Boosting Component-based Synthesis with API Usage Knowledge

Jiaxin Liu
College of Computer Science, National University of Defense Technology
Changsha, China
liujiaxin18@nudt.edu.cn

Wei Dong
Key Lab. of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology
Changsha, China
wdong@nudt.edu.cn

Binbin Liu
College of Computer Science, National University of Defense Technology
Changsha, China
liubinbin09@nudt.edu.cn

ABSTRACT

Component-based synthesis is one of the hottest research areas in automated software engineering. It aims to generate programs from a collection of components like Java library. However, the program space constituted by all the components in the library is fairly large, which leads to a vast number of candidate programs generated for a long time. The intractability of the program space affects the synthesis efficiency of the program and the size of the program generated. In this paper, we propose ITAS, a framework of iterative program synthesis via API usage knowledge from the Internet, which can significantly improve the efficiency of program synthesis. ITAS aims to constrain the program space by combining two main ideas. First, narrow down the program space from the outside via the guidance of API usage knowledge. Second, expand the program space from the inside via iterative strategy based on knowledge. For evaluation, we collect a set of programming tasks and compare our approach with a program synthesis tool on synthesizing these tasks. The experiment results show that ITAS can significantly improve the efficiency of program synthesis, which can reduce the synthesis time by 97.1% than the original synthesizer.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

KEYWORDS

Knowledge Search, Iterative Strategy, Component-based Synthesis

ACM Reference Format:
Jiaxin Liu, Wei Dong, and Binbin Liu. 2020. Boosting Component-based Synthesis with API Usage Knowledge. In *35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3417113.3423370

Corresponding author: Wei Dong. This work was supported by National Natural Science Foundation of China (No.61690203), and National key R&D program of China (No. 2018YFB0204301).

1 INTRODUCTION

With the burgeoning of reusable application programming interfaces and components (APIs), programmers often utilize APIs in software libraries to make programming tasks easier to solve. However, the process is boring and time-consuming, especially when the programmer is not familiar with the library [13]. To simplify the programming process, component-based synthesis provides a way to automatically assemble the required components to implement a task. It aims at generating a program using a library of components, where each component is a domain-specific function that could be used in the desired program [9]. In this way, it can free the programmers' hands to engage in challenging algorithm or architecture work.

However, there is one of the inherent challenges in program synthesis, i.e. intractability of the program space. It leads to two main shortcomings in general component-based synthesis. The first one is a long time in finding a solution. As there are thousands of APIs in a Java library, the program search space constructed by it is fairly large. Hence, it is hard to find the solution in a limited time. The second one is the small scale of the generated program. As the search space increases exponentially with the program size, only small-scale programs can be synthesized in practice.

In recent years, with the development of hugely successful, widely used, open-source software systems and platforms, there are already billions of code and millions of metadata instances on the Internet [1]. These big codes contain massive knowledge that can be learned to guide programming. As shown in the previous study, to solve the unknown knowledge in development, programmers spend an average of 19% of their programming time on the Internet [4]. Hence, the knowledge searched from the Internet is an important auxiliary role for programming. Analogous to the behavior of programmers, we can also ask the synthesis tool to retrieve API usage knowledge from the Internet to assist program synthesis. Through the guidance of knowledge, we can narrow down the scope of the components required by synthesis to constrain the program space.

In this paper, aiming at the intractability of program space, we propose a framework of iterative program synthesis based on the API usage knowledge from the Internet. There are two main ideas in our work, i.e. shrink from the outside and expand from the inside, based on API usage knowledge. First, narrow down the program space from the outside via the guidance of knowledge. For this aspect, we extract the APIs usage knowledge from the Internet. By the usage of extracted APIs instead of all the APIs in the library as components, it can reduce the number of APIs while ensuring the existence of necessary APIs. And the program space is narrowed as the number of components decreases. To achieve this, we have implemented a tool named ARGS (API Recommendation via General Search). Second, expand the program space from the inside via iterative strategy. For this aspect, we incrementally construct the program search space, which starts from a small-scale space and iteratively add new components to it for approaching the space required by the target program. To achieve this, we propose an algorithm of iterative program synthesis. Combining the two ideas, we propose the framework to improve the effiency of program synthesis and name it ITAS (Iterative program synthesis with ARGS). When given a natural language description of a task, it can constrain the program space to improve the efficiency of program synthesis and solve some complex programs.

