Probabilistic Synthesis for Program with Non-API Operations

Yating Zhang, Wei Dong, Daiyan Wang, Jiaxin Liu and Binbin Liu
College of Computer Science, National University of Defense Technology, Changsha, China
{zhangyating18, wdong}@nudt.edu.cn, daiyanw9610@163.com, {liujiaxin18, liubinbin09}@nudt.edu.cn

***Abstract*—The current approaches to program synthesis focus more on works that contain only API methods. However, our survey shows that the ratio of non-API operations to API calls is very close in Java programs. Although non-API operations are difficult to obtain specific information from user intent, those operations in such as mathematics and logic are frequent and important, and that is an indispensable key in practical development tasks. We present CANA (Capsulate Non-API), an improved approach of probabilistic synthesis for non-API operations. CANA synthesizes programs with non-API operations through two main ideas. One is to encapsulate non-API operations into API calls, and the user also can provide related information when describing incomplete specifications. The other is heuristic strategies to solve the difficult problems that select variables with the same type. Experiments show that CANA can synthesize programs contains combinations of non-API operations in seconds.**

***Index Terms*—Probabilistic Program Synthesis, Non-API Operations**

I. INTRODUCTION

Program synthesis is the task of automatically synthesizing the program that realizes the user's intention [1]. The intent of the user expresses the desired program function, and there are usually natural language descriptions [2], input-output examples [3], and other types [4]. The current approaches [5] to synthesis usually frame the problem by searching in the space of program [6] to find the target result. These methods continue to face a core challenge, namely, that it is difficult to find the target program from the huge search space in limited time [7]. Moreover, the generated code is usually simple in function, such as focusing only on generating the Application Programming Interface (API) calls. Other than that are non-API operations, which refer to all processing in code other than API calls from the library or package, including binary operators, variable initialization, array handling, and so on.

There are two reasons for pure-API generation. One is that the API calls usually have the sequences relationships [8], which are easy to discover and learn. For example, the return type of an API call is frequently used as the parameter of the next API call, which also inspires many program synthesis based on components or searches [3]. Many kinds of researches [9] [8] [10], therefore, construct a probabilistic model to mine and learn the sequence of API calls, then solve the problem of code generation. On the other hand, non-API operations (such as infix expressions) usually occur in the same type and certain constant values. There are also variable assignment expressions where the right side is a constant. These pieces of information are difficult to obtain from user intent. For example, for an plus operation between two integers $i1$ and $i2$, the expected value is $i1 + i2$, and the result may become $i1 + i1$.

Nevertheless, We sampled over 80,000 Java methods on GitHub where do not contain the test method. The result shows that each method has an average of 8.23 lines, 2.51 API calls, and 2.41 non-API operations. This result proves the frequency of non-API operations is almost as same as API calls in the program. Actually, program synthesis based on logic derivation [11] [12] is more conducive to handling such operations, but those approaches assume a complete formal specification of the desired user intent was provided, which in many cases proved to be as complicated as writing the program itself [5]. Although non-API operations are difficult to obtain specific information from user intent, those operations in such as mathematics and logic are frequent and important, and that is an indispensable key in practical development tasks.

To solve these programs, we present CANA (Capsulate Non-API), a framework for probabilistic synthesis that can synthesize Java methods contains non-API operations. CANA synthesizes programs with non-API operations through two main ideas. One is to encapsulate non-API operations into API calls, and then users also can provide related information when describing incomplete specifications. The other is to handle the synthesized results. We divide synthesized programs into two parts. The first is variable initialization and the other is variable selections. We choose initial values from empirical values such as 0, $null$. The second is that we provide heuristic strategies to complete the program, which to solve the difficult problem that select variables with the same type.

We select a program synthesis tool named BAYOU [9] as an example to verify the effectiveness of the framework. BAYOU [9] implies a probability encoder-decoder model that uses Bayesian probabilistic inference and trains a neural generator not on code but on program sketches. The sketch is an abstraction of a program that leaves out low-level information such as variable names, while retaining only high-level information such as API calls and control structures. We grab over 700 high-star Java projects from GitHub, extract over 60,000 methods from those projects to train our models, and adopt 5,000 methods as our test data set. In summary, this

Corresponding author: Wei Dong. This work was supported by National Natural Science Foundation of China (No.61690203, No.61532007) and National Key Reseach and Development Program of China (No.2017YFB1001802).

