API parameter recommendation based on language model and program analysis

Cuong Tran Manh, Kien Tran Trung, Tan M. Nguyen, Thu-Trang Nguyen, Son Nguyen, Hieu Dinh Vo
Faculty of Information Technology, VNU University of Engineering and Technology
Email:{17020076, 18020026, 18020050, trang.nguyen, sonnguyen, hieuvd}@vnu.edu.vn

***Abstract*—APIs are extensively and frequently used in source code to leverage existing libraries and improve programming productivity. However, correctly and effectively using APIs, especially from unfamiliar libraries, is a non-trivial task. Although various approaches have been proposed for recommending API method calls in code completion, suggesting actual parameters for such APIs still needs further investigating. In this paper, we introduce FLUTE, an efficient and novel approach combining program analysis and language models for recommending API parameters. With FLUTE, the source code of programs is first analyzed to generate syntactically legal and type-valid candidates. Then, these candidates are ranked using language models. Our empirical results on two large real-world projects Netbeans and Eclipse indicate that FLUTE achieves 80% and +90% in Top-1 and Top-5 Precision, which means the tool outperforms the state-of-the-art approach.**

***Index Terms*—APIs, code completion, parameter recommendation, language model, program analysis**

I. INTRODUCTION

Application program interfaces (APIs) are extensively and frequently used in software development to leverage existing libraries and improve programming productivity. Unfortunately, due to a large number of API elements, the insufficiency of APIs' documents, and the unavailability of code examples, it is non-trivial to learn and remember APIs usages [1]. Therefore, programmers often struggle to precisely use APIs.

Indeed, code auto-completion is a practical solution for facilitating this challenge and has become an active research topic. Particularly, code completion is a crucial feature of the Integrated Development Environments (IDEs). This feature is expected to speed up the programming process by suggesting suitable elements in the statements. However, suggestions of current IDEs are generally based on type compatibility and visibility. Consequently, such suggestions are limited and unsuitable in complex cases.

There have been various studies about code completion [2], [3]. These studies explore different program properties and recommend different kinds of code elements. For instance, Liu et al. [3] have proposed a solution using language model and neural architecture to generate recommendations for identifiers, keywords, punctuation, etc. In another research, Nguyen et al. [2] combined program analysis and language model to generate the remaining tokens for incomplete code statements.

Regarding APIs recommendation, several techniques have been proposed and also obtained promising results [1], [4]. For example, Nguyen et al. [1] built a statistical model to recommend method calls for Android and Java APIs. Zhou et al. [4] also proposed a model for code completion of which performance after queries is boosted by user interaction information. However, these models mostly focus on method calls (i.e. the names of APIs). As a result, developers need to manually fill the corresponding actual parameters.

To accurately use APIs, not only method calls but also corresponding parameters are required. There is still a lack of research about auto-completing API parameters. In practice, 60% of the method declarations are parametrized and 50% of the actual parameters are not correctly suggested by the existing code completion systems [5]. There are several studies recommending API parameters [5], [6] by mining similar usage instances in the existing source code database. However, since candidates are genereted based on similar existing instances, the candidates could be invalid and syntactically incorrect regarding the recommending context.

In this paper, we present our early research on a novel and effective approach for auto-completing API actual parameters (aka arguments), FLUTE. For a recommendation point (i.e. the position in the program where the code completion request is made), we aim to suggest potential parameters which are *syntactically and semantically* correct. Firstly, to guarantee recommended parameters are syntactically correct regarding the coding program, *program analysis* is applied. In particular, we analyze the control flow graph (CFG) of the program to identify variables/objects and their data types. After that, based on type of the formal parameter, all the potential candidates are generated. Secondly, to address the challenge of semantic correctness, we train *language models* to predict the occurrence likelihood of each candidate in the recommending context. Importantly, to better capture the patterns of API usages, our models are trained on both the lexical form and the abstraction form of the data (i.e. Excode, Sec II-B). We also take into account the lexical similarity of the candidates and the formal parameter in the recommendation process.

Our experiments on two projects Eclipse and Netbeans show that FLUTE is better than the state-of-the-art approach about 35% in precision and 10% in recall. Specially, FLUTE obtains about 80% and 95% in precision for Top-1 and Top-10, respectively. These figures for recall are 66% and 79%.

In summary, our main contributions in this paper are:

- An novel approach for auto-completing API parameters
- Experimental evaluations on large real-world projects

II. APPROACH

Fig. 1 shows the overview of our approach. In general, FLUTE receives the program source code as input and then returns a list of candidates which are ranked according to their probabilities to be actual parameters of the being invoked methods.