To evaluate the effectiveness of our framework, we select a program synthesis tool named SYPET [6] as an example. When the user provides some specifications, including a collection of libraries, the function signature, and some test cases, SYPET could synthesize the desired program in limited time. For comparison, we collect a set of programming tasks and manually write natural language descriptions for them. Then, we combine ITAS with SYPET and compare the combined version with the original SYPET by synthesizing these programming tasks. Our experimental results show that our framework is a great help to improve the efficiency of component-based synthesis. It could solve more tasks than the original SYPET and has a 97.1% improvement in synthesis time.

To summarize, this paper makes the following contributions:

- We propose a general program synthesis framework ITAS based on API usage knowledge and iterative strategy. When given a natural language query of a task, it could narrow down the scale of the program search space.
- We present an algorithm of iteratively constructing program space based on the knowledge searched from the Internet. It can constrain the scale of program space and optimize the process of finding a solution.
- We perform an evaluation of ITAS by combining it with a state-of-the-art synthesis tool SYPET and compare the synthesis results of them. Our experiment shows that ITAS can significantly improve the efficiency of program synthesis.

The rest of this paper is organized as follows: we first introduce the motivation of our work in Section 2. The detail of the framework ITAS is shown in Section 3. Section 4 presents the experimental results, which evaluates the effectiveness of our work. In Section 5, we list some related work. Finally, we conclude this paper in Section 6.

2 MOTIVATION

Component-based synthesis often builds a program search space based on the component library provided by the user. But there are thousands of components in the library, which will bring a hard combination problem. Although some advanced constraint solving and space enumeration techniques can explore a larger program space in a reasonable time, it is still not enough for the program synthesis in a generic programming language like Java.

In this paper, we take a state-of-the-art synthesis tool SYPET as an example to illustuate the problem of program space. It takes as input a function signature that defines the input and output types of the desired program, one or more test cases, and a library of API components with which a program search space (Petri net) is constructed. When given these inputs for a task, it can synthesize a straight-line Java program. Two main phases compose the synthesis process, i.e. sketch generation and sketch completion. In the phase of the sketch generation, SYPET performs a reachability analysis on the Petri net to enumerate reachable paths from the input types to the output type. The sequence of transitions (APIs) in a reachable path corresponds to a program sketch. In the phase of sketch completion, the program sketch is completed into a candidate program by type-check arguments and variables. Finally, it verifies the correctness of candidate programs according to the test cases provided by the user.

Considering the task of “Generate an XML from file and query it using XPath”, it cannot be solved in 10 minutes by SYPET. We analyze why the task is failed from two aspects of space and time. In the aspect of space, SYPET utilizes 4276 APIs of the library to construct the space in which contains 10635 edges. However, the task only needs permutations and combinations of 6 APIs in deed. Hence, there are many unnecessary components in the space, which may render a vast number of useless traversal. Although there are solutions in the space, it is hard to find it. In the aspect of time, we count the time (in seconds) spent in each phase of program synthesis to determine where most time of program synthesis is cost. As shown in Table 1, there are 5 phases in program synthesis, where “Soot time” represents the time of constructing space. We can see that most time is spent on the phase of sketch generation to search the candidate API sequence, which takes up 98.7% of the total time. Therefore, even if the search techniques are advanced enough currently, the efficiency of searching for candidate programs is still a bottleneck in program synthesis.

Table 1: An example of time spent of each phase in synthesis for a programming task.

