user first provides a method signature and a set of APIs, with which a Petri net is constructed. At the same time, we build a probability calculator with the natural language description of the task and the data extracted from Javadoc and Stack Overflow. The probability calculator assigns each edge in the reachability graph a probability related to this task. Thus, a PRG is obtained. The probability can measure the likelihood of an API to be used in implementing the programming task. When searching for reachable paths in PRG, we propose several extension and optimization strategies to further extend the capability and improve the performance of our approach.

With the initial and target markings of the Petri net determined by the given input and output types respectively, we perform a probability-based search in PRG, during which the reachable path with a higher probability will be enumerated first. The firing sequence in one reachable path corresponds to a program sketch. Afterwards, the sketch is completed with type-check parameters and variables to generate a candidate program. If the candidate program passes all the test cases, the desired program is synthesized; otherwise it rolls back to generate a new candidate program by completing the sketch with other parameters and variables. If there are no candidate programs for this sketch that can pass all the test cases, a new program sketch will be generated by enumerating another reachable path with a higher probability.

Next, we mainly focus on the construction of the PRG model and the extension and optimization strategies, which are the main contributions of this paper. The techniques mentioned in other steps will be illustrated if necessary and the details of them can be found in the paper of SyPet [12].

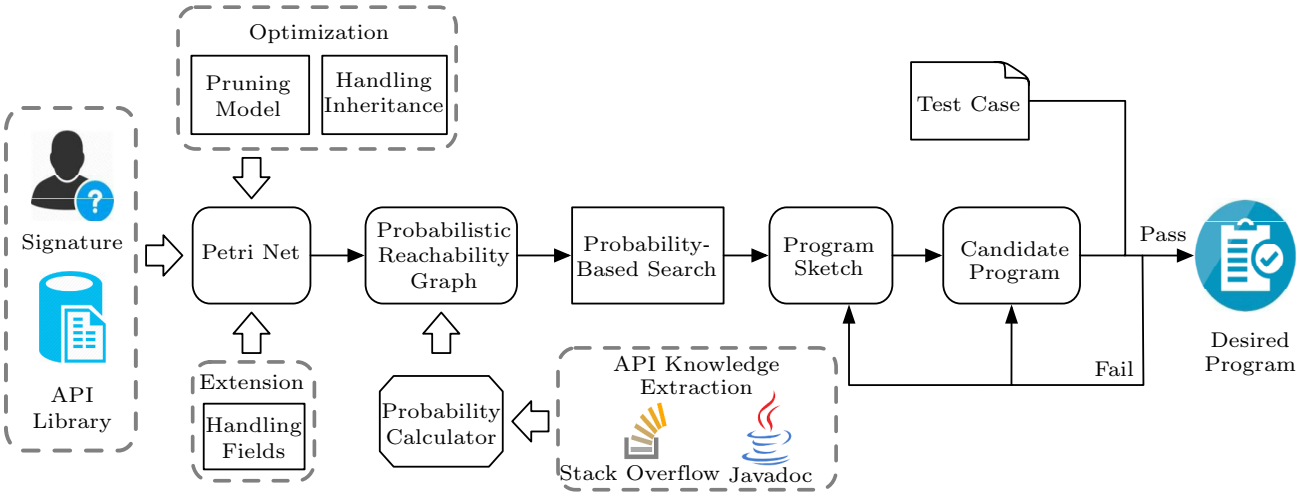

Fig.7. Overview of ProSy.

4 Probabilistic Model Construction

Given a natural language description and the signature of the desired task, ProSy constructs the PRG with the data extracted from Javadoc and Stack Overflow. This process can be divided into two phases. The first phase is constructing a probability calculator from various data resources, and the second phase is constructing the PRG model.

4.1 Data Preparation

To construct the probability calculator, the knowledge of API usage for the given task is needed. This knowledge will be learned from Javadoc documentation and Stack Overflow posts.

4.1.1 Javadoc Data

Javadoc is API documentation in the format of HTML, which contains a detailed description of each API. We first crawl HTML files of Javadoc from the Internet and then use the package BeautifulSoup[③] in Python to extract the signature and natural language description of each API. For the description, we segment it, remove stopwords, and stem each word in it. Finally, we obtain Javadoc data pairs in the form of (c_i, D_i), where c_i indicates the signature of an API and D_i indicates the tokenized word set of the description.

4.1.2 Stack Overflow Data

Stack Overflow is an online Q&A forum for programmers. We download the posts file from Stack Exchange Data Dump[④]. Each question post in the dataset contains one or more tags indicating the topic of the question. We filter the posts with the tag of $\langle java \rangle$ and get a total number of 1 300 000 Java posts. Each question in the post may contain one or more answers, many of which are irrelevant or redundant. We only consider the answers with a score greater than zero. We also discard the answers that do not contain any code snippets. We then match each answer with the corresponding question and obtain a dataset with more than 280 000 Q&A pairs. We then extract the code snippets enclosed in tag $\langle code \rangle$ from the selected answers. After that, we use regular expressions to find all the APIs appeared in these code snippets in the form of `M.n()` or `new M()`. For a question, we only keep its title and get the tokenized word set of the title as we do to the description in Javadoc data.

For each $(Question, Answer)$ pair, we obtain the data pair in the form of (S, T), where S represents the set of APIs appeared in the answer and T represents the set of tokenized words in the title of the question. We traverse all the data pairs (S, T) twice. For the first round, we traverse all the data in S to recognize the signature c_i of each API that appears in the dataset, and then create a word set T_i for each API c_i. For the second round of traversing, we put all the words that appear in the same $(Question, Answer)$ pair with c_i into the word set T_i. Finally, we get Stack Overflow data pairs in the form of (c_i, T_i).

We use an example to show how we prepare the Stack Overflow data in Fig.8. There are two data pairs which are shown in the upper part of the figure. In the first round, we can recognize four APIs from these data pairs, i.e., DocumentBuilder.parse, InputSource.<init>, InputStreamReader.<init>, and ByteArrayInputStream.<init>. We then create four word sets for those four APIs. In the second round, we recognize those words that appear in the same Q&A pair with these APIs and put them into the word sets of them. Finally, we get four data pairs as shown at the bottom of the figure.

In addition, we count the number of occurrences of each API from all the data pairs in the dataset and obtain a set of APIs together with their occurrences in the form of (c_i, n_i). Here, n_i represents the number of occurrences of API c_i. For the example in Fig.8, DocumentBuilder.parse has an occurrence number of 2, and the other three APIs have an occurrence number of 1. We do this to recognize those general-purpose APIs that are commonly used.

4.2 Probability Calculator

After collecting the data from Javadoc and Stack Overflow, we construct a probability calculator with them. The probability calculator first calculates the score of each API for the specified task. The score calculation is based on the hypotheses below.

• The more similar the description of an API is to the programming task, the more possible it is to be used in the task.

• The more frequently an API appears in the dataset, the more possible it is to be used in the programming task.

The second hypothesis is to consider the APIs with

③https://www.crummy.com/software/BeautifulSoup, Oct. 2020.

④https://archive.org/details/stackexchange, Oct. 2020.

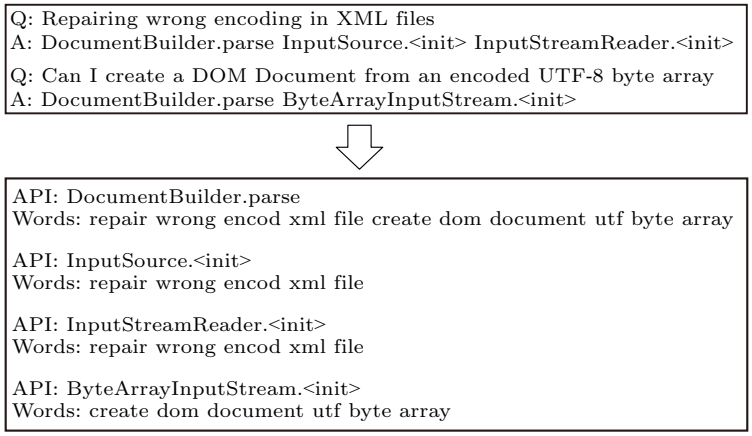

Fig.8. Example of preparing Stack Overflow data.

general purpose. The score calculation can be divided into two parts. The first part is calculated by measuring the text similarity of the description between the API and the desired method. The other part is calculated by measuring the frequency of the API's occurrences in the Stack Overflow dataset.

4.2.1 Similarity Function

The similarity function is used to measure the text similarity between two descriptions. There are many approaches to learning distributed representations of natural language words such as Word2Vec [16] and GloVe [17]. They can compute continuous vector representations of words from a corpus, such that similar words are close together in the vector space. In this paper, we use Word2Vec as the model to learn the embedding for each word.

Given two sets of words W_1 and W_2, the similarity between them is defined as the mean value of two similarities from W_1 to W_2 and W_2 to W_1:

$$sim(W_1, W_2) = \frac{1}{2}(sim(W_1 \to W_2) + sim(W_2 \to W_1)).$$

The similarity from W_1 to W_2 is defined as follows:

$$sim(W_1 \to W_2) = \frac{\sum_{\omega \in W_1} sim_{\max}(\omega, W_2) \times idf(\omega)}{\sum_{\omega \in W_1} idf(\omega)}.$$

Here, $sim_{\max}(\omega, W_2)$ is the maximum value of the cosine similarities between the embedding of ω and each word in W_2. In our work, each set of tokenized words is considered as a document and idf represents the inverse document frequency. The similarity from W_2 to W_1 is defined analogously.

Suppose that the desired method is represented as m, and D represents the set of tokenized words in its description. Given a data pair (c_i, D_i) in Javadoc, the similarity function between the desired method m and API c_i is $sim(D, D_i)$. For the data in Stack Overflow, not all the APIs in the library appear in the dataset. For the APIs in the dataset, given the data pair (c_i, T_i), the similarity function between the desired method m and API c_i is $sim(D, T_i)$.