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00082

paper makes the following contributions:

- We present a framework called CANA for probabilistic synthesis that can synthesize Java methods contains non-API operations, then we take a synthesis tool BAYOU as an example to verify the effectiveness of the framework.
- We construct the domain-specific language of the Java program, which enables non-API operations to participate in probabilistic neural network training and strengthens the relevance to user intentions.
- We provide some heuristic strategies to solve the problem that select variables with the same type.

The rest of this paper is organized as follows: we first introduce the motivation for our work in Section II. The details of the CANA framework shown in Section III which contains the design of domain-specific language(DSL) and heuristic strategies. Section IV presents the experimental results, which evaluates the effectiveness of our work. In Section V, we discuss some related work. Finally, Section VI gives our conclusions and discusses our future research.

II. MOTIVATION

Many program synthesis approaches searches in the program space or generated step-by-step, which usually focus on the code snippet-level or method-level work that only contain API calls. They usually treat the API calls as components in Java library [5] [13] or nodes in the syntax tree [9].

There are two reasons for generating programs which only contains API calls. One is that the API calls usually exist the sequences relationships [8] which are easier to discover and learn. For example, the return type of an API call is frequently used as the parameter of the next API call, which is also the inspiration of many program synthesis based on components and search graphs [3]. Many kinds of researches [9] [8] [10] solve the problem by constructing a probabilistic model, which can mine and learn the sequence of API calls. On the other hand, non-API operations such as infix expressions usually occur between the same type and certain constant values. Besides, some variable assignment expressions of which the right side of the equal sign is usually a constant. Those are difficult to obtain from user intent. For example, for an addition operation between two integers $i1$ and $i2$, the expected value is $i1 + i2$, and the result may become $i1 + i1$.

The following is a java program example which is synthesized by the probability synthesis tool BAYOU. BAYOU takes a program draft as input which consists of a framework of class, a function head, and some labels(contains the API method name of the type name). Although it contains complex control structures and API calls, it is very terrible for disposing of primitive types and non-API operations.

We sampled 10,000 Java files on GitHub which contains 84995 methods. The statistical results showed that each method has an average of 8.23 lines, including 2.51 API calls and 2.41 non-API operations, where non-API operations contain array operations, the primitive types initialization, and the expressions. The results show that every 2 code operations include 1 non-API operation, this result reflects the

```
1  public class SubStringTest {
2   public static String cutString(String file,
3    String suffix, int i1, int i2) {{
4      String s1;
5      boolean b1;
6      int i3;
7      if((b1 = suffix.endsWith(suffix))){
8        i3 = file.length();
9        s1 = file.substring(i2, i2);
10     } else {}
11     return s1;
12 }}}
```

Fig. 1: A synthesized program from a synthesizer tool named BAYOU.

frequency and importance of non-API operations. Actually, program synthesis based on a logical derivation [11] [12] is more useful for handling such operations, but this method requests providing a complete and correct specification, which is as difficult as writing a program yourself in many cases . Although it is difficult to handle, we cannot always ignore these operations in the inductive program synthesis.

We propose a framework named CANA, which solves the problem of program synthesis containing non-API code through two main ideas. We use the probabilistic program synthesis tool BAYOU [9] as an example to elaborate in detail. BAYOU is one of the state-of-the-art program synthesis systems which contains a probability encoder-decoder model that trained on program sketches for generating API-heavy Java code. The sketch is a tree-structured specification that only preserves the control structure, API methods, and types.

III. METHOD

A. Framework

As shown in Fig. 2, this is our framework for synthesizing non-API code, of which two main ideas are marked with rectangular rectangles. One is to encapsulate non-API operations, where we realize this by building a modified DSL. The second is heuristic strategies that address the problem of primitive type initialization and selection when synthesizing Java programs.