Phase	Soot Time	Sketch Generation	Sketch Completion	Compil-ation	Running Test Cases
Time(s)	4.59	592.01	0.2	3.03	0.17

Inspired by it, we focus on how to narrow down the program space to improve the effectiveness of program synthesis. There are two insights into this work. First, we can improve the efficiency of synthesis. If we can reduce the program space, there is no doubt that the synthesis time will be decreased. Second, we may solve more tasks or the larger size of the program. As the scale of the space becomes fairly small, the synthesizer can traverse the space faster before timeout. We propose a framework named ITAS, which can improve the efficiency of program synthesis and solve more tasks. The details of our framework will be introduced in Section 3.

3 APPROACH

In this section, we illustrate our approach in detail. Firstly, we give an overview of how to combine our approach with the program

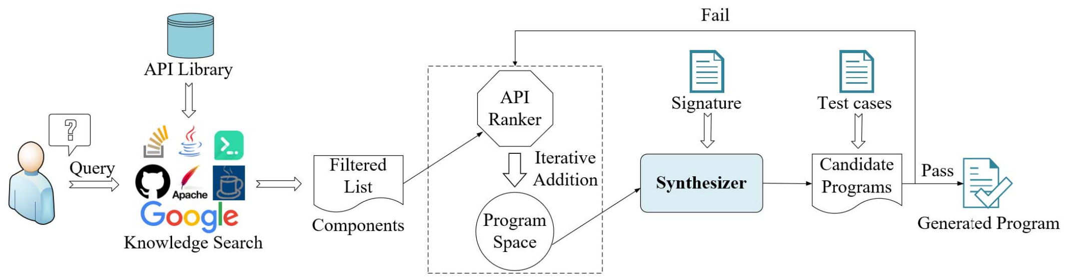

Figure 1: The framework of iterative program synthesis with API usage knowledge from the Internet.

synthesizer. Then we detailedly introduce our two ideas of how to constrain the program space for the improvement of synthesis efficiency.

3.1 Overview

As shown in Figure 1, we give a framework of how to combine the API usage knowledge from the Internet and iterative strategy with the program synthesizer. There are two main parts of the framework. The first part is the acquirement of valuable knowledge from the Internet, where the knowledge refers to the usage information of task-related APIs. As the knowledge on the Internet plays a great role in development, inspired by this, we implement a tool based on the search engine to assist synthesis and name it Args. It achieves the function of collecting the necessary API usage knowledge from the Internet like programmers. Given the query of a programming task, Args takes it as input and extracts a list of APIs from the Internet. After filtering according to the API library provided by the user, we get a list of APIs belonging to the specified library, which can be used as the components of program synthesis. The second is the iterative construction process of program space. For the already obtained API list, we first preprocess them and construct the API rankers. Then we initially select a few APIs from the list based on the rankers as an initial set and use the set to build a basic program search space. If the synthesis tool can find a candidate program in the current program space that passes all test cases, it returns the generated program. Otherwise, we expand the space by adding new components according to the API ranker once per round, until the program synthesizer finds the correct program or runs timeout.

3.2 Knowledge Search

As mentioned above, there are thousands of APIs in a library provided by the user. However, the desired program only uses a little part of them, which a lot of them are useless. Although it is hard to find the necessary APIs accurately from the library, we could use knowledge to exclude the useless APIs as much as possible to narrow down the program space. With the development of the technique of the Internet and so much code available, we can easily obtain the code knowledge from the open-source repositories like Github, online forums like StackOverflow, textual tutorials like Javadoc and other programming examples sites like Codota. To get the knowledge from the multi-sources, we propose retrieving it via the general search engine.