The similarity function of API c_i represented as $sim(m, c_i)$ is a real number in the range of $[0, 1]$, which is defined as the weighted average of two similarity functions above:

$$sim(m, c_i) = \omega_1 sim(D, D_i) + \omega_2 sim(D, T_i).$$

We have $\omega_1 + \omega_2 = 1$. If c_i never appears in the Stack Overflow dataset, we use the similarity function of Javadoc as the final function. We discuss the derivation of an optimal ω_1 in Subsection 6.5.

4.2.2 Frequency Function

Based on the second hypothesis, if an API appears often, it is more likely to be used in implementing the task. Therefore we increase the similarity for those APIs based on the number of their occurrences. The frequency function is a real number in range $[0, 1]$. Given the occurrences data (c_i, n_i) for each API in Stack Overflow, suppose that the maximum occurrence number of APIs is $N_{\max}$ and the minimum occurrence number of APIs is $N_{\min}$, then the frequency function $freq(c_i)$ is calculated as follows:

$$freq(c_i) = \frac{n_i - N_{\min}}{N_{\max} - N_{\min}}.$$

Given the similarity function $sim(m, c_i)$ and the frequency function $freq(c_i)$, we define the score of API c_i

as follows:

$$score(c_i) = \min\{1, sim(m, c_i) + freq(c_i)\}.$$

By this means, a probability calculator is constructed, and a score is assigned to each API.

4.3 PRG Construction

Before constructing the PRG, we first construct the reachability graph of the Petri net. Due to the existence of clone transitions, each place may have an unbound number of tokens by firing the clone transitions infinite times. Therefore, the reachability graph is infinite, and enumerating reachable paths in it will not terminate. A feasible solution is to construct a finite reachability graph $R^*(N)$ instead of the infinite one. The construction of the finite reachability graph $R^*(N)$ is the same as the construction of the reachability graph expect for an additional constraint on the reachable markings.

For a place p, if the maximum weight of all its outgoing edges is k, we can ignore those markings that assign more than $k+1$ tokens to p. As long as we have $k+1$ tokens in p, no transitions can be disabled due to p. Moreover, no matter what transition is fired, we have at least one token left in p. Because every place has a corresponding clone transition, we can always create k tokens in place p by firing the clone transitions sufficient times. That is the reason why we can ignore those markings. In this way, we can bound the reachability graph without losing completeness. The construction of the reachability graph follows the algorithm in SyPet [12], which is shown in Algorithm 1.

Algorithm 1. Algorithm of Constructing the Finite Reachability Graph

```
   Data: Petri net N = (P, T, E, W, M_0)
   Result: Finite reachability graph R*(N)
1  R*(N) := ({M_0}, ∅, M_0);
2  Φ := {M_0};
3  while Φ ≠ ∅ do
4      Choose M ∈ Φ;
5      Φ := Φ − {M};
6      forall the T ∈ enabled(M) do
7          (M', p) := fire(M, T);
8          if ∀e ∈ out(p).M'[p] > W[e] + 1 then
9              Continue
10         end
11         if M' ∉ Nodes(R*) then
12             Nodes(R*).insert(M');
13             Φ = Φ ∪ {M'};
14         end
15         Edges(R*).insert((M, T, M'))
16     end
17 end
18 return R*
```

The input of the algorithm is a Petri net N. The nodes of the finite reachability graph $R^*(N)$ are markings of the Petri net. Each edge in $R^*(N)$ is a 3-tuple (M, T, M'), which means we can reach marking M' from M by firing transition T. We use a worklist Φ to represent the markings that have not been processed, which is initialized by M_0. In each iteration, we choose an item M from Φ. For each enabled transition T at marking M, we get the marking M' and the output place p by firing transition T. Lines 8 and 9 make sure that the constructed reachability graph is finite. We traverse all the outgoing edges of place p and get the maximum weight k of these edges. We then ignore those markings that assign more than $k+1$ tokens to p. We then add marking M' to the nodes of $R^*(N)$ and add it into the worklist. At last the edge (M, T, M') is added to the edges of $R^*(N)$.

The PRG is constructed on the basis of the reachability graph by assigning a probability to every edge in it. There are mainly two types of transitions in the Petri net, i.e., transitions for APIs and clone transitions for places. After calculating the score for each API, we calculate the score for each clone transition. Each clone transition corresponds to a place. For each place, we record the score of all the outgoing API transitions. The score of the clone transition is calculated as the average value of the scores of all the other outgoing transitions of the place.

The probability of each edge in the probabilistic reachability graph is calculated based on the score of the transitions.

$$Pr(e) = \frac{1}{z} score(c_i), \text{ where } F(e) = c_i,$$

where e represents the edge of the reachability graph, c_i represents the transition (API) labeled on edge e, and z is the normalized function such that they form a probability distribution. Assuming the premarking of e is M, the normalized function z is calculated by summing all the scores of the enabled transitions at marking M. Considering the reachability graph in Fig.2 and assuming the calculated score of three APIs for transitions T_1, T_2, and T_3 are 0.5, 0.3, and 0.6, respectively, the PRG constructed is shown in Fig.9.

The more similar an API is to the desired method, and the more frequently an API appears in the dataset, the higher the probability is. After assigning a probability to each edge, we get a PRG. The probability of each edge in the PRG represents the likelihood of an API to be used in implementing the programming task.

When enumerating reachable paths in the PRG, we prefer the transitions with a higher probability to form a "most promising" path.

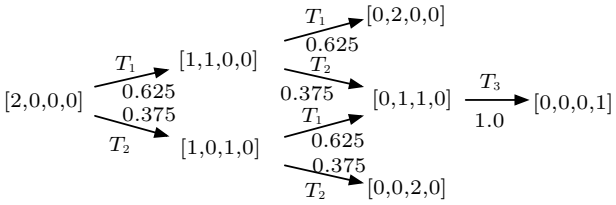

Fig.9. Example of probabilistic reachability graph.

The basic procedure of enumerating reachable paths in PRG is like a search problem in graph. The reachable paths are enumerated in increasing order of length. Supposing we are enumerating a reachable path of length k, we start with the initial marking. In each step we choose to fire the transition with the highest probability among all the transitions that are enabled. If the firing sequence of transitions of length k does not reach the target marking, we backtrack to find another firing sequence with a higher probability.

4.4 Enumerating Reachable Paths with Symbolic Encoding

There are far too many reachable paths in the PRG and enumerating reachable paths by naive searching is very time-consuming. To make it tractable, we encode the problem of enumerating a "most promising" reachable path to a pseudo-boolean problem and use an off-the-shelf SAT solver to solve it. The pseudo-boolean problem is the task of finding a satisfying assignment to a set of PB-constraints that minimize a given objective function [18].

We first introduce another graph induced from the Petri net, which is called induced graph, represented as $\alpha(N)$. The definition of the induced graph is given below.

Definition 4 (Induced Graph). *Given a Petri net $N = (P, T, E, W, M_0)$, an induced graph, denoted as $\alpha(N)$, is a directed graph (V, E'), where $V = P$ and $(p_1, p_2) \in E'$ iff there is a transition $t \in T$ such that $(p_1, t) \in E$ and $(t, p_2) \in E$.*

The nodes of $\alpha(N)$ are places in the Petri net. There is an edge from place p_1 to place p_2 in $\alpha(N)$, if it is possible to reach p_2 from p_1 in the Petri net by firing a single transition. An example of the induced graph of the Petri net in Fig.1 is shown in Fig.10. We can see the distance between two places (the number of transitions fired) clearly from the induced graph.

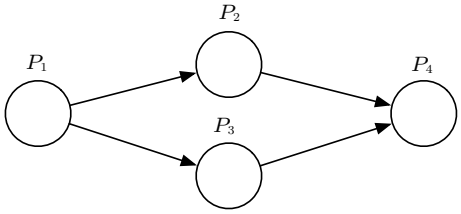

Fig.10. Induced graph of the Petri net in Fig.1.

The algorithm of enumerating the "most promising" reachable path is shown in Algorithm 2. We enumerate reachable paths in increasing order of length. For each input type t_i and target type t, we compute the shortest distance from the place representing t_i to the place representing t in the induced graph, represented as k_i. To ensure each input type can be converted to the target type eventually, the reachable path must contain at least k_i transitions. Therefore we set the shortest length of reachable paths k as the maximum value of each k_i, that is, $k = \max\{k_1, k_2, \ldots, k_n\}$.

Algorithm 2. Enumerating Reachable Paths with Symbolic Encoding

```
   Data: Petri net N, input types t_1, t_2, ..., t_n, output type t
   Result: "most promising" reachable path
1  k_i := ShortestLength(α(N), t_i, t);
2  k := max{k_1, k_2, ..., k_n};
3  while true do
4      Φ := Encode(N, k);
5      Ψ := true;
6      while true do
7          δ := Solve(Φ ∧ Ψ, Σ_i c_i l_i);
8          if δ = ⊥ then
9              break;
10         else
11             return Decode(δ);
12         end
13         Ψ := Ψ ∧ Block(δ);
14     end
15     k := k + 1
16 end
```

When enumerating a reachable path of length k, the conditions that need to be met are encoded as pseudo-boolean constraint Φ[5]. The constraint Φ is satisfiable if and only if there exists a reachable path in the reachability graph of length k. Another formula Ψ is used to make sure that each path can be enumerated only once as shown in line 12. To make sure that the

[5]For the symbolic encoding, we follow the encoding of SyPet. The details of the encoding are beyond the scope of this paper. Please refer to [12] if interested.

reachable path enumerated is a most promising one, we encode the score of each transition as a heuristic objective function. The objective function is in the form of $\Sigma_i c_i l_i = c_0 l_0 + c_1 l_1 + \ldots + c_{k-1} l_{k-1}$, where l_i is a literal and c_i is a coefficient.