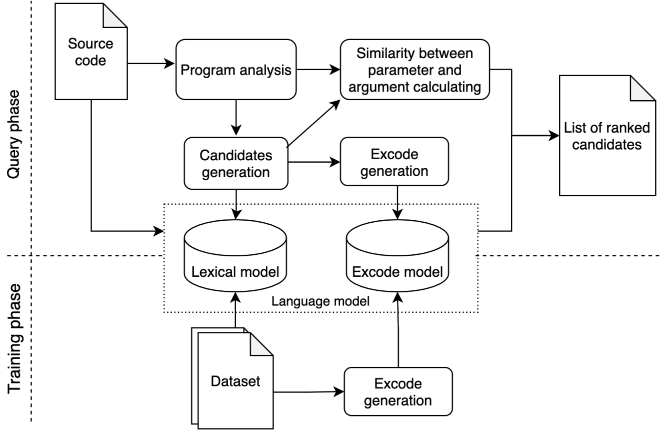

Fig. 1: Approach overview

The recommendation process includes four main steps. First of all, the program analysis technique is employed to generate a set of all type-valid and syntactically correct candidates. After that, to determine the suitability of each candidate in the recommending context, the occurrence likelihood of each candidate in the context is estimated by two language models namely lexical model and Excode model. In particular, the lexical model is an n-gram model trained on the lexical form of data, while the Excode model is another n-gram model trained on the abstraction form of data (i.e. Excode). In addition, we also calculate the similarity of the candidates and the formal parameters (lexsim calculation). Finally, these occurrence probabilities and similarity values are synthesized to rank the candidates.

A. *Potential candidates generation*

When a developer requests for suggestion, FLUTE will find candidates for the parameter. A candidate is valid if it satisfies two conditions: (1) it is accessible and syntactically correct with the context of the developing method/program; (2) it is type-valid with the formal parameter of the being invoked API.

In order to generate all the valid candidates for a recommendation point, first, FLUTE analyzes the source code to determine the current context including variables and constants with their corresponding types. Next, FLUTE identifies the expected type of the candidates by figuring out the data type of the corresponding formal parameter. Finally, based on the obtained context and the expected data type, all the potential candidates are generated.

We use the example shown in Fig. 2 to demonstrate our process in generating potential candidates. In this figure, at line 18 in class `Example`, the developer is invoking method

```
1  public class Student{
2     private String stud_name;
3     private int stud_id;
4     private float stud_gpa;
5     public static int num_of_students = 0;
6     Student(String _name, int _id, float _gpa){}
7     private boolean classify(){}
8     public int classify(float _gpa){}
9     public String classify(String _stud_name){}
10    public boolean classify(int _stud_id){}
11 }
12
13 public class Example{
14    public static void main(String args[]){
15       String name = "Bean";
16       int id = 100100;
17       float gpa = 4.0f;
18       Student s = new Student(name, id, gpa);
19       boolean status = s.classify(...
20    }
21 }
```

Fig. 2: Example of a parameter recommendation request. In this example, at line 19, the auto-complete tool is trigged to suggest the parameter of the method `classify`.

`classify`, and FLUTE is triggered to recommend an actual parameter for the method.

Firstly, *program analysis* is employed to analyze the context of the recommendation request. In this work, context refers to the code segment from the recommendation point back to the beginning of the developing method (lines from 14 to 19). More specifically, at this step, we analyze the CFG to determine all the available variables/objects and their corresponding data types. This is the foundation for the next steps.

Secondly, by the information obtained from the previous step, we determine the expected data type. As shown in Fig. 2, the method `classify` is a member of class `Student` and it is accessed by a `Student` object in method `main` of class `Example`. Indeed, there are four versions of `classify` in this class, however, only the fourth one (line 10) is fit for the current invoking context. The reason is that the return type of `s.classify` at line 19 is assigned to a variable type `boolean`, and it should be a `public` method due to being accessed outside of the class `Student`. In other words, the expected data type of the recommending parameter is `int`.

Thirdly, potential candidates are generated based on the valid variables/objects which are obtained from the program analysis step. Particularly, all the available variables/objects which have members (i.e., public fields and the return type of public methods)/the other public static fields and methods of the classes with appropriate type will be considered for candidates generation. For example, in Fig. 2, the expected type is `int`, the available variables/objects are `name`, `id`, `gpa`, and `s`. Therefore, the generated potential candidates set includes `id` which is available in the context, `Student.num_of_student` which is a public static field type `int` of class `student`, and the other appropriate members of JDK, such as `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, etc.