We trained the model before applying it to the generation task because that BAYOU contains a neural generator. We grab the Java program from Github and parse it according to the new DSL we build. The obtained program sketches are then used for model training to get the $Model$ in Fig. 2. With this encapsulation and training, the user can also enter non-API operation names to represent the task's requirements compared to before. The model then generates sketches based on user input and we adopt some heuristic strategies to process these sketches. It includes the initialization definition of variables and the selection of variables correction, and finally get the generated java program.

B. Non-API Operations Encapsulation

Here, we expound the encapsulation principles and details of non-API operations. Since most of the program synthesis focuses on synthesizing programs contains the API calls, we

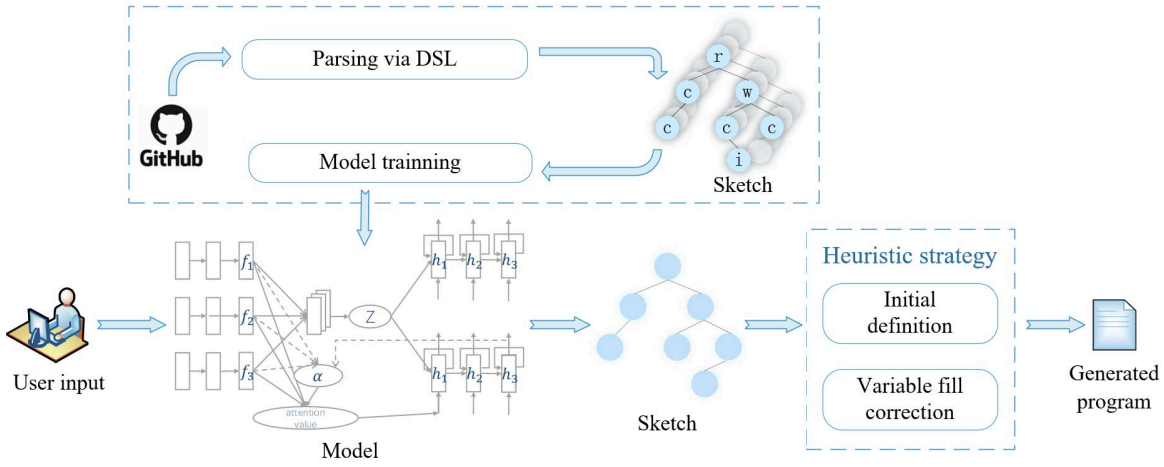

Fig. 2: The framework of CANA.

can use this as a basis to encapsulate non-API operations into the API call. The advantage of this approach is that we can solve the non-API synthesis problem with reusing the current module. For example, a program synthesis based on graph search treats API call as a node in a graph, and we can similarly handle non-API operations as nodes in a graph.

The model in BAYOU training on sketches, where the API call is a node in the sketch. The grammar of sketches is shown in Fig. 3. Here, Y is the sketch of the program, which contains a sequence of API calls, as well as some control structures: branch statements, loop statements, and exception handling statements. $Cexp$ is a subset of Java expressions, representing a limited set of API calls. τ_0 represents the object on which the method is called, where a represents the name of the calling method, $\tau_1, ..., \tau_k$ represent a set of API data types that be used in parameters. $Cseq$ represents a series of API calls. $Catch$ represents a series of exception handling types, $\tau_1, ..., \tau_k$ represent types of exception , $Y_1, ..., Y_k$ represent the specific exception handling process. Note that the sketch does not contain constant or variable names.

$$
\begin{aligned}
Y &::= skip \mid call\ Cexp \mid Y_1; Y_2 \mid \\
&\quad\ if\ Cseq\ then\ Y_1\ else\ Y_2 \mid \\
&\quad\ while\ Cseq\ do\ Y_1 \mid try\ Y_1\ Catch \\
Cexp &::= \tau_0.a(\ \tau_1, ..., \tau_k) \\
Cseq &::= List\ of\ Cexp \\
Catch &::= catch(\ \tau_1)\ Y_1\ \ ...\ catch(\ \tau_k)\ Y_k
\end{aligned}
$$

Fig. 3: The grammar of sketches.