To obtain the most promising reachable path, each transition is encoded as a literal, and the score of the transition is encoded as a coefficient. The literal is assigned to 1 if the corresponding transition is fired in the reachable path and to 0 otherwise. The coefficient of each transition can reflect the possibility of each transition being fired. We use the negative value of the score as the coefficient. Therefore the higher the score of the transition, the smaller the coefficient.

We encode the problem of enumerating the most promising reachable path into solving a PB-problem with optimizing functions. In solving the PB-problem, the formula Φ has to be satisfied, and the value of the objective function is preferable to be "the smaller, the better". If we find a satisfiable solution to the problem, we decode the solution into a reachable path in PRG. After each reachable path is enumerated, we add a blocking clause to Ψ to prevent the same path to be enumerated again. After enumerating a reachable path in PRG, a firing sequence of transitions is obtained, which corresponds to a program sketch. It is then completed and verified with the test cases as mentioned in Section 3 and [12].

4.5 Handling Field Accesses

Next, we illustrate an extension of handling field accesses. There is an example below.

Example 2. The user desires a method "getDayOfYear" with an input type `Date` and an output type `int`. The user intent is to get the number of days since the beginning of the year.

One implementation manually written by the programmer is shown in Fig.11.

```
int getDayOfYear(Date date) {                          1
    Calendar calendar = Calendar.getInstance();        2
    calendar.setTime(date);                            3
    int dayOfYear = calendar.get(Calendar.             4
        DAY_OF_YEAR);
    return dayOfYear;                                  5
}                                                      6
```

Fig.11. Implementation written manually.

Consider the statement in line 4 in Fig.11. It accesses the `DAY_OF_YEAR` field of class `Calendar`. We can see that the variable of type `int` is neither passed from the arguments nor defined in the statements before it. From the perspective of the Petri net, this place of `int` is neither generated from places of source types nor generated from the `void` place. This kind of programs is often seen in practice, but the approach like SyPet cannot solve it.

However, we can force to synthesize this program by passing this field as an argument from the method's source types as shown in Fig.12, where `int dayOfYear = Calendar.DAY_OF_YEAR`. But the implementation is not so natural as what most programmers would do.

```
int getDayOfYear(Date date, int dayOfYear) {
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(date);
    int dayOfYear = calendar.get(dayOfYear);
    return dayOfYear;
}
```

Fig.12. Another implementation example.

As shown in Fig.13, to solve this problem, we add a new transition: `new_field`, which can generate the fields in class from place `void`. It takes one token from place `void` (which is assigned one token by the initial marking), and creates one token in the place representing the type of the field. When constructing the Petri net, the fields in each class are analyzed, and then a transition for each field is created, which starts from place `void` and ends to the place indicating the type of the field (from `void` to `int` in Fig.13). Thus, we can extend the capability of our approach by synthesizing such kind of programs.

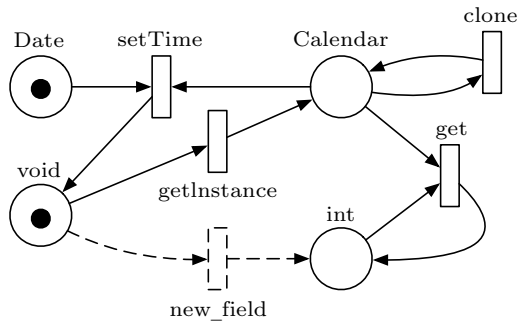

Fig.13. Petri net for the extension of handling field accesses.

But on the other hand, adding this kind of transitions can bring much overhead. Taking the class `Calendar` for example, the number of its fields is over 50. Therefore, we have to create more than 50 new transitions to the Petri net. More importantly, all these transitions can convert type `Calendar` to `int`, which will enlarge the search space a lot.

We solve this problem with the help of the probabilistic model in a similar way. We extract the data from Javadoc for each field accompanied by their description. We also count the frequency of occurrences of each field in the Stack Overflow dataset. Then we assign a score for each `new_field` transition in the same way as other transitions in PRG. In this way, the fields more related to the programming task will be preferred to be used and the efficiency of synthesis can be improved. This process is similar and will not be given in detail.

5 Optimization Strategies

In this section, we propose some optimization strategies to further improve the performance of program synthesis. They are strategies of pruning the model and handling inheritance relationships.

5.1 Pruning Strategy

Given the initial and target markings, there exist many reachable paths in PRG. Normally, enumerating and testing every firing sequence will be time-consuming and unnecessary. Since the synthesis process searches the solution in increasing order of length, when enumerating a reachable path of length k, it does not need to search the places and transitions which only appear in reachable paths with the length larger than k. We can prune these places and transitions in each search round, and search reachable paths in the pruned Petri net. The following theorem will ensure the correctness of pruning.

Theorem 1. *Given a Petri net N, $\mathbb{S}$ denotes the set of places of the input types in conjunction with the place of type `void`. $\mathbb{T}$ denotes the place of the output type. For each place p, let $d_{\mathbb{S}}$ denote the shortest distance from the elements in $\mathbb{S}$ to p and $d_{\mathbb{T}}$ denote the shortest distance from place p to target place $\mathbb{T}$ in the induced graph $\alpha(N)$. If $d_{\mathbb{S}} + d_{\mathbb{T}} > k$, and let M be a marking such that $M(p) > 0$, then there is no path of length k in the reachability graph from initial making M_0 to target making M^* that passes through M.*

Proof. We prove the theorem by reduction to absurdity. Suppose that there exists a path of length k that passes M from initial marking M_0 to target marking M^*. Let k_1 be the length of the path from M_0 to M and k_2 be the length of the path from M to M^* in the reachability graph. Then $k_1 + k_2 = k$. Since $d_{\mathbb{S}}$ and $d_{\mathbb{T}}$ are the shortest distances from the elements in set $\mathbb{S}$ to p and from p to the element in set $\mathbb{T}$ respectively, we have $k_1 \geqslant d_{\mathbb{S}}$ and $k_2 \geqslant d_{\mathbb{T}}$. Then $k_1 + k_2 \geqslant d_{\mathbb{S}} + d_{\mathbb{T}}$. We have $d_{\mathbb{S}} + d_{\mathbb{T}} > k$, thereby $k_1 + k_2 > k$ which contradicts $k_1 + k_2 = k$. □

Recall the Petri net in Fig.3. There exists a reachable path of length $k = 4$, as shown in Fig.14, where T_1–T_3 represent transitions `toLocalDate`, `daysBetween` and `getDays` respectively. We have $\mathbb{S} = \{$`DateTime`, `void`$\}$ and $\mathbb{T} = \{$`int`$\}$. For place `LocalDate`, the shortest distance from the elements in $\mathbb{S}$ to `LocalDate` (from `DateTime` to `LocalDate`) $d_{\mathbb{S}} = 1$ and the shortest distance from `LocalDate` to target place `int` $d_{\mathbb{T}} = 2$. Then we have $d_{\mathbb{S}} + d_{\mathbb{T}} = 3 < 4$.

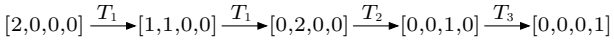

Fig.14. Reachable path in the reachability graph.

According to this theorem, when enumerating a reachable path of length k in the reachability graph, we do not need to consider the marking that assigns non-zero tokens to p, if $d_{\mathbb{S}} + d_{\mathbb{T}} > k$. Therefore, we can prune place p from the Petri net without affecting the completeness. For each transition, if either one of its preplaces or postplaces is pruned, we prune the transition from the Petri net.

The reachable paths are enumerated in increasing order of length. For each length k, we first prune the model using k and then get a pruned Petri net N_k. The PRG is constructed based on the pruned Petri net N_k. For different lengths of the reachable paths, we search in different pruned Petri nets. In this way, the search space can be effectively narrowed down and the efficiency of synthesis can be improved.

5.2 Handling Inheritance Relationships

Example 3. The user desires a method named "daysOfMonth" with two arguments of input type `String`: one represents the date and the other represents the date format. The output type is `int`. The intent of this programming task is to get the number of days in the month from a date string.

One solution program of the desired method is shown in Fig.15.

```
int daysOfMonth(String arg0, String arg1) {
    DateTimeFormatter v1 = DateTimeFormat.
        forPattern(arg1);
    DateTime v2 = DateTime.parse(arg0, v1);
    DateTime.Property v3 = v2.dayOfMonth();
    int v4 = v3.getMaximumValue();
    return v4;
}
```

Fig.15. Solution program of example 3.

Part of the original Petri net is shown in Fig.16 (without the transition and arcs in the dotted line). We omit the weight of the arcs and clone transitions for simplicity.

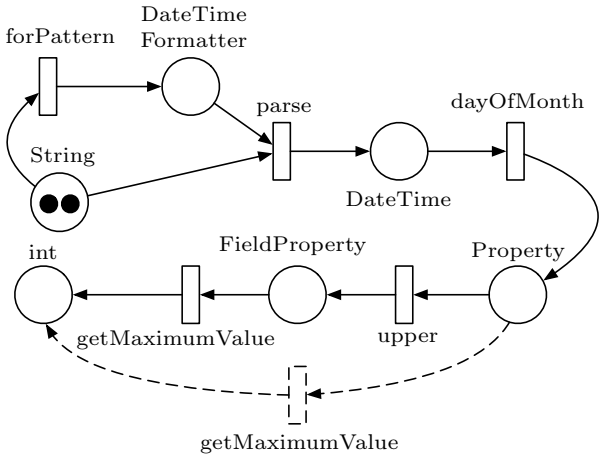

Fig.16. Petri net of handling inheritance relationships.