B. Excode transformation

This section presents extended code tokens (Excode) which we use to represent source code and the candidates generated in the previous phase for better capturing their general patterns. Table I shows the detailed rules that we use to generate Excode for code tokens. These rules are adapted from a previous work [7] so that they fit the new context. Specially, for code tokens, we encode their token types and data types in *Excode* and exclude specific information such as variable names and constant values because of the following reasons.

First, the *token type* of a code token refers to the role of the token in a program with respect to a programming language. The typical token roles include variable, field access, method call, type, keyword, operator, etc. Retaining token types preserves code structure and ensures syntactic correctness of candidates produced by the semantic language model.

Second, *data type* is included in Excode to differentiate tokens which are lexically identical but functionally distinct. As a result, the type constraints, the accessibilities to fields and methods of a variable can be validated. For example, two variables are both named `x`, however, one is a `String` and the other is an `Array`, these two variables will have accessibilities to different classes' members.

Third, *variable names* and *constant values* heavily depend on the particularities of the project, the programmer's practice, and the locality of the variables. Consequently, a language model training in the lexical form of code would encounter difficulties in identifying the code fragments having the same meaning but with different variables' names. For instance, although statements at line 5 and line 16 in Fig. 2 are lexically different, they have the same meaning of assigning a numerical value to an integer variable. Replacing the variable names with their types makes the representation of two fragments quite similar, therefore increasing that code pattern's frequency. On account of that, the language model not only captures the source code at a higher abstraction level but also learns code patterns from one place to suggest others. On the other hand, the names of data types, methods, and fields are kept since they are designed with reusability in mind.

C. Occurrence likelihood estimation

In order to provide candidates which are not only syntactically correct but also semantically correct regarding the recommending context, the occurrence likelihood of each candidate in the context is estimated by language models. In particular, we employ two n-gram models, namely the lexical model and the Excode model. The lexical model is trained on the lexical form of source code, meanwhile, the Excode model is trained on the Excode form. After that, these two models are used to estimate the probability of each candidate in the recommending context with the corresponding representation form. The reason for using these two models is that in code completion/suggestion, patterns of source code in both lexical level and high abstraction level are important and useful for code completion/suggestion [8].

D. Lexsim calculation

In practice, the actual parameters and the formal parameters are often significantly similar or completely dissimilar in terms of lexical tokens [9]. We also validated this statement in the two projects, Eclipse and Netbeans. It shows that 68.24% actual parameters and formal parameters are totally similar and 22.33% of them are completely different. This means that among the generated candidates, *the more lexical similar to the formal parameter a candidate is, the higher probability it is the expected actual parameter*. For example in Fig. 2, the expected parameter is `id` which is indeed more similar to the formal parameter (`_stud_id`, line 10) than the others.

To improve the accuracy of the parameter recommendation models, we also take into account the lexical similarity (lexsim) of the actual parameters and the corresponding formal parameters. Particularly, we leverage the formula proposed by Liu et al. [9] to calculate the similarity of a candidate (c) and the corresponding formal parameter (p), as follows:

$$lexsim(c,p) = \frac{|comterms(c,p)| + |comterms(c,p)|}{|terms(c)| + |terms(p)|} \quad (1)$$

In this formula,

- ***terms(s)***: The decomposition of *s*, based on underscores and capital letters.
- ***comterms(n1, n2):*** is the longest subsequence of *terms(n1)* and *terms(n2)*.

E. Score calculation

Overall, to evaluate whether a candidate should be the actual parameter of the recommendation point, we consider three criteria: (1) its occurrence likelihood evaluated by the lexical model, (2) its occurrence likelihood evaluated by the Excode model, (3) the lexsim of the candidate and the formal parameter. Particularly, *the score of a candidate is evaluated by the multiplication of these three values.* Since these values are from 0.0 (least suitable candidate) to 1.0 (most suitable candidate), we would like to leverage properties of multiplication such as multiplicative identity and the zero property. Finally, candidates are ranked by their scores in descending order.

III. Experiment

A. Experiment setup

To evaluate our approach, we conducted experiments on two large projects, Eclipse and Netbeans, which are used in a similar work [6]. Besides, we adopt 10-fold cross-validation to train and evaluate our models on each project. In this work, we employed JDT [10] to conduct program analysis and NLTK [11] library to build 6-gram MLE language models.

