KG2Code: Correct Code Examples Mining Service Based on Knowledge Graph for Fixing API Misuses

Yangqi Zhang(✉), Zhirui Kuai, Wenjin Yao, Zhiyang Zhang, and Li Kuang(✉)

Department of Software Engineering, School of Computer Science and Engineering, Central South University, Changsha, China
{zhangyangqi,8209180621,8209180518,zzy415573678,kuangli}@csu.edu.cn

Abstract. API misuse has become an important factor restricting the quality of software services. Existing API misuse detectors based on the API-constraint knowledge graph can not intuitively assist developers in fixing the API misuse. Correct code examples are more direct and straightforward for developers to modify and debug code. Therefore, we first enrich the API-constraint knowledge graph. Besides, we publish a service called KG2Code, which can map the API-constraint Knowledge Graph to the Correct Code examples. According to the different types of constraint relations in the API-constraint knowledge graph, we design a code snippet mining framework that extracts the corresponding correct API usage pattern from over 9528K Java repositories GitHub. KG2Code is implemented by the interactive visualization website. It helped users (1) learn how to use an unfamiliar API or fix an API misuse and (2) understand why API misuse occurs.

Keywords: Quality of software services · API-constraint knowledge graph · Mining software repositories

1 Introduction

If developers do not comply with API usage constraints in the actual software development, it will lead to API misuse or even software crash, which causes the software to be unreliable. For example, when using the *File* in Java, *File.createNewFile(String)* can only be called after *File.exist()* to avoid FileNotFoundException. Therefore, developers are often concerned with the solution to API misuse. In fact, whether in Github, in StackOverflow(SO), or the API reference documentation, there will be much implicit or explicit information to fix the API misuse.

The API reference documentation includes a wealth of knowledge in different aspects of the API, such as functionalities, constraints, directives, caveats, and resource specifications. The knowledge of constraint descriptions helps developers understand the correct usage of the API, making it easy to use the API. However, the constraint knowledge is scattered within the document of the API

© Springer Nature Switzerland AG 2021
H. Hacid et al. (Eds.): ICSOC 2021, LNCS 13121, pp. 875–882, 2021.
https://doi.org/10.1007/978-3-030-91431-8_65

elements (e.g., class), leading to many challenges for API constraint knowledge discovery and summarization. The Q&A knowledge forum (e.g., StackOverflow) also provides related API misuse questions and answers, but questions about API misuse are not necessarily correct, and many answers are not clear [1]. There are a large number of API usage examples in Github. Through these examples, developers can quickly understand the code and modify the incorrect usage of the API. However, it is difficult to locate the API we need from the massive Github repositories. Therefore, fixing the API misuse through the above three ways is not feasible in practice.

Inspired by the SO platform, we consider that correct code examples can better improve the efficiency and effectiveness of developers than API misuse description. Therefore, We publish a service called KG2Code, which can map an API-constraint knowledge graph to correct code examples. For a given API constraint triple, we extract correct code examples from the Java Github repositories. First, we crawl Java repositories and filter low-quality repositories by distributed software mining infrastructure [2]. And according to the class name and method name, each method in the repository can be found. Next, traverse the Abstract Syntax Trees (ASTs) of the two APIs with different constraint relations, and capture the correct code pattern of the API by converting different data such as control structure, calling sequence, guard conditions, etc. Finally, remove the part that we are not interested in by program slicing.

This paper makes the following contributions:

1. We expand the original API-constraint knowledge graph by adding constraint relations and merging more data;
2. We conduct an empirical study that reveals that correct code examples can effectively assist developers. We firstly propose an approach that can extract correct code examples from Github based on API-constraint knowledge graph, and we implement it as a visualization website;
3. Our manual inspection confirms the high quality of the correct examples mined by KG2Code.

2 Related Work

API pattern mining is our significant part of KG2Code. API pattern mining is divided into three parts: (1) By modeling the program as a code sequence or item set and inferring programming rules by mining frequent sequences, or frequent itemsets [3,4]. (2) Researchers apply formal concept analysis [5] to extract the call sequence in the program [6]. (3) Researchers mine the guard conditions of APIs by applying predicate mining technology [7].

Inspired by examplecheck [7], KG2code also mines the Github software code repositories, but the difference is that we mine through specific patterns in the knowledge graph. According to the knowledge graph, the calling sequence, guard conditions, and specific conditions of the control structure of the APIs correspond to different types of constraint relations. Besides, the SMT Solver [8] is used to determine the equivalence of the guard conditions.