In the API library of `org.joda.time`, the method `getMaximumValue` is in class `FieldProperty` but not `Property`. Actually, the class `Property` can invoke the `getMaximumValue` method because it inherits from `FieldProperty`. But when constructing the Petri net, there is no transition converting `Property` to `int` directly.

To solve this problem, SyPet creates a kind of transitions named the `upper` transition, which starts from a subclass and ends in the superclass. It can convert an object from the subclass to its superclass. By this means, `Property` can invoke the method of its superclass. Therefore, the length of the firing sequence enumerated by this means is 5, which is shown in Fig.17.

```
forPattern; parse; dayOfMonth; upper;
getMaximumValue;
```

Fig.17. Firing sequence of length 5.

The `upper` transition is not so natural and increases the length of the firing sequence. Therefore, we propose a strategy to improve it. For each superclass in the Petri net, we create places for its subclasses, if they do not exist in the Petri net. For each transition connected to the superclass, we copy this transition and connect it to its subclass in the same way. Considering the task in example 3, we copy the transition `getMaximumValue` for class `Property` instead of the `upper` transition, which can convert type `Property` to `int` directly, as shown in the dotted line in Fig.16. In this way, the firing sequence enumerated has the length of 4 as shown in Fig.18.

```
forPattern; parse; dayOfMonth; getMaximumValue;
```

Fig.18. Firing sequence of length 4.

By this means, we can shorten the length of the firing sequence and improve the search efficiency eventually. In addition, it handles inheritance relationships in a more natural way. Note that SyPet does not analyze the inheritance relationships between classes automatically. Actually, it uses a configuration file to indicate the inheritance relationships offline, which needs to be provided by the user beforehand. In our approach, we follow the manner of SyPet. Due to this, the number of new-added transitions for handling inheritance relationships is limited, which will not bring too much overhead to the performance of ProSy. Analyzing inheritance relationships automatically at runtime will be a better choice, but it will also be a challenge to deal with the overhead brought by this, which will be studied in our future work.

6 Empirical Evaluation

In this section, we evaluate our approach with some experiments which aim to address the following research questions.

*RQ*1. How effective is ProSy compared with the baselines?

*RQ*2. How much improvement can the probabilistic model and the optimization strategies bring to the original approach?

*RQ*3. What is the generality of our approach?

*RQ*4. How does ProSy perform under different parameter settings?

6.1 Experimental Setup

6.1.1 Benchmarks

In order to answer these questions, we conduct several experiments with different configurations. To compare our work with the baselines, we use the same 30 programming tasks of SyPet as the benchmark. These 30 tasks can be divided into four domains, i.e., Math (1–9), Geometry (11–15), Time (16–22), and XML (23–30), as shown in Table 1. Each task has a natural language description to show their intent. The column of "#APIs" indicates the number of component APIs in the solution of the task. The column of "#Test Cases" indicates the number of test cases used to verify the

Table 1. Summary of the 30 Programming Tasks

ID	Description	#APIs	#Test Cases
1	Compute the pseudo-inverse of a matrix	3	1
2	Compute the inner product between two vectors	3	1
3	Determine the roots of a polynomial equation	3	1
4	Compute the singular value decomposition of a matrix	3	1
5	Invert a square matrix	3	1
6	Solve a system of linear equations	6	1
7	Compute the outer product between two vectors	4	1
8	Predict a value from a sample by linear regression	6	2
9	Compute the i-th eigenvalue of a matrix	6	2
10	Scale a rectangle by a given ratio	4	1
11	Shear a rectangle and get its tight rectangular bounds	4	1
12	Rotate a rectangle about the origin by the specified number of quadrants	4	1
13	Rotate two-dimensional geometry object by the specified angle about a point	5	2
14	Perform a translation on a given rectangle	4	1
15	Compute the intersection of a rectangle and the rectangular bounds of an ellipse	3	1
16	Compute the number of days since the specified date	3	2
17	Compute the number of days between two dates considering timezone	3	3
18	Determine if a given year is a leap year	4	3
19	Return the day of a date string	3	2
20	Find the number of days of a month in a date string	4	2
21	Find the day of the week of a date string	4	2
22	Compute the age given the date of birth	3	3
23	Compute the offset for a specified line in a document	3	1
24	Get a paragraph element given its offset in a document	4	1
25	Obtain the title of a webpage specified by a URL	3	1
26	Return doctype of an XML document generated by a string	6	1
27	Generate an XML element from a string	6	1
28	Read an XML document from a file	3	1
29	Generate an XML from file and query it using XPath	7	1
30	Read an XML document from a file and get the value of root attribute specified by a string	5	1

Note: #: number of.

correctness of each task. The experimental results on these tasks are shown in Table 2.

In addition, we collect 20 more programming tasks from Stack Overflow and Java Cookbook [19], which cover the domains like Regular expression, Json parser, XML, Time and Date to further study the performance of ProSy on new tasks against the baselines. Some of these tasks are collected manually by analyzing the most frequently asked questions from Stack Overflow. Others are extracted from the examples in Java Cookbook. These tasks are listed in Table 3.

We collect 16 extra programming tasks that involve field accesses, which cover domains of Calendar, BigDecimal, XPath, Regular expression, and DateFormat to evaluate the strategy of handling field accesses in ProSy. These tasks are collected by referring to the Javadoc to analyze the fields of popular classes and then retrieve similar examples from Stack Overflow posts. These tasks are shown in Table 4.

To evaluate the generality of our approach, we implement the probabilistic model on another state-of-the-art API-based synthesizer FrAngel [13] and compare the performance on the 95 tasks provided in FrAngel paper [13]. Note that all these tasks do not appear in the Stack Overflow dataset for constructing PRG in Section 4.

6.1.2 Metrics

In order to evaluate the performance of ProSy quantitatively, we use the synthesis time as the metric. In all the experiments, we set TIMEOUT to 30 minutes. If the synthesis of a task reports TIMEOUT, 1 800 seconds will be recorded as the time cost of this task when calculating the average synthesis time. The synthesis time of a task in our experiments includes all the time for PRG construction, sketch generation and comple-

Table 2. Experimental Results of ProSy and SyPet on the 30 Programming Tasks

ID	Synthesis Time		#Places		#Transitions		#Sketches		#Programs	
	SyPet	ProSy	SyPet	ProSy	SyPet	ProSy	SyPet	ProSy	SyPet	ProSy
1	13.82	**5.09**	85.00	30.00	1 107.00	435.00	156.00	126.00	311.00	251.00
2	**0.56**	0.67	466.00	201.00	3 886.00	1 639.00	1.00	3.00	1.00	9.00
3	1.55	**0.40**	721.00	228.00	5 729.00	2 996.00	7.00	1.00	13.00	1.00
4	0.49	**0.38**	220.00	6.00	2 376.00	35.00	1.00	1.00	1.00	1.00
5	2.04	**0.72**	467.00	107.00	3 888.00	745.00	16.00	13.00	31.00	25.00
6	499.16	**59.69**	91.00	72.00	1 130.00	840.00	4 099.00	1 646.00	8 671.00	3 507.00
7	5.87	**0.91**	467.00	304.00	3 888.00	2 598.00	14.00	4.00	48.00	12.00
8	72.69	**0.86**	81.00	49.00	856.00	629.00	434.00	12.00	886.00	23.00
9	1 224.40	**220.81**	91.00	80.00	1 130.00	894.00	5 861.00	5 219.00	16 469.00	11 935.00
10	14.57	**1.94**	44.00	28.00	422.00	322.00	78.00	43.00	275.00	134.00
11	14.47	**1.71**	44.00	28.00	422.00	322.00	79.00	43.00	282.00	134.00
12	1.64	**0.61**	44.00	28.00	422.00	322.00	9.00	9.00	21.00	17.00
13	12.69	**5.04**	44.00	30.00	422.00	359.00	67.00	101.00	225.00	348.00
14	8.25	**1.82**	44.00	28.00	422.00	322.00	41.00	41.00	156.00	128.00
15	**0.38**	0.44	44.00	29.00	422.00	325.00	1.00	2.00	1.00	4.00
16	9.40	**1.81**	162.00	69.00	2 829.00	1 548.00	78.00	89.00	156.00	179.00
17	158.04	**5.19**	162.00	66.00	2 829.00	1 434.00	469.00	107.00	2 262.00	376.00
18	90.52	**14.00**	162.00	105.00	2 829.00	1 902.00	624.00	501.00	1 247.00	1 001.00
19	**0.96**	1.21	162.00	70.00	2 829.00	1 437.00	1.00	4.00	1.00	10.00
20	70.05	**61.52**	162.00	97.00	2 829.00	1 892.00	211.00	1 754.00	633.00	5 458.00
21	44.49	**31.94**	162.00	101.00	2 829.00	1 891.00	198.00	436.00	592.00	1 306.00
22	22.61	**3.63**	162.00	66.00	2 829.00	1 434.00	202.00	219.00	408.00	450.00
23	**0.76**	0.79	223.00	70.00	1 584.00	506.00	3.00	9.00	5.00	17.00
24	4.18	**1.16**	223.00	113.00	1 584.00	743.00	33.00	18.00	65.00	35.00
25	23.43	**14.12**	92.00	59.00	710.00	531.00	189.00	155.00	377.00	309.00
26	1.62	**1.26**	181.00	68.00	1 740.00	579.00	6.00	20.00	11.00	39.00
27	3.15	**1.49**	181.00	68.00	1 740.00	579.00	26.00	44.00	51.00	87.00
28	0.46	**0.39**	181.00	21.00	1 740.00	150.00	1.00	1.00	1.00	1.00
29	1.01	**0.81**	161.00	60.00	1 454.00	391.00	2.00	2.00	3.00	3.00
30	**0.81**	1.19	181.00	67.00	1 740.00	578.00	3.00	25.00	5.00	49.00
Average	76.80	14.72	183.67	78.27	1 953.90	945.93	430.33	354.93	1 106.93	861.63

Note: #: number of.

tion, candidate program compilation, as well as the time of running test cases[6]. We also record the number of places and transitions to evaluate the scale of program space. The numbers of candidate sketches and programs generated are also recorded to show the efficiency of each approach.