We modify the grammar for sketches by adding non-API operations. The revised portion is shown in Fig.4. We add Cop, which represents a subset of non-API operations. ι represents the identification of all non-API operations, op represents the name of the operation, such as addition and greater operations in infix expressions, $\varepsilon_1, ..., \varepsilon_k$ represent the types involved by the operator, usually are some primitive types. $Cseq$ are composed of expression and operator sequences. Cop is added to Y and encapsulates with the keyword of $call$. It's actually possible to encapsulate $op\ Cop$ instead of $call\ Cop$, but we chose the way in Fig.4 for ease of development and code reuse.

$$
\begin{aligned}
Y &::= skip \mid call\ Cexp \mid call\ Cop \mid \\
&\quad\ Y_1; Y_2 \mid if\ Cseq\ then\ Y_1\ else\ Y_2 \mid \\
&\quad\ while\ Cseq\ do\ Y_1 \mid try\ Y_1\ Catch \\
Cop &::= \iota.op(\varepsilon_1, ..., \varepsilon_k) \\
op &::= + \mid - \mid * \mid / \mid \&\& \mid \mid\mid \mid \\
&\quad\ > \mid >= \mid == \mid != \mid < \mid <= \\
Cseq &::= List\ of\ (Cexp \mid Cop)
\end{aligned}
$$

Fig. 4: Grammar for sketches contains non-API operations.

Despite the addition of some mathematical operations, we continue to maintain the inexistence of constants and variable names in sketches. We move the variable initialization to the code filling section, where we use heuristics to complete the variable filling. We move the processing to the variable filling part where using heuristics to improve it.

C. Heuristic Strategies

Here, we introduce the details of heuristic strategies.

1) Initial Definition: After the sketches generated from the model, they need to be combined into the complete Java programs. In this procedure, some parameters and call objects of calls and operations need to be initialized and allocated. Also, non-API operations often involve processing on the primitive types, and we must resolve the issues of the primitive type declaration and initialization. However, BAYOU only declares variables but not initialize operations. To ensure the correctness of the program syntax, we added complete initialization for the variables.

$$
\begin{aligned}
n &::= -1 \mid 0 \mid 1 \mid 2 \mid 0.0 \mid 1.0 \mid 2.0 \\
b &::= true \mid false \\
s &::= \text{""} \qquad other ::= null
\end{aligned}
$$

Fig. 5: The scope of the variable initialization.

The scope of variable initialization is shown in Fig.5, where n represents the int, $float$ and $double$ types, b represents the $bool$ type, s represents $String$ and $char$ types, and the rest are assigned $null$ because the initialization of other non-primitive types is finished by the constructor in the API call.

2) Variable Fill Correction: we found some errors when we used composite techniques to combine the Java program with sketches. When multiple variables of the same type are presented in the formal parameters of an API call, the generated program usually selects the same variable. However, this is unlikely to happen in a real code environment. For example, the API call $java.lang.String.substring(int, int)$ in Fig.1. The parameters in the $substring$ method are of type int, and it is easy to see the error (filling in the same variable $i2$) similar to Fig.1. Therefore, we modified the variable filling rule and increase three heuristic strategies to improve the correctness of the program, it is also useful for program synthesis without result validation. The heuristic strategies are as follows.

- For adjacent API calls or non-API operations, the most recently assigned variable in the context should be preferentially selected from the variable candidate list. Because this variable is likely to be used in the next call.
- For the same type of parameters in the API call or non-API operation, the variable with the fewer references should be preferentially selected from the variable candidate list. It can avoid the same variable used in the same operation as many times as possible.
- The variable name often contains some semantic knowledge, which usually is some type of information. This information should also be added to the calculation of the similarity distance between the target variable and the candidate list.

The specific workflow is as follows. In the first, we obtain the API call name, type name, and other information of the current target, calculate the edit distance between the foregoing information and the names of candidate variables and select the target variable by the similarity. We then obtain a list of candidate variables, descends sorted according to the times of references.

IV. Experiment

In this section, we assess the effectiveness of our approaches.

A. Experiment Setup and Pre-processing

We construct a new dataset and evaluate the effectiveness of our framework on it. The experiment data grabbed from the over 700 Java projects with the highest star on GitHub. By analyzing and cleaning the data, 335,891 class-level code mainly using $java.io$, $java.lang$, $java.util$ libraries were extracted. We then pre-processed all code, translated the methods in Java files to program sketch. From the extracted programs, we screened above 60,000 sketches and 5,000 sketches to be in the training and testing data.