Therefore, we propose to reduce the scale of the component library with the Google search engine, by which achieving the pruning of space from outside. Analogous to the process that human programmers engage in obtaining programming knowledge from the Internet, we previously implemented a tool named Args to obtain task-related APIs from the Internet for the synthesizer. There are four main steps in Args to get the API list: retrieving relevant webpages, parsing code snippets, extracting APIs, and filtering them. Specifically, when given a natural language query, Args first retrieves massive relevant webpages from the Internet. Then it parses the code snippets embedded in it. After obtaining these snippets, it constructs the AST for them and extracts the APIs. Finally, it filters these APIs according to the library provided by the user and then returns a list of task-related APIs. In this part, the scale of the component library is reduced by excluding a lot of unrelated components. Then, the program space built based on the component library is also narrowed down accordingly.

With the help of knowledge searched from the Internet, we achieve a reduction in the number of components. Meanwhile, it also ensures that the API required for the synthesis of the target program still exists within the scope of the current component. However, due to the inaccuracy of information on the Internet, we only reduce the number of components from thousands to about one hundred. Therefore, there are still many useless APIs for the target program synthesis, which will lead to a lot of useless programs generated before finding the real solution. As reducing the scale of the program space from the outside has reached the upper limit, we propose to further narrow it from the inside.

3.3 Iterative Construction

In order to reduce the synthesis of useless programs before finding a solution, we hope that the program space only contains the API required by the task as much as possible. To achieve this, we propose to iteratively expand the program space from the inside. It aims to start with a small-scale space construction that is as task-relevant as possible, and gradually expand the space to approach the real space where the target program is located.

As shown in Algorithm 1, it describes the procedure of our iterative strategy. There are mainly three steps in it, the construction of API rankers, the initialization of program space, and the iterative process of program synthesis. Firstly, we preprocess the API list returned by Args, represented by L, and sort these APIs. We precalculate the IDF (Inverse Document Frequency) and DF (Document Frequency) value of each API by counting the frequency of it in L,

Algorithm 1 Procedure of iterative program synthesis.

```
Input: API list returned by ARGS $L$, natural language query $Q$,
    dictionary of pre-trained word and API vectors $D$
Output: A program $P$ that passes all the test cases
1:  procedure ITERATION($L, Q, D$)
2:      $S \leftarrow \emptyset$
3:      $I, R, C \leftarrow$ APIRANK($L, Q, D$)
4:      $S \leftarrow$ INITSPACE($I, R, C, L$)
5:      repeat
6:          $P \leftarrow$ GENPROGRAM($S$)
7:          if $P$ passes all the test cases then
8:              break
9:          end if
10:         $S \leftarrow$ APIADDITION($I, R, C, L$)
11:     until timeout
12:     return $P$
13: end procedure
```

where each API represents a document. To select the more task-related APIs for the construction of program space, we need to sort them according to the relevance. However, it is difficult to identify all the needed API according to the ranking one time, as there are some edges APIs in a program, which are used as completing the program instead of achieving the function. As shown in the following example, to calculate the difference between two dates, we only need to call the core API `Days.daysBetween`. However, to satisfy the user-defined input and output types in the signature, we need to add some edge APIs like `LocalDate.<init>` and `Days.getDays` for it to complete the entire program.

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

From the semantics, we can see that compared with the core API, the edge APIs have no obvious task relevance. Hence, if all APIs are sorted together without distinction, it is difficult to obtain the necessary edge APIs in the top few. In this case, to cover all APIs required by the target program, we need to set the rank of the last required API as the threshold for selecting the number of APIs. The threshold will be a large value in most cases. Heuristically, we decompose a program into 3 categories of APIs, i.e. constructor APIs, core APIs, and "return" APIs. Then we construct the 3 types of API rankers, respectively. The first ranker I is for constructor APIs that initializes the instance object. The constructor guarantees the existence of new objects. The second ranker R is for those APIs whose return type is consistent with the output type defined in the signature provided by the user. This type of APIs ensures that a program consistent with the output type provided by the user can be synthesized. Both categories of APIs are sorted by the DF value. The third C is used to select the core APIs that are crucial to achieve a certain function. We utilize the techniques in natural language processing (NLP) to rank them by calculating the similarity between the natural language description of the task query and the APIs. For achieving this, we introduce an API recommendation tool WORD2API [11]. It generated 126,853 word and API vectors and published them as a dictionary. We use the IDF value obtained by the preprocessing and calculate the semantic similarity according to the formula defined in [11], as shown in the following:

$$sim(W, A) = \frac{1}{2}\left(\frac{\sum(sim_{max}(w, A) \times idf(w))}{\sum idf(w)} + \frac{\sum(sim_{max}(a, W) \times idf(a))}{\sum idf(a)}\right)$$