Note that the correctness of each programming task is verified by running the test cases. There exist cases like that a generated program passes all the test cases. But it does not fully meet the user intent. In this case, it needs more rounds of user interaction to provide additional test cases. We only record the statistics for the last round.

All these experiments are conducted on Ubuntu 16.04 OS with Intel Xeon E5-2682 v4 CPU and a RAM of 64 GB.

6.2 Comparison Against the Baselines (RQ1)

To compare ProSy with SyPet, we first use them to generate the solution for the 30 tasks in Table 1. The results are shown in Table 2. The "ProSy" and "SyPet" columns indicate our approach (full version with all the optimizations) and SyPet (also with its optimizing package) respectively. The better values of the synthesis time are in bold.

The average synthesis time of ProSy is 14.72 seconds compared with 76.80 seconds of SyPet. On average, our approach can reduce the synthesis time of

[6]In SyPet paper, the synthesis time does not include the time of compiling the candidate programs. However, we think this part of the time is also an essential part of the synthesis process. Therefore we consider this part of the time when calculating the synthesis time.

Table 3. Summary of the Experimental Results on 20 Extra Programming Tasks

ID	Description	#APIs	#Test Cases	Synthesis Time (s)	
				SyPet	ProSy
31	Replace the string with regular expression	3	2	T/O	0.54
32	Find the matched string with regular expression	6	2	T/O	23.71
33	Parse json to string with JSON	2	2	2.13	0.95
34	Parse json to string with Gson	5	2	15.93	1.34
35	Load XML from the string	5	1	1.49	1.16
36	Load an org.w3c.dom.Document from XML in a string	5	1	T/O	0.85
37	Get the week year of a date	4	2	4.70	4.21
38	Get the last day of a week	4	2	T/O	19.49
39	Get the specified day of next n months	4	2	239.88	14.58
40	Determine if the date is several months ago	4	3	T/O	33.75
41	Change time zone of a specified time	5	2	T/O	4.91
42	Generate a current date stamp in a specified format	3	1	0.74	0.60
43	Cast a string to an SQL time	4	2	T/O	1.93
44	Compare two dates	5	2	34.98	4.08
45	Get the date when file was last time modified	4	1	T/O	7.63
46	Convert hex string to float	3	2	T/O	0.42
47	Read an integer from a file	3	1	1.19	0.55
48	Distance from a point to a line	2	3	T/O	0.54
49	Get division result using BigFraction	3	2	T/O	19.43
50	Reverse a string and append the result to it	6	2	T/O	5.43
Average				1 095.05	7.31

Table 4. Summary of the Experimental Results for Strategy of Handling Field Accesses

ID	Description	#APIs	#Test Cases	Synthesis Time (s)	
				SyPet	ProSy
51	Get the day of a year	5	2	T/O	2.98
52	Get the day of a week	5	2	T/O	2.95
53	Get the last day of a month	5	3	T/O	3.32
54	Get the week number of a year	5	2	T/O	5.47
55	Format a number towards positive infinity	4	2	T/O	3.61
56	Format a number towards negative infinity	4	2	T/O	3.67
57	Format a number towards zero	4	3	T/O	3.51
58	Format a number away from zero	4	2	T/O	3.19
59	Query a document using XPath to get the text of node	5	1	T/O	0.60
60	Query a document using XPath to get the number of node	5	1	T/O	0.81
61	Query a document using XPath to get the value of node	5	1	T/O	0.66
62	Match a string with a pattern enabling multiline mode	4	2	T/O	3.76
63	Match a string with a pattern enabling case-insensitive matching	4	2	T/O	4.15
64	Match a string with a pattern permitting whitespace and comments in pattern	4	2	T/O	3.63
65	Get the specified date in a short style	3	1	T/O	0.63
66	Get the specified date in a long style	3	1	T/O	0.48
Average				N/A	2.71

SyPet by 80.83%, which indicates that ProSy is obviously more efficient than SyPet. In 25 of 30 tasks, ProSy spends less time than SyPet. For the five remaining tasks (2, 15, 19, 23, and 30), the overhead of constructing a probabilistic model and other optimization strategies is the main reason why ProSy spends more time since the time needed for these tasks is quite short. We can see that ProSy can significantly reduce the synthesis time especially for those complicated tasks like 6, 9, and 17, which shows the potential of ProSy in solving more complicated programming tasks.

The "#Places" and "#Transitions" columns represent the number of places and transitions in the Petri net respectively, which can represent the scale of the

search space. Note that due to our pruning strategy, ProSy searches reachable paths of different lengths in different pruned Petri nets. Therefore, the numbers in the table represent the statistics of the pruned Petri net from which we finally get a satisfiable solution. As shown in the table, the average numbers of places and transitions in ProSy are 78.27 and 945.93 respectively while they are 183.67 and 1 953.90 in SyPet respectively. With the adoption of the pruning strategy, our approach can reduce the number of places by 57.38% and reduce the number of transitions by 51.58%. Constructing PRG based on the pruned Petri net can further speed up the search process.

The columns "#Sketches" and "#Programs" represent the number of candidate sketches and programs generated of each task respectively. As shown in Table 2, the average number of sketches is reduced from 430.33 to 354.93 by 17.52%. The average number of programs is reduced from 1 106.93 to 861.63 by 22.16%. In most of the 30 tasks, the numbers of sketches and programs are reduced. There are some special cases like tasks 20 and 21. The numbers of sketches and programs are increasing, but the synthesis time is reduced. The reason is the strategy of handling inheritance relationships. This strategy creates some new transitions into the Petri net, which increases the number of reachable paths. Due to the PRG model and other optimization strategies, the synthesis time is reduced.

Besides these 30 tasks, we collect 20 more tasks to further study the performance of our approach. The experimental results are shown in Table 3. "T/O" means that the corresponding task cannot be solved before timeout.

We can see that among the 20 tasks, ProSy can solve each of them within 60 seconds (7.31 seconds on average), while SyPet can only solve eight of them before timeout. It should be noted that in this experiment, SyPet does not use its optimizing function, and does not consider the inheritance relationships. It is because that the packages for handling these two aspects should be modified for these new tasks, but they are not open-sourced by SyPet.

In the optimizing function, SyPet needs the preprocessed data from Javadoc. The runnable version of SyPet only contains the data needed for the 30 programming tasks in the paper. For new tasks, the optimizing function will not work without these data. For inheritance relationships, SyPet does not analyze them automatically at runtime. It uses a configuration file to indicate the inheritance relationships. For new tasks, if the inheritance relationships are not indicated, there will be no reachable paths in the reachability path so that no solution can be found before timeout sometimes.

Now let us analyze the 40th programming task as an example. The user intent is to determine if the date is several months ago from now. The solution program is shown in Fig.19.

```
boolean beforeSeveralMonths(String date, int
        month) {
    DateTime now = DateTime.now();
    DateTime date1 = now.minusMonths(month);
    DateTime date2 = DateTime.parse(date);
    boolean res = date1.isAfter(date2);
    return res;
}
```

Fig.19. Solution program of task 40.

ProSy can solve it in 33.75 seconds while SyPet cannot solve it before timeout. There are two main reasons for this problem. One is that the number of places and transitions in the Petri net of SyPet is nearly twice of the number in the pruned Petri net of ProSy, which leads to that the solution space of SyPet is exponential times of that of ProSy. The other is that ProSy enumerates the reachable path with the highest probability in the PRG first, which can significantly improve the efficiency of program synthesis.

To further study the performance of ProSy, we conduct an experiment to compare ProSy with another API-based synthesizer Bayou[20]. Bayou can synthesize API-heavy code in Java using a Bayesian statistical approach. Given a set of evidence such as API calls or data types, it can predict a sketch of the program based on the evidence. Then Bayou completes the sketch into code. We extract all the APIs and data types from the solution of each task as the evidence of Bayou. The programs returned by Bayou are ranked according to their probabilities. To evaluate the performance of Bayou, we choose the top 10 solutions generated by Bayou as the candidate solutions. If the top 10 solutions contain the desired one, then we deem that this task can be solved by Bayou[7].

We divide the 50 programming tasks in Table 1 and Table 3 into six categories, i.e., Math, Geometry, Time, XML, Date, and Others. The experimental results are shown in Fig.20, which indicates how many tasks can

[7]Note that we do not train our own model for Bayou. We use the online version of Bayou at http://www.askbayou.com, Jan. 2019.

be solved by each tool. Among these tasks, ProSy can solve all of them, while Bayou can only solve six of them when enough evidence is provided (three in the domain of XML, two in Date, and one in Others).

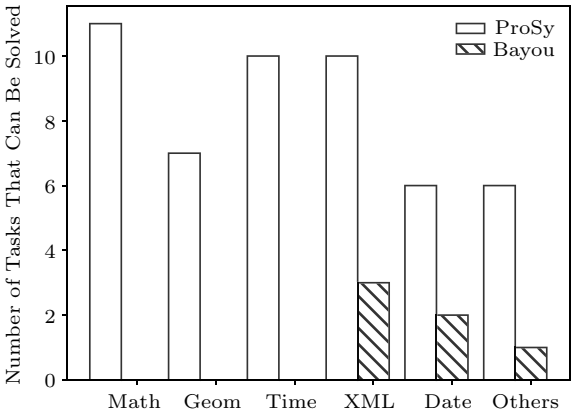

Fig.20. Comparison with Bayou on 50 programming tasks.