TABLE I: The descriptions of six metrics.

Metric	Description
1	This metric measures the program abstract syntax tree equivalence rate, which measures whether programs sketch are grammatically equivalent represented 1 or not 0.
2	This metric computes whether the calls are similar in terms of sequences.
3	This metric computes whether the calls are similar in terms of sets.
4	This metric computes the percentage in the number of control structures, as a ratio of the benchmark. The control structures contain branches, loops, and try-catch statements.
5	This metric computes the difference in the number of statements, as a ratio of the number from the benchmark.
6	This metric computes the average time to generate 10 programs.

In order to obtain better performances, we adjusted the neural network. The difference from the original BAYOU [9] is that we add an attention layer, which provides better composite performance than the original model. Our hyper-parameters are saved in a configuration file. Among them, 128, 64, and 64 units are used to three FFNNs in the encoder for calls, types, and keywords, respectively, and 256 units are used for two LSTMs in the encoder. We used a 32-dimensional space to concatenate these two parts, a mini-batch size of 50, a learning rate of 0.0002 for the Adam gradient-descent optimizer [14], and ran the training for 100 epochs in total. The model training took about 6 hours. The experiments are conducted using Oracle HotSpot JVM 1.8.0_101 on an Intel Xeon Server with a 64G E5- 2682v4 CPU with 16G GPU memory, running Ubuntu 16.04.

B. Model Accuray

We use the metrics from Murali et al. [9] to evaluate the models. These metrics measure the equivalence of program abstractions, sketch, which do not contain low-level information, such as variable names. The descriptions of every metric are shown in the Table I.

TABLE II: The metric of the result.

Label	Baseline	CANA			
Oberveability	100%	100%	75%	50%	25%
M1	0.35	0.35	0.35	0.24	0.14
M2	0.65	0.65	0.65	0.75	0.86
M3	0.50	0.43	0.44	0.56	0.72
M4	0.11	0.27	0.27	0.17	0.12
M5	0.23	0.41	0.41	0.32	0.36
M6	0.82	1.04	1.04	0.85	0.62

The experimental results are shown in Table II. These results come from the average of the corresponding metrics calculated by the test set, and we also observe the synthesized results by

using four different input proportions of 100%, 75%, 50%, and 25% as the input as of experiment.

As shown in Table II, the AST equivalence declined as the observable of the label decreased. To eliminate the effects of the dataset, we used the extracted dataset to train the model which only contains API calls and control structures as our baseline. The test set is also 5000 programs, and M1 remains at 0.35 with 100% evidence input. Non-API operations usually involve processing in basic data types, and an operation can often occur in multiple type combinations, which lead to a rapid increase in the non-API operations of the same name. For example, $int + int$ and $String + int$ are different operations, where the former aims to sum integers, and the latter aims to splice string. This is a difficult task for training and learning the model. However, as shown in Table II, the performance difference is marginal between our model and the baseline with a 100% label of inputs. This means that we make non-API operations effectively involved in model training and learning.

Also, to illustrate the effectiveness of our work more intuitively, we selected four examples as shown in Table III. For each example, we gave an input shown in Fig. 6, which consists of a method framework, and some labels that the target program would use. There is an example of a task in Fig. 6, where we want to get all the folder names based on the absolute and relative paths of the file, and the result is returned as $List$. Besides API calls and types, we can also provide names for non-API operations. For example, $call : plus$ in Fig. 6 means we want to be able to do a string concatenation by "+" operation.

```
1  public class Main {
2   public List<String> SplitStringList
3    (String relPath,
4    String absPath,
5    String split)
6    {{
7      ///call:plus call:asList call:split
8    }}}
```

Fig. 6: A task input, which is a draft program.

Two of the generated programs $SplitStringList$ are shown in the second row of Table III. The left side is the result of the baseline synthesis and the right side is the result of CANA synthesis. They are all from the top 10 rankings. This result in Table III shows that, first of all, we have a reasonable initialization of the variables compare with baseline. Our results performed well in the same type of fill, and we don't have a similar problem in line 7 of Table III when we do a string concatenation, and we synthesized programs contains non-API operations. Totally, we synthesized the programs that took non-API operations, syntax correctly, and met the input specification.