In this formula, W and A respectively represent the set of words in the query and the set of APIs. In practice, we treat the A as one API to calculate the similarity between a query and an API. And sim_{max} represents the maximum cosine similarity between a set of words and an API. Then, we sort these APIs and regard top-ranked as the core APIs.

Secondly, we initialize the set S according to the 3 rankers and use the set to build a small-scale program space. For knowing how many APIs are needed for a program to guide the construction of space, we conducted statistics for the proportion of API calls on the programs in Java projects from Github.

Table 2: A statistical result of the proportion of API calls in the program.

Total Project	Total Program	Program Size	API Calls
722	167450	12.61	6.76

As shown in Table 2, the results show that a program is averagely 12 lines of code, of which 6.76 lines are API method calls. Based on this result, we determine the number of each category APIs in the initialization process. For the constructor API, since there are at least one "return" API and one core API in the program, its initial range should be between 0 and 5. We choose a maximum value of 5 to cover more constructors that may be needed. For the "return" API, as there is only one return value for a Java program, we only select the top-1 initially. For the core API, in particular, there are often some necessary APIs that tend to occur together with it. For example, `<DocumentBuilderFactory.newInstance, DocumentBuilderFactory.newDocumentBuilder, DocumentBuilder.parse>` is a common API sequence in a program. Hence, we also consider adding the context of the core APIs into space together with it. Based on the data in Table 2 and our trials, we finally set the context window as 4 to cover more potential APIs related to core APIs. Finally, we select the top-5 APIs in I, top-1 of R, top-1 of C, and its context methods in L, to construct the initial space.

At last, we introduce the process of iterative program synthesis. We conduct the first round of program search in the initial space constructed before. If the solution is found in the current space, the process ends and returns the generated program. Otherwise, we add the next API in I, the next API in R, and the next API in C together with its context APIs to the set S for the next iteration. The process finally ends with a timeout or finding the solution.

4 EXPERIMENT

In this section, we evaluate the effectiveness of our framework to study how much they can improve the efficiency of synthesis.

4.1 Experiment Setup

4.1.1 Benchmarks. In order to evaluate the performance of our framework on component-based synthesis, we collect 30 programming tasks where 14 of them are provided by SyPet. The other 16 tasks are collected from open-source repositories like GitHub and online forums like StackOverflow. Since Word2API only generates vectors for the Java Standard Edition (SE) APIs, our framework cannot be used for tasks that depend on the third-party library. And we manually write a natural language description for each task which can be used as a query for the input of our framework.

4.1.2 Metrics. As there are no recognized criteria in program synthesis to judge the effect of synthesis, we define two evaluation criteria. First, we count the number of programming tasks solved correctly. In our experiment, we set a timeout to 10 minutes. As our multiple trials show, if a task cannot be done within 10 minutes by SyPet, no solution can be found for a longer time. Thus, if a task cannot be solved within 10 minutes, we record this task as "cannot be solved". Second, we count the time spent to find the solution in the program space. If the task is "cannot be solved", we record the spent time as 10 minutes. All the experiments are conducted using Oracle HotSpot JVM 1.8.0_101 on an Intel Xeon Server with an E5-2682v4 CPU and 64G of RAM, running Ubuntu 16.04.

4.2 External Analysis

To evaluate how much improvement our approach Itas can bring to synthesis, we compare the synthesis results of the original open-source version of SyPet [1] and the version combined with Itas.

Table 3: Comparison of synthesis time on the 30 programming tasks between SyPet and Itas.