In this experiment, we only evaluate the ability of ProSy and Bayou on synthesizing correct solutions for the same set of tasks. Comparing these two synthesizers in a fair and systematic way is difficult. Firstly, the technical routes of ProSy and Bayou are different. ProSy is a search-based approach. It uses the Petri net to model the API usage and performs a reachability analysis on it to generate programs. Bayou is a deep-learning based approach. It predicts programs based on the generative model and the model needs to be trained on a large dataset. Secondly, the inputs of these two methods are different. The inputs of ProSy include a method signature, a natural language description, a candidate API library, and test cases. The input of Bayou is a set of evidence such as API calls or data types. In our experiment, we provide all the APIs and data types from the solution of each task as the evidence of Bayou. Thirdly, the way of validating candidate programs of these two methods is different. ProSy evaluates them on test cases provided by users. The returned program is ensured to pass all the test cases (if it exists). Bayou can only ensure that the programs generated are free from grammatical errors. But the correctness of the programs cannot be ensured. Fourthly, the metrics of evaluating these two methods are different. In ProSy, we set a timeout to 30 minutes and Bayou does not have a time limit. It only returns candidate programs which are ranked according to the trained generative model.

Next, we experiment on 16 tasks to evaluate the strategy of handling field accesses. The results are shown in Table 4. "N/A" means "not applicable". All these tasks cannot be solved by SyPet but can be solved by ProSy in a short period of time (2.71 seconds on average), which indicates that the strategy of handling field accesses is very useful to synthesize such kind of tasks. By using this strategy, we can extend the capability of our approach.

Based on the experimental results above, we can make the following conclusion.

*Answer to RQ*1. ProSy can significantly improve the efficiency of synthesis and narrow down the search space compared with SyPet. The strategy of handling field accesses can extend the capability of our approach.

6.3 Evaluation of PRG and Optimization Strategies (RQ2)

To evaluate the performance of PRG and the optimization strategies in our approach, we experiment on the 30 programming tasks of SyPet together with the 20 newly collected tasks with different configurations. The experimental results are shown in Table 5. The numbers of sketches and programs in the table only calculate the average value of those tasks that can be solved before timeout. The statistics for those tasks that cannot be solved are not recorded.

Table 5. Summary of the Experimental Results of ProSy Under Different Configurations

	Success Rate	Synthesis Time (s)	#Sketches	#Programs
ProSy	50/50	11.75	282.76	761.32
Only PRG	43/50	275.08	472.88	1 277.79
Only Pruning	41/50	352.09	518.07	1 601.28
Only Inheri	48/50	105.84	320.16	1 060.08
None	41/50	356.73	869.02	2 757.59

"ProSy" indicates the full version of ProSy with PRG and all the optimization strategies. "Only PRG" indicates the version with only the probabilistic model, and "Only Pruning" indicates the version with only the pruning strategy. "Only Inheri" indicates the version with only the strategy of handling inheritance relationships. "None" indicates the original version without any optimization strategies[⑧].

[⑧]Note that "None" is the open-source version of SyPet. SyPet adopts two optimizing strategies of pruning and objective function, which are removed from the version of "None". ProSy and SyPet are implemented on top of the "None" version.

The numerator in the fraction of the success rate represents the number of tasks that can be solved. The denominator represents the total number of tasks. The numbers of tasks that can be synthesized by "Only PRG", "Only Pruning" "Only Inheri" and "None" are 43, 41, 48, and 41 respectively. The average synthesis time of "ProSy", "Only PRG", "Only Pruning" "Only Inheri" and "None" is 11.75, 275.08, 352.09, 105.84, and 356.73 seconds respectively.

Comparing "Only PRG" with "None", we can see that by using the probabilistic model, we can reduce 22.88% of the synthesis time on average. The numbers of candidate sketches and programs are reduced by 45.58% and 53.66% respectively. Comparing "Only Pruning" with "None", we can see that 1.3% of the synthesis time can be reduced on average by using the pruning strategy. That is because both two versions of ProSy cannot solve nine tasks out of 50, which enlarges the average synthesis time. The numbers of candidate sketches and programs are reduced by 40.38% and 41.93% respectively. Comparing "Only Inheri" with "None", we can see that 70.33% of the synthesis time can be reduced on average by using the strategy of handling inheritance relationships. The numbers of candidate sketches and programs are reduced by 63.15% and 61.55% respectively.

We can see that the probabilistic model and the optimization strategies of pruning and handling inheritance relationships can effectively reduce the synthesis time and eventually improve the efficiency.

Another view of the results of synthesis time is shown in Fig.21, which can display the dispersal of the results. The horizontal line at the top and the bottom represents the maximum and minimum value of the synthesis time respectively. The top and the bottom edge of the box represent the third quartile and the first quartile value of the synthesis time respectively. The red line and the green dotted line in the middle of the box represent the median value and the mean value of synthesis time respectively. Note that we take the natural logarithm for each synthesis time to deal with the problem of data sparsity.

We can see that the median value, mean value, as well as the first and the third quartile values, are all reduced in "Only PRG", "Only Pruning", and "Only Inheri" compared with "None", which means the probabilistic model and the strategies of pruning and handling inheritance relationships can reduce the synthesis time.

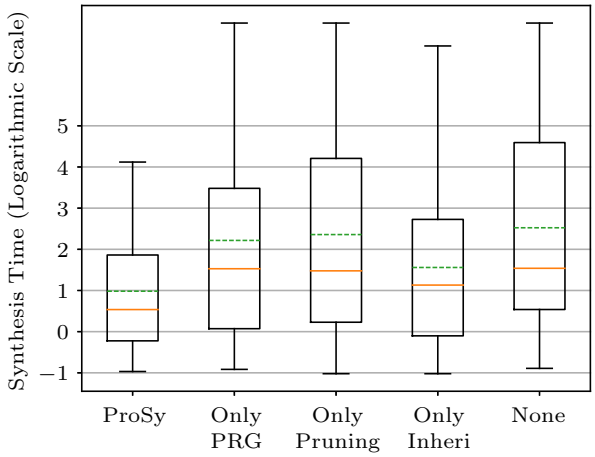

Fig.21. Synthesis time (natural logarithm) of 50 tasks on different configurations.

From the experimental results above, we can conclude as follows.

*Answer to RQ*2. Both PRG and the optimization strategies of the pruning model and handling inheritance relationships can improve the efficiency of program synthesis.

6.4 Evaluation of Generality of ProSy (RQ3)

The approach of acquiring knowledge from various code resources and constructing a probabilistic model to reduce the program space is a general approach, which can be applied to any API-based synthesizers. To further evaluate the generality of our approach, we implement it on another API-based synthesizer FrAngel [13]. FrAngel can generate Java programs that are composed of APIs and control structures. The input of FrAngel is the same as that of SyPet. It generates programs by randomly sampling programs and evaluating them on the test cases. The basic procedure of FrAngel is a random search.

FrAngel[9] first analyzes the APIs in the component library specified by the user and then uses these APIs as basic fragments. In synthesis, it randomly samples from these fragments to generate programs. To implement our approach on it, we build the probability calculator as we do in ProSy to obtain the probability of each API in the library. In FrAngel, it samples fragments all at random. Unlike the original approach, we also build a probabilistic model and use the score of each API to guide the sampling process. The API with a higher score has a higher rate to be sampled. We then experiment on it compared with the original version.

[9] https://github.com/kensens/FrAngel, Oct. 2020.

We use the 95 programming tasks (group SyPet, Geometry, and Control Structures) provided in FrAngel [13]. We do not experiment on the 25 tasks in group GitHub, because these tasks evolve too many project-specific and self-defined APIs, which is unable to retrieve knowledge about them. FrAngel uses a random search in synthesizing. Therefore we run each version of FrAngel 10 times and record the average value of the statistics of these 10 runs. The experimental results are shown in Table 6.

The column "FrAngel" represents the open-source version of FrAngel. The column "Prob. Model" represents the modified version of FrAngel with the probabilistic model. We record the success rate, synthesis time, and the number of candidate programs generated for each benchmark group. For the tasks in group SyPet, the average synthesis time is reduced by 56.09%, and the number of candidate programs is reduced by 51.77% on average. For the tasks in group Geometry, the average synthesis time is reduced by 3.84%, and the number of candidate programs is reduced by 4.98% on average. For the tasks in group Control Structures, the average synthesis time is reduced by 23.34%, and the number of candidate programs is reduced by 24.58% on average.

In group SyPet and Control Structures, the synthesis time and the number of candidate programs are both reduced significantly. However, the improvement in group Geometry is only a little. In the SyPet group, the component library is specified by package name like `java.awt.geom`, which contains many classes. The number of APIs in these classes is quite large. In group Geometry, some tasks' component libraries are not specified by the user but inferred from the input and output types. In some tasks, the component library only contains one or two classes. The number of APIs in it is often small. That is the reason why our approach cannot bring too much speedup for tasks in group Geometry.

Another view of the experimental results is shown in Fig.22. It shows the success rate of synthesizing in 30 minutes for different benchmark groups. The dotted line represents the modified version and the solid line represents the original version. Different groups of tasks are represented in different colors. For each benchmark group, our modified version can generate more tasks in less time than the original version of FrAngel. The experimental result shows that by combining the probabilistic model, the performance of synthesis can be effectively improved.

From the experimental results above, we can conclude as follows.

*Answer to RQ*3. The approach of building a probability model to reduce the program space is a general approach, which can be applied to other API-based synthesizers to improve the efficiency of synthesis.