V. RELATED WORK

The problem of program synthesis has long been considered the holy grail of Computer Science [5]. In this section, we discuss the related work in program synthesis for non-API operations and program synthesis base on DSL.

A. Program Synthesis for Non-API Operations

Program synthesis takes into account both high-level abstractions and low-level details, so synthesizing code that includes basic operations and basic types is a difficult task. Since the code is a hierarchical structure, some of approaches [15] [16] synthesize programs on the tree structure. These trees are called abstract syntax trees and are constructed explicitly or implicitly by the compiler after the lexical analysis of valid code sequences. The models from Maddison and Tarlow [15] are based on probabilistic context free grammars (PCFGs) and neuro-probabilistic language models, which are extended to incorporate additional source code-specific structure. Rabinovich et al. [16] build abstract syntax networks, in which outputs are represented as abstract syntax trees and constructed by a decoder with a dynamically-determined modular structure paralleling the structure of the output tree. Also, some works to synthesize non-API operations by randomly generating code snippets. FrAngel [19] is a component-based program synthesis tool that starts by randomly generating code, then constraining and standardizing the resulting program through input and output use cases. Another approach solves low-level syntax and detail problems by building syntax models. Yin and Neubig [20] propose a novel neural architecture powered by a grammar model to explicitly capture the target syntax as prior knowledge.

B. Program Synthesis Base on DSL

Many synthesis approaches are based on top-down enumeration searches as a way to build the required programs. A common theme is to scale up the search space that a program synthesis and one of the key ideas is syntactically restricting the space of possible programs [5]. This syntactic bias can be expressed by various means such as a domain-specific language (DSL).

Although DSLs are more restrictive than full-featured programming languages like Java, it can enable more efficient special-purpose search algorithms [21]. Programming by Example (PBE) usually designs a DSL to constraint the program space and combines various techniques to accelerate the search process [22]. One of the most famous tools is FlashFill [23]. Singh and Gulwani [23] study the problem of predicting a correct program from a huge set of programs in an expressive DSL that has been induced by a small number of examples. Among them, the DSL for syntactic string transformations contains programs that take an n-ary tuple of strings as input and return a string as output. The Neural FlashFill [24] system contains the Recursive-Reverse-Recursive Neural Network (R3NN) that incrementally synthesizes a program in the DSL given the continuous representation of the examples. The recent work of Balog et al. [21] and Parisotto et al. [24] combine ideas from existing enumerative search techniques with learned heuristics to learn to efficiently synthesize code, usually written within a DSL.

TABLE III: The comparisons of four examples between baseline and CANA. Those are selected from the top-10 programs that are closest to the answer.

Baseline	CANA

public class SplitStringList {
  public List<String> transfer(String relPath,
  String absolPath, String split) {{
  String[] s1;
  s1 = absolPath.split(split);
  return null;
}}}
	public class SplitStringList{
	  public List<String> transfer(String relPath,
	  String absolPath, String split) {{
	  String[] s2 = null;
	  String s1 = "";
	  List<String>l1= null;
	  s1 = relaPath + absolPath;
	  s2 = s1.split(split);
	  l1 = Arrays.asList(s2);
	  return l1;
	}}}

public class GetListLastItem {
  public String getListLastItem(List<String> news,
  int last, List<Iterable<String>> _l1,
  List<Float> _l2) {{
  String arg01;
  boolean b1;
  String s1;
  boolean b2;
  int i1;
  boolean b3;
  b1 = news.add((arg01 = String.valueOf(last)));
  b2 = _l1.add(news);
  s1 = news.get(last);
  b3 = news.add(s1);
  i1 = _l2.size();
  return arg01;
}}}
	public class GetListLastItem {
	  public String getListLastItem(List<String> news,
	  int last) {{
	  int i2 = 0;
	  int i1 = 0;
	  String s1 = null;
	  i1 = news.size();
	  i2 = i1 - last;
	  s1 = news.get(i2);
	  return s1;
	}}}

public class Encryption {
  public int encryption(int key, List<String> _l1) {{
  int i1;
  i1 = _l1.size();
  return key;
}}}
	public class Encryption {
	  public int encryption(int key){{
	  int i2 = 0;
	  InputStream arg01 = null;
	  int i1 = 1;
	  Scanner s1 = null;
	  s1 = new Scanner((arg01 = System.in));
	  i1 = s1.nextInt();
	  i2 = i1 + key;
	  return i2;
	}}}