ID	SyPet	Itas	ID	SyPet	Itas	ID	SyPet	Itas
1	2.43	0.79	11	0.41	0.24	21	0.49	0.23
2	2.86	2.02	12	T/O	0.65	22	12.57	1.73
3	0.87	6.15	13	0.94	0.92	23	3.53	0.43
4	3.99	5.41	14	1.42	1.41	24	0.6	1.46
5	2.31	0.34	15	0.51	0.36	25	0.41	0.57
6	0.46	8.46	16	1.54	0.28	26	35.54	0.39
7	7.46	0.7	17	55.63	0.27	27	T/O	1.48
8	8.71	1.36	18	1.01	0.27	28	8.64	2.58
9	4.18	0.64	19	177.82	0.34	29	T/O	6.99
10	3.43	0.46	20	9.75	5.59	30	T/O	29.07

The synthesis result is shown in Table 3, each item of the column denotes the synthesis time (calculated in seconds) for each programming task and "T/O" represents the task cannot be solved. If a tool reports timeout for a task, 600 seconds will be recorded as the time cost of this task when calculating the average synthesis time. From the table, we can calculate that SyPet takes about 91.6 seconds and Itas takes 2.7 seconds for a task on average. The synthesis time is reduced by 97.1% with Itas. To further illustrate the effectiveness of our method, we also only compare the tasks that both can synthesize in time. The result shows that they respectively take 13.4 seconds and 1.7 seconds. It indicates that Itas can significantly improve the efficiency of program synthesis. In terms of the number of tasks, we can see that SyPet can solve 26 tasks and Itas can solve all of them. We manually inspect these tasks and make two points of analysis on why SyPet could not solve them. First, the size of the program. The program size of the two unresolved tasks is a little larger for SyPet, which renders a large program space needed to traverse. Second, the number of API combinations. For a task that with the same input and output types defined by the user, this synthesis process often falls into an infinite loop of permutations and combinations. This is the case with the other two tasks. The results show that Itas can solve the tasks that are difficult for SyPet, which indicates that the reduction of program space is effective for the synthesis of more complex tasks. Overall, Itas can significantly improve the efficiency of component-based synthesis and help to solve more tasks.

[1]https://github.com/utopia-group/SyPet

4.3 Internal Analysis

In order to evaluate the effectiveness of the two main parts of our approach, i.e. the API usage knowledge from Internet and iterative synthesis strategy. We compared the original SyPet with the version that combined the API knowledge from the Internet (Args) and the iterative strategies based on the knowledge (Itas).

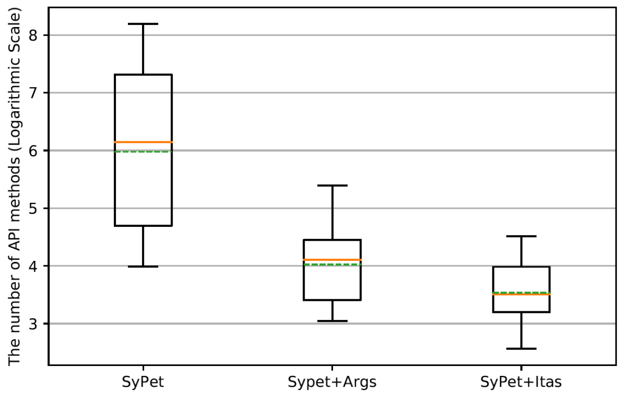

Figure 2: Comparison of the number of API components on 30 programming tasks among SyPet, Args and Itas.

Firstly, we compare the number of components that are used to build the program space by each task. As shown in Figure 2, we use a boxplot to describe the distribution of the number of components. Note that the logarithmic conversion of the y-axis is based on *e*. In comparison to SyPet, the number of components for program synthesis is significantly reduced through the auxiliary effect of API usage knowledge from the Internet. Moreover, with the addition of the iteration strategy, the number of components is further reduced. On average, the number of components has been reduced from the original use of 781 to 72 and eventually reduced to 37.