B. Metrics

In order to evaluate FLUTE and compare its performance with the state-of-the-art approach, we applied *precision* and

TABLE I: Excode construction rules

Token type	Construction rule	Example code → excode
Assignment operator o	ASSIGN(*name(o)*)	+= → ASSIGN(PLUS)
Unary operator o	UOP(*name(o)*)	++ → UOP(posIncrement)
N-ary operator o	OP(*name(o)*) [2]	+ → OP(PLUS)
Separator sp	SEPA(*sp*)	; → SEPA(;)
Bracket	To corresponding reserved token [2]	{ → OPBLK) → CLOSE_PART
Data type T	TYPE(T) [2]	short → TYPE(short)
Variable v	VAR(*type(v)*) [2]	num *(int)* → VAR(int)
Literal l	LIT(*type(l)*) [2]	0x01 → LIT(num), true → LIT(boolean)
Method call m	M_ACCESS(*type(caller(m))*, *name(m)*, *argcount(m)*)	str.length() → M_ACCESS(String,length,0)
Constructor call c	C_CALL(*class(c)*,*class(c)*)	new String(...) → C_CALL(String,String)
Field access f	F_ACCESS(*type(caller(f))*,*f*)	s.area → F_ACCESS(Shape,area)
Special literal	To corresponding reserved token [2]	0 → LIT(zero), null → LIT(null), ? → LIT(wildcard)
Control statement cs	STSTM(*name(cs)*) at the beginning of its scope, ENSTM(name(cs)) at the end of its scope	if {...} → STSTM{IF}...ENSTM{IF}, for (...) {...} → STSTM{FOR} ...ENSTM{FOR}, return → STSTM{RETURN} ...ENSTM{RETURN}
Type declaration c	CLASS{START,*class(c)*} at the beginning of its scope CLASS{END,*class(c)*} at the end of its scope	Class Car {...} → CLASS{START,Car} ...CLASS{END,Car}
Method declaration m	METHOD{*rt(m)*, *name(m)*} at the beginning of its scope, ENDMETHOD at the end of its scope	String toString() {...} → METHOD{String, toString} ... ENDMETHOD

TABLE II: The performance of FLUTE on all APIs in two projects Netbeans and Eclipse

Project	Top-k	Precision	Recall
Netbeans	1	78.91%	66.72%
	3	89.81%	75.93%
	5	92.11%	77.88%
	10	94.36%	79.78%
Eclipse	1	79.53%	66.00%
	3	91.11%	75.61%
	5	93.28%	77.41%
	10	95.14%	78.95%

recall measures. These two metrics are also used by Asaduzzaman [6].

$$Precision = \frac{relevant\ recommendations}{made\ recommendations} \quad (2)$$

$$Recall = \frac{relevant\ recommendations}{recommendation\ points} \quad (3)$$

In these formulas, *made recommendations* is the total number of times our completion technique recommends parameters; *relevant recommendations* is the total number of times the target parameter is presented in our Top-k recommendations; *recommendation points* is the number of parameters in the test set.

C. Experimental results

1) Whole-project setting: Table II shows evaluation results of FLUTE for the *Whole-project* settings, which include APIs in all the libraries of Eclipse and Netbeans. As can be seen, FLUTE achieves about 80% and 95% precision at Top-1 and Top-10, respectively. In addition, these figures for recall are 66% and 80%, respectively. These results show that for all the parameters of the APIs in these projects, 80% of them can be precisely recommended by FLUTE. In other words, FLUTE can help developers automatically complete 80% of all the actual parameters that they need to fill during programming. This could significantly help to improve the development productivity.

TABLE III: The performance of FLUTE and PARC on APIs of SWT (in Eclipse), AWT, and Swing (in Netbeans)

Project	Top-k	Precision		Recall	
		PARC	FLUTE	PARC	FLUTE
Eclipse	1	47.65%	**71.63%**	46.65%	**54.86%**
	3	65.05%	**79.21%**	63.68%	60.66%
	5	-	**80.87%**	-	61.94%
	10	72.26%	**85.71%**	70.73%	65.64%
Netbeans	1	46.46%	**76.48%**	44.86%	**62.06%**
	3	66.20%	**86.15%**	66.75%	**69.91%**
	5	-	**87.34%**	-	**70.87%**
	10	72.06%	**88.36%**	69.57%	**71.70%**