3 Construction of API-Constraint Knowledge Graph

To mine correct code examples, we first need to construct the API constraint knowledge graph. Expect the four API constraint relations, which include call-order, state-checking, value-checking, and trigger, we add three types of fine-grained constraint relationships: redundant-checking, duplicate-checking, and synchronized-checking, which also corresponds to the frequent API misuse types in the MuBench [9]. We define the seven constraint relations which extend the prior work [10] for the first time. As the construction of the API constraint knowledge graph is similar to the prior work, A detailed description will not be given here. The overall construction framework of the API constraint knowledge graph is shown in Fig. 1.

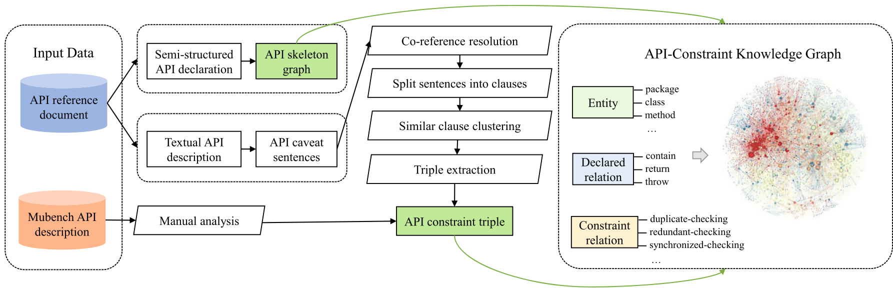

Fig. 1. The construction of API-constraint knowledge graph

The entity of knowledge graph consists of API elements: package, class, method, exception, parameter, return value, and value literals. Literal values such as null, −1, true, negative numbers, or a range such as [0, 9]. The knowledge graph contains two types of relations: declaration relations and constraint relations. Declare relations such as a package contains a class, a class contains a method, a method returns a numeric literal, or a method throws an exception. In terms of the constraint relations, by referring to the most frequent API misuse types of the MuBench, we added three fine-grained constraint relationships, and we expanded the constraint types to seven types. Now let's discuss the specific usage of these seven constraint types in the knowledge graph. (see Fig. 2.)

Call-Order: API misuse caused by missing an API call or incorrect call order. It means the method has to be called before a certain method or the method has to be called after a certain method to avoid API misuse. For example, the file should be closed after being written to prevent resource leakage, which means like *PrintWriter.close*() should be called after the PrintWriter method (close the PrintWriter after writing to avoid resource leak). The knowledge graph can also express chain calls through multi-hop relations.

State-Checking: API misuse caused by missing state-checking or incorrect state-checking. For example, we need to check the boolean value of *hasnext*()

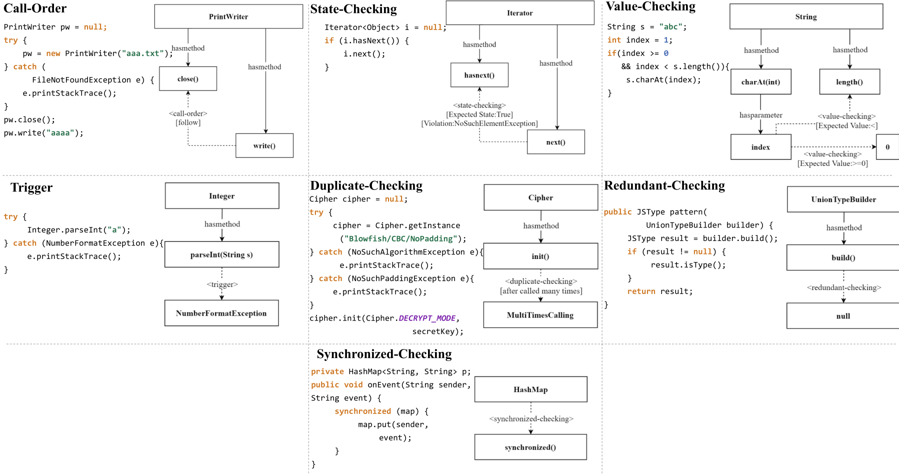

Fig. 2. The constraint relations with the corresponding knowledge graph and code example

or $isempty()$ before $Iterator.next()$. It is correct when $hasnext()$ is true, or $isempty()$ is false. Otherwise, it will cause API misuse, which leads to NosuchElementexception. It is worth noting that it is very easy to confuse call-order because it seems to be an order relation between the two methods. We have to pay attention to that state-checking requiring state-checking on the method's return boolean value, while call-order does not need it.

Value-Checking: Determine whether the value of the parameter in the API follows API usage constraints of the method. For example: for the $ArrayList.Get()$ method, and it is necessary to check if the index is out of bounds.