Cannot be synthesized because it involves array operations.	public class OutputStreamWrite{
	  public boolean outputStreamWrite(OutputStream os,
	  byte[] buffer, int max, int len) {{
	  boolean b1 = true;
	  try {
	  os.write(buffer);
	  os.flush();
	  if ((b1 = len <= max)) {
	  os.close();
	  }else {}
	  }catch (IOException _e) {}
	  return b1;
	}}}

VI. Conclusion

The current approach to program synthesis focuses more on code-segment and method-level work that includes only API methods. In this paper, we study the problem of non-API operation in inductive program synthesis and analyze its two difficulties. We propose a framework called CANA to solve these two problems, which contains two main ideas. One is to encapsulate non-API operations into API calls so that they can participate in model training, then users also can provide information when describing incomplete specifications. The other is that we provide some heuristic strategies to solve the problem of selecting the same type of variables. We take a program synthesis tool named BAYOU. Experiments show that we have effectively added non-API operations and are able to perform the synthesis task reasonably well, while maintaining the performance of the original model. Our work so far has only obtained a preliminary result, and we will continue to study in-depth in the future to analyze the similarity and difference between API calls and non-API operations, so as to better solve the problem of program synthesis.

References

[1] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, 2010, pp. 13–24. [Online]. Available: https://doi.org/10.1145/1836089.1836091

[2] S. Gulwani and M. Marron, "Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 803–814. [Online]. Available: https://doi.org/10.1145/2588555.2612177

[3] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017, pp. 599–612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009851

[4] A. Solar-Lezama, "The sketching approach to program synthesis," 12 2009, pp. 4–13.

[5] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: https://doi.org/10.1561/2500000010

[6] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6679385/

[7] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, 2018. [Online]. Available: https://doi.org/10.1145/3212695

[8] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 419–428. [Online]. Available: https://doi.org/10.1145/2594291.2594321

[9] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. [Online]. Available: https://openreview.net/forum?id=HkfXMz-Ab

[10] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing developer assistant: improving developer productivity by recommending sample code," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 956–961. [Online]. Available: https://doi.org/10.1145/2950290.2983955

[11] C. C. Green, "Application of theorem proving to problem solving," in *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, 1969, pp. 219–240. [Online]. Available: http://ijcai.org/Proceedings/69/Papers/023.pdf

[12] R. J. Waldinger and R. C. T. Lee, "PROW: A step toward automatic program writing," in *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, 1969, pp. 241–252. [Online]. Available: http://ijcai.org/Proceedings/69/Papers/024.pdf

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 215–224.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[15] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, 2014, pp. 649–657. [Online]. Available: http://proceedings.mlr.press/v32/maddison14.html

[16] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1139–1149. [Online]. Available: https://doi.org/10.18653/v1/P17-1105

[17] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Inf. Softw. Technol.*, vol. 74, pp. 204–218, 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.01.004

[18] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, "Determinants of pull-based development in the context of continuous integration," *Sci. China Inf. Sci.*, vol. 59, no. 8, pp. 080 104:1–080 104:14, 2016. [Online]. Available: https://doi.org/10.1007/s11432-016-5595-8

[19] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 73:1–73:29, 2019. [Online]. Available: https://doi.org/10.1145/3290386

[20] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 440–450. [Online]. Available: https://doi.org/10.18653/v1/P17-1041

[21] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. [Online]. Available: https://openreview.net/forum?id=ByldLrqlx

[22] S. Gulwani and P. Jain, "Programming by examples: PL meets ML," in *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, 2017, pp. 3–20. [Online]. Available: https://doi.org/10.1007/978-3-319-71237-6_1

[23] R. Singh and S. Gulwani, "Predicting a correct program in programming by example," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, 2015, pp. 398–414. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_23

[24] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. [Online]. Available: https://openreview.net/forum?id=rJ0JwFcex