In response to such a result, we make the following two points of analysis. First, loose relations among the components in the API library. Although an API library is often used for the task of a certain domain, for a specific task, only a few APIs in it are often used together. It makes other APIs in the library relatively useless. Since there is low cohesion among APIs, this provides a large number of APIs that can be excluded with the guidance of knowledge. Second, the inaccuracy of the knowledge searched from the Internet. Considering the search engine is mainly for full-text search, it does not consider the characteristics and semantics of the code. Hence, we can analyze the semantic information between natural language and APIs for fine-grained processing. By iterative strategy, it can obtain the task-related APIs by category according to semantics and continue to obtain other necessary APIs based on the acquisition of some required APIs. This avoids the selection of a large number of APIs and guarantees the rapid acquisition of all the required APIs. Overall, we achieve a reduction in the scale of components globally with the help of knowledge from the Internet, and we further refine the scale locally by iterative strategy.

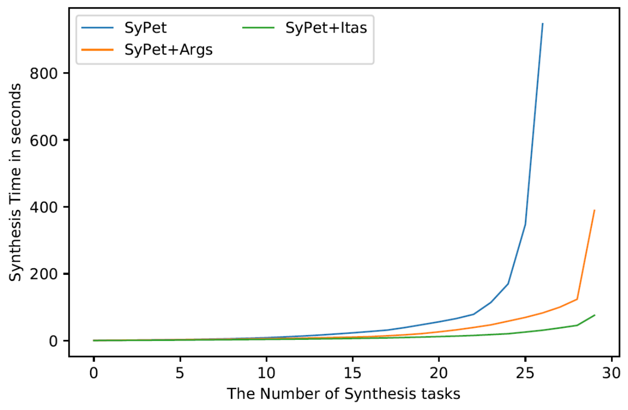

Figure 3: Comparison of the synthesis efficiency of for SyPet, Args and Itas.

Secondly, we compare the synthesis efficiency of them. As shown in Figure 3, the curve in the graph represents the total time cost with the number of tasks solved. First, we can see that SyPet cannot solve all the tasks, instead, both Args and Itas can solve them. In addition, Itas (2.7 seconds) takes the least time to solve the task, followed by Args (13.0 seconds), and finally SyPet (91.6 seconds) on average. This result is consistent with the reduction in the number of components in Figure 2. It can be explained that narrowing down the scale of the program space is effective for improving the synthesis efficiency. As the waiting time that people can accept is usually only 1 to 3 seconds, with the role of iterative strategy, we significantly reduce the synthesis time to a human acceptable range. Overall, the knowledge from the Internet and the iterative strategy plays a great role in the improvement of synthesis efficiency.

5 RELATED WORK

Program synthesis is one of the key research areas in software engineering [15]. At present, there are many excellent methods in the field of program synthesis. From the perspective of search strategy, we divide them into four categories: syntax-guided synthesis, deep learning-based, code search-based, and component-based synthesis.

Syntax-guided synthesis aims to find a program satisfying semantic specification as well as user-provided structural hypothesis [10]. Many syntax-guided synthesis algorithms have been proposed [19] [17] [8]. Besides, Huang et al. [10] proposed a framework that combines concrete search and symbolic search to enumerate the program code from short to long according to the height of the decision tree. Rajeev Alur et al. [2] compared 6 solvers over 1500 benchmarks in the syntax-guided program synthesis competition.

Deep learning-based synthesis often uses a sequence generation criterion, in which sequence-to-sequence models are trained to maximize the likelihood of known reference programs [12]. DeepCoder [3] is a neural network automatic programming system. It uses programs in the field of competition as data sets, which can guide and improve the synthesis process of inductive programs. Ling et al. [12] present a neural network architecture that generates an output sequence conditioned on an arbitrary number of input functions. Yin and Neubig [21] propose a neural architecture powered by a grammar model to explicitly capture the target syntax.