6.5 Sensitivity Analysis of the Weighing Function (RQ4)

In order to evaluate the performance of ProSy under different parameter settings, we conduct an experiment to analyze the weighing parameter in the similarity function. The similarity function is the weighted average of the similarity function for Javadoc and Stack Overflow. Here, we analyze the parameter of ω_1 through an experiment. We vary the parameter from 0 to 1 to evaluate its impact on the synthesis time of the 50 tasks.

We vary ω_1 from 0 to 1 (ω_2 from 1 to 0) incrementing by 0.1 each time and obtain the synthesis time on the 50 benchmarks. The results are shown in Fig.23, where we take the natural logarithm for each synthesis time. We can see that the median value and the mean value of the synthesis time reach the minimum value when ω_1 is 0.5.

*Answer to RQ*4. The value of ω_1 has an effect on the performance of ProSy. The optimum value for ω_1 is 0.5.

7 Related Work

In this section, we discuss some work that is related to our study from four categories.

Table 6. Summary of the Experimental Results on FrAngel

Benchmark Group	Success Rate (%)		Synthesis Time (s)		Number of Programs	
	FrAngel	Prob. Model	FrAngel	Prob. Model	FrAngel	Prob. Model
SyPet (30)	99.33	99.62	72.64	31.89	6 049 159.64	2 917 204.22
Geometry (25)	88.40	91.11	313.74	301.67	5 334 538.03	5 068 772.73
Control structures (40)	93.25	96.66	265.69	203.67	2 603 580.93	1 963 560.80

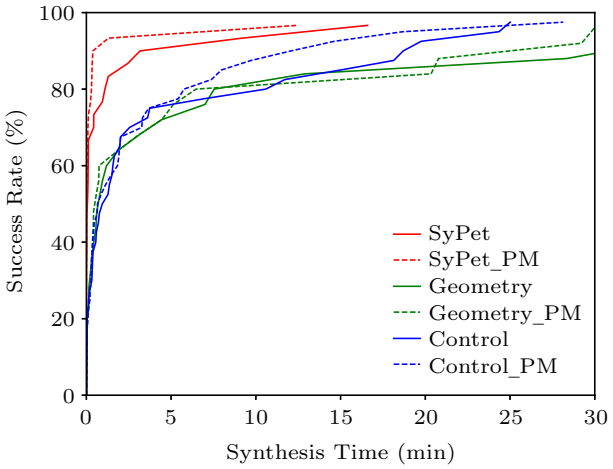

Fig.22. Success rate of synthesizing in 30 minutes of FrAngel.

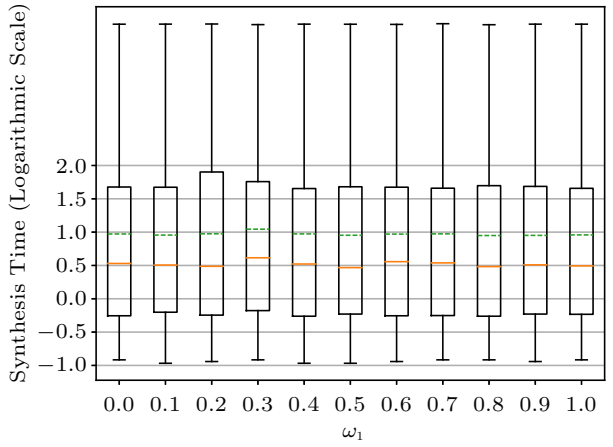

Fig.23. Synthesis time (natural logarithm) of 50 tasks under different parameter settings.

7.1 Program Synthesis

Programming by examples (PBE) is an important sub-field of program synthesis, where the specifications are provided in the form of input-output examples. PBE usually synthesizes programs in an underlying domain-specific language to constrain the program space and combines various techniques to accelerate the search process [3]. The specification is normally very straightforward and easy to collect; therefore it has been studied by many researchers [21–25]. In PBE, the users need to provide a DSL and some input-output examples for the desired program's behavior. Then it can synthesize a ranked set of DSL programs that are consistent with the examples. Unlike these studies, we use input-output pairs only as test cases to verify the correctness of candidate programs.

Component-based synthesis is to construct a loop-free program from a given library of components. It have been widely used in many domain-specific applications including bit-vector programs [5], geometry constructions [26], deobfuscating [11], string manipulation [9], data structure transformation [27], and table transformation [10], etc. In component-based synthesis, users should provide a library of components. Each component is a domain-specific function that may be used in the desired program. Some studies also annotate each component with a specification [11]. Unlike these studies, our work and SyPet [12] as well as FrAngel [13] are designed for synthesizing general Java programs with various intents, and the components here refer to Java APIs. Apart from the component library, we ask the user to provide a natural language description to describe the intent of the desired method, which is then used to acquire API usage knowledge from different code resources.

Program sketching is a synthesis methodology where the intent of a programming task is given by a program sketch and leaves the rest of the implementation to the synthesis procedure [6]. Apart from the sketch, the users should define the behavior of the program through code, either in the form of a reference implementation, or as a set of parameterized unit tests. The synthesis procedure can be done by counterexample-guided inductive synthesis (CEGIS). Similarly, we use an off-the-shelf SAT solver to complete the program sketch. Unlike the above work, the program sketch in our approach is generated automatically instead of provided by users. Bayou [20] is another work which learns a Bayesian probabilistic model from code and uses it to infer program sketches.

Neural programming is to generate programs with deep learning on neural networks. Some approaches build neural networks conditioned on input/output examples and learn to predict programs [7, 8, 28]. These techniques are constrained to generate programs in domain-specific languages. There is also a class of approaches that generate programs in general programming languages like Python from natural language descriptions. Many of them adopt encoder-decoder architectures to predict programs. The attention mechanism and other modules are combined to improve the performance of the model. They have shown the effectiveness on benchmarks like Trading Card Games (TCGs) such as Magic the Gathering and Hearth-stone, as well as benchmarks of semantic parsing [29–31]. Some work builds a neural network to incorporate statistical information to guide the synthesis process [32]. In neu-

ral programming, the user should prepare a dataset to train the neural network and the input to the generative model is consistent with the input of the training phase of the model.

Another work that has much in common with our approach is MARS [33]. It takes as input a multi-layer specification, which is composed of input/output examples, natural language description, and partial code snippets. It also encodes the synthesis problem into a Max-SMT problem. The input/output examples are encoded as hard constraints which have to be satisfied and other specifications are encoded as soft constraints, which are preferably satisfiable. In our work, the synthesis problem is encoded to a PB-problem. The basic rules of enumerating reachable paths are encoded as hard constraints and the probabilities of API usages are encoded as an objective function, which is similar to soft constraints of MARS.

7.2 Code Search

Code Search returns code snippets to users by searching from the codebase. Most approaches in code search are based on techniques of information retrieval, such as Sourcerer [34] and Portfolio [35]. To improve the performance of code search, different approaches are proposed. Some approaches adopt query expansion or reformulation to enhance the specifications [36, 37]. Some approaches take into consideration the characteristics of source code [38, 39]. Many approaches adopt semantic analysis on source code [40–42]. Many approaches also retrieve knowledge from various sources to improve performance [43]. Both the approaches of code search and program synthesis can return code snippets when given a specification. However, the key difference lies in that code search can only retrieve existing code snippets from the codebase, while program synthesis can generate code snippets from scratch that may not exist before.

7.3 API Recommendation

API recommendation aims to return relevant APIs given some kind of specifications. Different approaches have been proposed in recent years. APIREC [44] uses a statistical learning model trained on fine-grained code changes to recommend API calls. DeepAPI [45] trains a recurrent neural network encoder-decoder model to generate API sequences from a given natural language query. BIKER [46] uses both Stack Overflow and API documentation to recommend APIs in class and method level. RecRank [47] applies a ranking-based discriminative approach to rerank the API identified by GraLan. Word2API [48] uses the CBOW model to generate word and API vector embeddings published as a dictionary. Given a word or a query, based on the vector dictionary and the corresponding similarity calculation formula, a ranked list of APIs is returned. The idea of API recommendation is similar to the probabilistic model in our approach. However, our work differs from the approaches above. Firstly, the ways of calculating the probabilities of APIs are different. Different approaches adopt different ways of calculating API probabilities from different sources. Secondly, the purposes of the probabilistic model are different. The probability model used in our approach is to acquire APIs for API-based synthesis. However, the model in API recommendation approaches is mostly used to recommend APIs to programmers.

7.4 Representing Code as Graph

In this paper, we use Petri net to model the relations between APIs and data types. There is some related work which also represents code as a graph. PROSPECTOR [49] uses a signature graph to represent API "Jungloids", the nodes of which are the class types declared by the API and the edges represent the elementary jungloids of the API. Another work models API invocations as an API graph [50] for API reuse where its nodes represent classes or methods and its edges indicate the invocation relationship between the nodes. T2API [51] uses an API usage graph [52] to synthesize the API usage template from a query which can represent data and control dependencies among API elements. Unlike these studies, Petri net represents API methods and data types separately and can represent the procedure of consuming and creating tokens which contains more information on API usage.

8 Conclusions

In this paper, we studied how to improve the performance of API-based synthesis by utilizing the knowledge from various coding resources. We found the major obstacle in existing work is the huge space when searching the solution program for a given task. Thus, we proposed the method of assigning each API in reachability space a probability, which is calculated based on the task description and the API usage knowledge in Javadoc and Stack Overflow. Several extension and

optimization strategies were also studied. The experimental results showed that the proposed method can obtain significant improvement in program synthesis.

Although the proposed method can obtain better capability and performance, there are still more potentialities that can be dug. Now only the probability for individual API is calculated. Actually, APIs are usually used with certain patterns, which may bring more benefits to synthesis since its purpose is to find a proper API sequence. In addition, now the output program has no branches or loops, which is important for real tasks. All these aspects will be considered in our future work.