Trigger: Trigger is to check whether the exception handling is missing in the code, which leads to the API misuse. For example, $Interge.parseInt()$. If the string does not contain a parsable integer, $Interge.parseInt()$ may throw a NumberFormatException.

Duplicate-Checking: If some APIs are called multiple times, they will be misused. For example, $cipher.init()$ is called twice along one possible execution path, which causes an infinite loop.

Redundant-Checking: A method does a redundant checking, which prevents a necessary part of a usage and is executed along a certain execution path. One case is redundant null checks. For example: $UnionTypeBuilder.build()$ returns a JSType that can never be null. Branching on a null check, therefore, results in dead code.

Synchronized-Checking: In multi-threaded environments, some container classes must be the thread-unsafe condition. For example, the $HashMap$ in

JDK1.8 is thread-unsafe if a usage does not obtain a lock before updating a $HashMap$ that is accessed from multiple threads.

4 KG2Code

KG2Code consists of two phases. One is the offline phase, which extracts the constraint triples in the knowledge graph and mines correct code examples from Github's high-quality repositories. While the other is the online phase, which generates the KG2Code results by the visualizing website. The KG2Code overview is shown in Fig. 3.

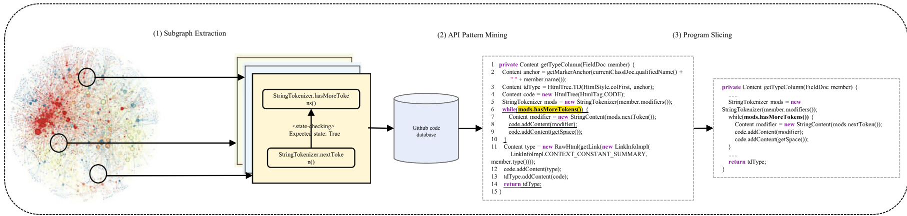

Fig. 3. Overview of KG2Code

4.1 Extract the Subgraph

It's simple to extract the subgraph from the API constraint knowledge graph. We extract the subgraph from neo4j by different relation types. However, we only extract the subgraph for four API constraint types of relations: call-order, state-checking, synchronized-checking, and trigger. This is because the relationship type needs to correspond to the code pattern that can be extracted-such as value-checking, the variables involved in a program change dynamically during the actual running of the program, and in some value-checking examples, getting the correct code structure requires checking if the variables are in an interval. However, it is represented by another variable in the range of the program, and it is difficult to ensure that the code snippet meets the requirements of value-checking.

4.2 Extract the Structure of the API

We only extract the subgraph for four API constraint types of relations: call-order, state-checking, synchronized-checking, and trigger.

For a given API, we search for code snippets on GitHub based on the constrained triples, which are from the knowledge graph. We used a distributed software mining infrastructure to filter out some of the low-quality Java repositories by some limits, such as the number of repository contributors and the

number of version updates. We only consider repositories with at least 100 revisions and 2 contributors. Then we use the relevant syntax to traverse ASTs of Java files and match the methods and classes of interest by the name of the class and the name of the method. In order to extract API-specific patterns, KG2Code models each program as a structured call sequence, which extracts variable names, but still retains the call sequence, control structure, guard conditions. Furthermore, we extracted different API patterns for different constraint types.

For the call-order relation, we need to record the order of API calls properly. In some cases, the methods are not called sequentially, for example, $code().addContent(getSpace())$; this expression is a case of nested calls, we assume that the method inside the parentheses will complete the call first when it is run so that this sentence will be processed as 'code -> getSpace -> addContent'.

For the state-checking relation, we need to keep the guard condition of each API call. We use the conjunction of the lifted predicates in all relevant control structures. In other words, we record all the branching conditions on the method call path and connect them with &. We then use the Z3 solver to determine whether the two conditions are equivalent. We will formalize the equivalence of two guard conditions as a satisfiability problem.

For the synchronized-checking and trigger relation, we traverse the abstract syntax tree to retrain the method's control structure, including try-catch, switch-case, synchronized, return, various loops, and so on. For the synchronized-checking relation, we need to record whether the synchronized modifier is added. For the trigger relation, we need to check whether the API is contained within the associated try-catch block.

Finally, we matched the correct code pattern for each API by the constraint relation type and counted the number of correct code patterns conforming to each API. The GitHub link of this file is reserved.

4.3 Program Slicing

We need to do static code slicing to filter out any statements that are not related to the API method of interest. In this step, we retain the control structure of the method obtained in the previous step. On this basis, we record the variables involved in the API of interest, including the caller of the API, the receiver of the API, and the parameters used by the API. We use these variables for static code slicing.For example (see Fig. 3.), '$Contentmodifier = newStringContent(mods.nextToken());$', this sentence contains the method of interest: $nextToken()$, so the variables we use to slice are '$mods$' and '$modifier$'. All statements that contain these two variables before and after this statement will be retained. The resulting statement and associated control structures make up the result of the slice.