Code search-based program synthesis combines the classic code search process. In the early stage of program synthesis, the code search is used to obtain the code snippets that are closest to the needs, and then further to synthesize program. SWIM [16] can retrieve the relevant API names according to the requirements provided by users. Then it retrieves the possible API sequence pattern from the mined pattern data set and finally packs it into a code fragment. Hunter [20] establishes constraints on the problem of program synthesis through integer linear programming and transforms it into API-based code generation problems.

Component-based synthesis generates a loop-free program from a library of components. Each component is a domain-specific function that could be used in the desired program [9]. It has been widely used in domain-specific applications, such as table transformation [5], data structure transformation [7], and string manipulation [14]. There are also some work in this field which can generate Java programs such as SyPet [6] and FrAngel [18].

6 CONCLUSION

In this paper, we propose a general framework named Itas to improve the efficiency of program synthesis. When given a natural language description of the task, it can speed up the program search process. In this framework, we realize the fusion of massive knowledge on the Internet with traditional program synthesis techniques, and implemented a tool Args. Besides, we propose an iterative strategy by which can further accelerate the solution efficiency through the iterative construction of program space.

We evaluate our approach on 30 programming tasks. The experimental results show that the iterative program synthesis based on the API usage knowledge from the Internet is a great help to improve the efficiency of component-based program synthesis. Through further comparative analysis within the approach, we illustrate that each part of it is complementary to each other.

Inspired by the conclusion drawn from this paper, future work will orient towards the expansion of the program. Besides, due to the domain limitation of Word2API, a new method for measuring the similarity between API and natural language is needed to support more tasks.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput.*

Surv. 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017 (EPTCS, Vol. 260)*, Dana Fisman and Swen Jacobs (Eds.). 97–115. https://doi.org/10.4204/EPTCS.260.9

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

[4] Joel Brandt, Philip Guo, Joel Lewenstein, Mira Dontcheva, and Scott Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. 1589–1598. https://doi.org/10.1145/1518701.1518944

[5] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 422–436. https://doi.org/10.1145/3062341.3062351

[6] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-based synthesis for complex APIs. 599–612. https://doi.org/10.1145/3009837.3009851

[7] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

[8] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[9] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[10] Kangjing Huang, Xiaokang Qiu, Qi Tian, and Yanjun Wang. 2018. Reconciling Enumerative and Symbolic Search in Syntax-Guided Synthesis. *CoRR* abs/1802.04428 (2018). arXiv:1802.04428 http://arxiv.org/abs/1802.04428

[11] Xiaochen Li, He Jiang, Yasutaka Kamei, and Xin Chen. 2018. Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. *CoRR* abs/1810.09723 (2018). arXiv:1810.09723 http://arxiv.org/abs/1810.09723

[12] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1057

[13] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. 2005. Jungloid Mining: Helping To Navigate the API Jungle. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 40, 48–61. https://doi.org/10.1145/1065010.1065018

[14] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 408–418. https://doi.org/10.1145/2594291.2594297

[15] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 179–190. https://doi.org/10.1145/75277.75293

[16] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 357–367. https://doi.org/10.1145/2884781.2884808

[17] Shambwaditya Saha, Pranav Garg, and P. Madhusudan. 2015. Alchemist: Learning Guarded Affine Functions. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 440–446. https://doi.org/10.1007/978-3-319-21690-4_26

[18] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *PACMPL* 3, POPL (2019), 73:1–73:29. https://doi.org/10.1145/3290386

[19] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 287–296. https://doi.org/10.1145/2491956.2462174

[20] Yuepeng Wang, Yu Feng, Ruben Martins, Arati Kaushik, Isil Dillig, and Steven P. Reiss. 2016. Hunter: next-generation code reuse for Java. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 1028–1032. https://doi.org/10.1145/2950290.2983934

[21] Pengcheng Yin and Graham Neubig. 2017. 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*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 440–450. https://doi.org/10.18653/v1/P17-1041