References

[1] Gulwani S, Polozov O, Singh R. Program synthesis. *Foundations and Trends in Programming Languages*, 2017, 4(1/2): 1-119.

[2] Pnueli A, Rosner R. On the synthesis of a reactive module. In *Proc. the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp.179-190.

[3] Gulwani S, Jain P. Programming by examples: PL meets ML. In *Proc. the 15th Asian Symposium on Programming Languages and Systems*, November 2017, pp.3-20.

[4] Alur R, Bodík R, Juniwal G, Martin M M K, Raghothaman M, Seshia S A, Singh R, Solar-Lezama A, Torlak E, Udupa A. Syntax-guided synthesis. In *Proc. the 13th International Conference on Formal Methods in Computer-Aided Design*, October 2013, pp.1-8.

[5] Gulwani S, Jha S, Tiwari A, Venkatesan R. Synthesis of loop-free programs. In *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2011, pp.62-73.

[6] Solar-Lezama A. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 2013, 15(5/6): 475-495.

[7] Parisotto E, Mohamed A, Singh R, Li L, Zhou D, Kohli P. Neuro-symbolic program synthesis. In *Proc. the 5th International Conference on Learning Representations*, April 2017.

[8] Balog M, Gaunt A L, Brockschmidt M, Nowozin S, Tarlow D. DeepCoder: Learning to write programs. In *Proc. the 5th International Conference on Learning Representations*, April 2017.

[9] Perelman D, Gulwani S, Grossman D, Provost P. Test-driven synthesis. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014, pp.408-418.

[10] Feng Y, Martins R, Geffen J V, Dillig I, Chaudhuri S. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2017, pp.422-436.

[11] Jha S, Gulwani S, Seshia S A, Tiwari A. Oracle-guided component-based program synthesis. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp.215-224.

[12] Feng Y, Martins R, Wang Y, Dillig I, Reps T W. Component-based synthesis for complex APIs. In *Proc. the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2017, pp.599-612.

[13] Shi K, Steinhardt J, Liang P. FrAngel: Component-based synthesis with control structures. *Proc. the ACM on Programming Languages*, 2019, 3(POPL): Article No. 73.

[14] Mei H, Zhang L. Can big data bring a breakthrough for software automation? *SCIENCE CHINA Information Sciences*, 2018, 61(5): Article No. 056101.

[15] Petri C A, Reisig W. Petri net. *Scholarpedia*, 2008, 3(4): Article No. 6477.

[16] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv:1301.3781, 2013. https://arxiv.org/abs/1301.3781, Sept. 2020.

[17] Pennington J, Socher R, Manning C D. Glove: Global vectors for word representation. In *Proc. the 2014 Conference on Empirical Methods in Natural Language Processing*, October 2014, pp.1532-1543.

[18] Eén N, Sörensson N. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2006, 2(1/2/3/4): 1-26.

[19] Darwin I F. Java Cookbook. O'Reilly, 2001.

[20] Murali V, Qi L, Chaudhuri S, Jermaine C. Neural sketch learning for conditional program generation. In *Proc. the 6th International Conference on Learning Representations*, April 2018.

[21] Gulwani S. Automating string processing in spreadsheets using input-output examples. In *Proc. the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2011, pp.317-330.

[22] Singh R, Gulwani S. Learning semantic string transformations from examples. *Proc. the VLDB Endowment*, 2012, 5(8): 740-751.

[23] Albarghouthi A, Gulwani S, Kincaid Z. Recursive program synthesis. In *Proc. the 25th International Conference on Computer Aided Verification*, July 2013, pp.934-950.

[24] Singh R, Gulwani S. Synthesizing number transformations from input-output examples. In *Proc. the 24th International Conference on Computer Aided Verification*, July 2012, pp.634-651.

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

[26] Gulwani S, Korthikanti V A, Tiwari A. Synthesizing geometry constructions. In *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2011, pp.50-61.

[27] Feser J K, Chaudhuri S, Dillig I. Synthesizing data structure transformations from input-output examples. In *Proc. the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015, pp.229-239.

[28] Kalyan A, Mohta A, Polozov O, Batra D, Jain P, Gulwani S. Neural-guided deductive search for real-time program synthesis from examples. In *Proc. the 6th International Conference on Learning Representations*, April 2018.

[29] Ling W, Blunsom P, Grefenstette E, Hermann K M, Kociský T, Wang F, Senior A W. Latent predictor networks for code generation. In *Proc. the 54th Annual Meeting of the Association for Computational Linguistics*, August 2016.

[30] Yin P, Neubig G. A syntactic neural model for general purpose code generation. In *Proc. the 55th Annual Meeting of the Association for Computational Linguistics*, July 2017, pp.440-450.

[31] Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing. In *Proc. the 55th Annual Meeting of the Association for Computational Linguistics*, July 2017, pp.1139-1149.

[32] Lee W, Heo K, Alur R, Naik M. Accelerating search-based program synthesis using learned probabilistic models. In *Proc. the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2018, pp.436-449.

[33] Chen Y, Martins R, Feng Y. Maximal multi-layer specification synthesis. In *Proc. the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, August 2019, pp.602-612.

[34] Linstead E, Bajracharya S K, Ngo T C, Rigor P, Lopes C V, Baldi P. Sourcerer: Mining and searching Internet-scale software repositories. *Data Min. Knowl. Discov.*, 2009, 18(2): 300-336.

[35] McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C. Portfolio: Finding relevant functions and their usage. In *Proc. the 33rd International Conference on Software Engineering*, May 2011, pp.111-120.

[36] Lu M, Sun X, Wang S, Lo D, Duan Y. Query expansion via WordNet for effective code search. In *Proc. the 22nd IEEE International Conference on Software Analysis*, March 2015, pp.545-549.

[37] Haiduc S, Bavota G, Marcus A, Oliveto R, Lucia A D, Menzies T. Automatic query reformulations for text retrieval in software engineering. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.842-851.

[38] Lv F, Zhang H, Lou J, Wang S, Zhang D, Zhao J. CodeHow: Effective code search based on API understanding and extended Boolean model (E). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering*, November 2015, pp.260-270.

[39] Gu X, Zhang H, Kim S. Deep code search. In *Proc. the 40th International Conference on Software Engineering*, May 2018, pp.933-944.

[40] Ke Y, Stolee K T, Goues C L, Brun Y. Repairing programs with semantic code search (T). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering*, November 2015, pp.295-306.

[41] Reiss S P. Semantics-based code search. In *Proc. the 31st International Conference on Software Engineering*, May 2009, pp.243-253.

[42] Stolee K T, Elbaum S G, Dobos D. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 2014, 23(3): Article No. 26.

[43] Sirres R, Bissyandé T F, Kim D, Lo D, Klein J, Kim K, Traon Y L. Augmenting and structuring user queries to support efficient free-form code search. In *Proc. the 40th International Conference on Software Engineering*, May 2018, pp.945-945.

[44] Nguyen A T, Hilton M, Codoban M, Nguyen H A, Mast L, Rademacher E, Nguyen T N, Dig D. API code recommendation using statistical learning from fine-grained changes. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp.511-522.

[45] Gu X, Zhang H, Zhang D, Kim S. Deep API learning. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp.631-642.

[46] Huang Q, Xia X, Xing Z, Lo D, Wang X. API method recommendation without worrying about the task-API knowledge gap. In *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, September 2018, pp.293-304.

[47] Liu X, Huang L, Ng V. Effective API recommendation without historical software repositories. In *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, September 2018, pp.282-292.

[48] Li X, Jiang H, Kamei Y, Chen X. Bridging semantic gaps between natural languages and APIs with word embedding. arXiv:1810.09723, 2018. https://arxiv.org/abs/1810.09723, Sept. 2020.

[49] Mandelin D, Xu L, Bodík R, Kimelman D. Jungloid mining: Helping to navigate the API jungle. In *Proc. the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005, pp.48-61.

[50] Chan W, Cheng H, Lo D. Searching connected API subgraph via text phrases. In *Proc. the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 2012, Article No. 10.

[51] van Nguyen T, Rigby P C, Nguyen A T, Karanfil M, Nguyen T N. T2API: Synthesizing API code usage templates from English texts with statistical translation. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, pp.1013-1017.

[52] Nguyen T T, Nguyen H A, Pham N H, Al-Kofahi J M, Nguyen T N. Graph-based mining of multiple object usage patterns. In *Proc. the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, August 2009, pp.383-392.

Bin-Bin Liu is a Ph.D. candidate in software engineering at National University of Defense Technology, Changsha. He received his B.S. degree in computer science and technology and M.S. degree in software engineering from National University of Defense Technology, Changsha, in 2013 and 2015 respectively. His research interests include intelligent methods in software engineering, and data-driven software engineering.

Wei Dong is a professor in College of Computer Science, National University of Defense Technology, Changsha. He received his B.S. and Ph.D. degrees in computer science from National University of Defense Technology, Changsha, in 1997 and 2002 respectively. From 2007 to 2008, he was a visiting scholar in Technical University Munich, Munich. His research interests include program analysis and verification, runtime monitoring of mission-critical systems, and intelligent methods in software engineering.

Jia-Xin Liu is a Master student in software engineering at National University of Defense Technology, Changsha. She received her B.S. degree in software engineering from Hunan Normal University, Changsha, in 2018. Her research interests include program synthesis and API recommendation.

Ya-Ting Zhang is a Master student in software engineering at National University of Defense Technology, Changsha. She received her B.S. degree in software engineering from Hunan Normal University, Changsha, in 2018. Her research interests include program synthesis and machine learning.

Dai-Yan Wang is a Master student in software engineering at National University of Defense Technology, Changsha. She received her B.S. degree in management from China Pharmaceutical University, Nanjing, in 2018. Her research interests include program synthesis and machine learning.