2) Specific libraries setting: To compare the performance of FLUTE and PARC [6], we conduct experiments on the APIs of the library SWT in the Eclipse project and the libraries AWT and Swing in the Netbeans project (Table III). Overall, FLUTE is better than PARC about 35% in precision and 10% in recall. Specifically, at Top-1, the number of parameters that FLUTE correctly recommends at the first position in the ranked list is 1.6 times higher than that number of PARC (71.63% vs. 47.65%). This means that among 10 requests of parameter recommendation, there are around 7 requests that the first candidate in the recommended list exactly hits the expected target, while this figure for PARC is nearly 5. Moreover, with FLUTE, if developers investigate about 10 candidates in the recommended list for each request, they can find the correct

parameters for nearly 90% of requests.

IV. Threats to validity

There are several threats to validity in our work. First, a threat may come from the dataset that we used for our experiments. In this work, we used only two projects Eclipse and Netbeans, therefore, our results may not be generalized for all kinds of APIs in different software projects. To reduce this threat, we chose large real-world projects which contain multiple modules and are widely used in many other studies. Also, we plan to collect more data for future work. Second, another threat may come from our implementation. In order to reduce this threat, we carefully tested and manually reviewed our implementation.

V. Related work

Various approaches have been proposed for APIs auto-completion, however, they mostly focus on completing the name of the method [1], [4], [12]. Specially, Nguyen et al. [1] extract method call sequences from mobile apps' bytecode and then build a statistical model for recommending method calls of Android APIs and Java APIs. Besides, Xie et al. [12] propose a context-aware API recommendation approach which considers not only the APIs from the third-party libraries but also project-specific APIs. Moreover, in order to enhance the effectiveness and personalize the API recommendation, Zhou et al. [4] leverage the user interaction information with the recommended results to train a learning-to-rank model to re-rank the suggested candidates. These approaches have obtained promising results on auto-completing method names, nevertheless, they leave the tasks of completing the actual parameters for developers.

About API parameters recommendation, there are Precise [5] and PARC [6], [13], which are both developed as Eclipse plugins. These approaches are based on the key idea that APIs are often used in similar contexts. They create a database of API usage contexts from the existing codebase. Then, for a parameter recommendation query, an ordered list of candidates is generated by finding similar usage instances in the database. However, by these approaches, the candidates are generated from the existing code, thus, they could be unavailable and even invalid in the coding program. For FLUTE, the notable difference is that FLUTE applies the program analysis technique to analyze the current program and generate lists of valid candidates. After that, to better obtain their general patterns, candidates are represented by *Excode*. Finally, they are scored and ranked by language models. Therefore, all the recommended candidates from FLUTE are at least syntactically correct and type conformance with the expected formal parameters. Furthermore, by this method, FLUTE can overcome the limitation of the existing approaches when the recommending contexts are totally different from the usage patterns in the database.

VI. Conclusion

APIs are extensively used in developing programs, various approaches have been proposed to improve productivity by auto-completing API invocations. However, most of these studies concentrate on recommending method names, the actual parameters are left for developers. This paper introduces FLUTE, a novel method for auto-completing API actual parameters. First, FLUTE conducts program analysis to generate all valid candidates for the parameter requiring positions. Next, these candidates are scored and ranked by the language models. Finally, FLUTE outputs a list of candidates which are ranked according to their suitability with the context of the program. Our experimental results show that FLUTE can obtain about 80% in precision and 66% in recall for Top-1. Furthermore, these numbers for Top-5 candidates are +90% and nearly 80%, respectively.

References

[1] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: a statistical approach," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 416–427.

[2] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 710–721.

[3] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[4] Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma, and H. C. Gall, "Boosting api recommendation with implicit feedback," *IEEE Transactions on Software Engineering*, 2021.

[5] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical api usage," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 826–836.

[6] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring api method parameter recommendations," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 271–280.

[7] S. Nguyen, T. N. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 710–721. [Online]. Available: https://doi.org/10.1109/ASE.2019.00072

[8] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542.

[9] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1063–1073.

[10] "Eclipse Java development tools (JDT)." [Online]. Available: https://www.eclipse.org/jdt/

[11] "Natural Language Toolkit." [Online]. Available: https://www.nltk.org

[12] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "Hirec: Api recommendation using hierarchical context," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 369–379.

[13] M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Parc: Recommending api methods parameters," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 330–332.