5 Tool Implementation and Evaluation

We built an API knowledge graph for JDK 1.8[1]. The API constraint knowledge graph includes 52,754 entities and 85,196 relationships, which includes 2,397 classes, 26,902 API methods, 50,711 parameters, 648 exceptions. There are 58025 declared relationships and 27,171 constraint relationships, including 1586 call-order relations, 24,395 value-checking relations, 890 state-checking relations, 109 duplicate-checking relations, 85 redundant-checking relations, and 106 synchronized-checking relations. These API-constraint relations involve 19,385 methods, 6,823 parameters, and 5,347 return and 10,289 throw relations.

We scanned more than 9 million Java repositories on the 2019 October GitHub dataset. We have implemented KG2Code as an online website. The website front-end is implemented by using D3.js, and the back-end is implemented by built-in python and nodeJS. Developers can enter a search query of the required API. when querying *java.swing. StringTokenizer.nextToken*, it shows a description of the API, the display of the API constraint subgraph from the knowledge graph, the corresponding code example, and the number of code examples with the same pattern. Each API can also be accessed through the link to the original Github repository (see Fig. 4.).

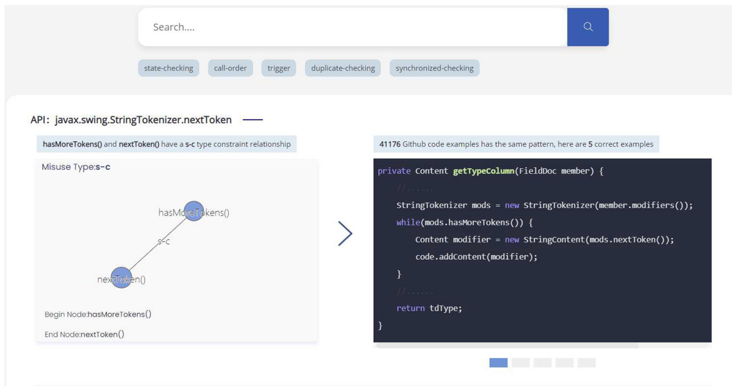

Fig. 4. A snapshot of the KG2Code website

We can map 108 API-constraint relations in the MuBench to correct examples that correspond to the API pattern. To check whether the correct examples mined by KG2Code indeed conform to the desirable API usage. We manually check 300 random code snippets mined by KG2code involving 30 API misuses from the MuBench. Each API misuse contains 10 correct examples. They all match the correct usage Java file provided. These results demonstrate that our proposed approach the correct examples mined by KG2Code are effective.

[1] https://docs.oracle.com/javase/8/docs/api.

6 Conclusions and Future Work

This paper first proposes a service named KG2Code, a mining framework based on API-constraint knowledge graph for correct code examples in Github. Furthermore, we expand the previous API-constraint knowledge graph with three more fine-grained types of constraint relations, derived from API reference documentation and the MuBench. The quality of correct examples has been demonstrated by manual inspection. In the future, we will tackle the challenges that the API reference documentation and Github code repositories will continue to evolve and update as time goes on.

Acknowledgement. This work has been supported by the National Key R&D Program of China (No. 2018YFB1402800), the National Natural Science Foundation of China (No. 61772560), and the Fundamental Research Funds for the Central Universities of Central South University (No. 2021zzts0746).

References

1. Ren, X., Sun, J., Xing, Z., Xia, X., Sun, J.: Demystify official API usage directives with crowdsourced API misuse scenarios, erroneous code examples and patches. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 925–936 (2020)
2. Upadhyaya, G., Rajan, H.: On accelerating source code analysis at massive scale. IEEE Trans. Softw. Eng. **44**(7), 669–688 (2018)
3. Li, Z., Zhou, Y.: PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Softw. Eng. Notes **30**(5), 306–315 (2005)
4. Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage API usage patterns from source code. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 319–328. IEEE (2013)
5. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer Science & Business Media, New York (2012)
6. Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6,000 projects: lightweight cross-project anomaly detection. In: Proceedings of the 19th international symposium on Software testing and analysis, pp. 119–130 (2010)
7. Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M.: Are code examples on an online Q&A forum reliable? A study of API misuse on stack overflow. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 886–896. IEEE (2018)
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
9. Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M.: MUBench: a benchmark for API-misuse detectors. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 464–467 (2016)
10. Ren, X., et al.: API-misuse detection driven by fine-grained API-constraint knowledge graph. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 461–472. IEEE (2020)