CodeSpider: Automatic Code Querying with Multi-modal Conjunctive Query Synthesis

Chengpeng Wang
cwangch@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Abstract

Querying code conveniently is an appealing goal to the software engineering community. This work advances this goal by presenting a multi-modal query synthesis technique. Given a natural language description and code examples, we synthesize a conjunctive query extracting positive examples and ignoring negative ones, which is further used to query desired constructs in a program. To prune the huge search space, we generate well-typed query sketches for refinement by analyzing code examples and API signatures. We also introduce two quantitative metrics to measure the quality of candidate queries and select the best one. We have implemented our approach as a tool named CodeSpider and evaluated it upon sixteen code querying tasks. Our experimental results demonstrate its effectiveness and efficiency.

CCS Concepts: **• Software and its engineering → Automatic programming**; **• Human-centered computing → User interface programming**.

Keywords: Multi-modal Program Synthesis, Query Synthesis, Code Querying

ACM Reference Format:
Chengpeng Wang. 2022. CodeSpider: Automatic Code Querying with Multi-modal Conjunctive Query Synthesis. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3563768.3563954

1 Introduction

Programmers often need to explore their projects by code querying in various scenarios of development, including API understanding [7], code refactoring [10], and bug detection [8]. Although there have been several industrial products and academic works on code querying, they often compromise between ease of use and capability [1, 5, 11]. Specifically, most mainstream IDEs [5] provide convenient built-in templates but only support the string matching-based search of restrictive program constructs. Besides, query-based program analyzers, e.g., CodeQL [1], store program constructs with relational representations as its components and provide a collection of APIs for query writing. However, laborious manual efforts are involved in query language learning and query writing, which degrades usability significantly.

In this work, we aim to synthesize a query in a code query language for a code querying task. Specifically, a querying task is instantiated by a specification, where a natural language (NL) description depicts the querying intent, and code examples show the constructs which should be or not be queried. For example, Fig. 1 shows the query specification of finding the methods receiving a *Log4jUtils* object as a parameter. A synthesized query leverages the components and APIs in the query language to specify the relations of constructs as the querying condition, which is conjunctive and permits string predicates, supporting construct filtering and string matching-based search simultaneously. A practical solution to our multi-modal synthesis problem would provide a user-friendly interface for code querying.

Description: Methods receiving a parameter with Log4jUtils type.	
public void foo(Log4jUtils a) { return; }	// positive example
private void goo(int a) { return; }	// negative example

Figure 1. An example of query specification

Unfortunately, it is non-trivial to synthesize the query for a code querying task. First, a large number of components and APIs in the query language induce a huge search space [2, 4, 6], increasing the difficulty of synthesizing the query efficiently. Second, there are often multiple query candidates satisfying the constraints induced by examples, while many of them can suffer the over-fitting problem. To obtain the target query, we have to synthesize queries efficiently and measure the quality of query candidates effectively.

To solve the problem, we present a new synthesis technique CodeSpider where the analyses of code examples, query language library, and the NL description collaborate simultaneously. To prune the search space, we analyze code

SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9901-2/22/12…$15.00
https://doi.org/10.1145/3563768.3563954

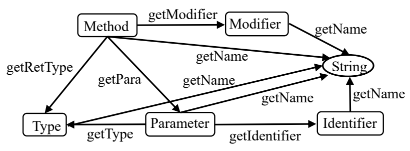
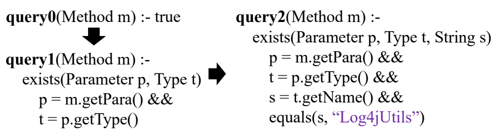


Figure 2. The TTN of Fig. 1



```
query0(Method m) :- true
query1(Method m) :-
  exists(Parameter p, Type t)
    p = m.getPara() &&
    t = p.getType()
query2(Method m) :-
  exists(Parameter p, Type t, String s)
    p = m.getPara() &&
    t = p.getType() &&
    s = t.getName() &&
    equals(s, "Log4jUtils")
```

Figure 3. An example of query refinement

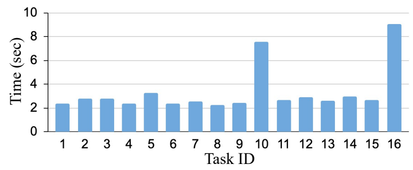


Figure 4. Time cost of the synthesis

examples to narrow down the components and APIs potentially used in the query, and leverage the API signatures to construct a *type-transition net* (TTN), of which each sub-graph induces a query sketch. Then we enumerate well-typed conjunctive queries by refinement and examine the feasibility by evaluating them upon code examples. To select the best query for the querying task, we introduce the *structural complexity* and the *entity coverage* to measure the quality of candidate queries. We have evaluated CodeSpider upon sixteen code querying tasks, which cover various kinds of program constructs. CodeSpider successfully synthesizes all the queries and consumes 3.35 seconds averagely.

2 CodeSpider in a Nutshell

Take the query specification in Fig. 1 as an example. CodeSpider works with the following three stages.

Sketch Generation. We diff the program constructs of positive and negative examples to identify possible components in the query. In Fig. 1, for example, only the methods, modifiers, types, parameters, and identifiers are different, and thus, the corresponding components can occur in the query. Then we construct the TTN according to API signatures and generate all the query sketches by enumerating its subgraphs. To support string matching, we consider the APIs that return strings in the TTN construction by default.

Query Refinement. We enumerate the sub-graphs of the TNN and evaluate induced queries upon the code examples. Specifically, we start from the weakest conjunctive queries, in which the condition is *true*, and refine the queries by adding new edges and appending string constraints. If a query misses positive examples, we discard it and end the refinement. Otherwise, we continue to strengthen conjunctive conditions until we obtain a candidate query that separates the positive and negative examples. In Fig. 3, for example, `query0` and `query1` both identify the negative examples, and we finally obtain a candidate `query2` by refining `query1`. After the refinement, we can obtain all the candidate queries.

Query Selection. To select the query, we propose the *entity coverage* and the *structural complexity* to quantify the query quality. First, we extract the named entities [3] of the NL description, which indicate possible components, APIs, and string literals in the query, and the target query tends to cover as many entities as possible. Second, we measure the structural complexity of a query by the number of its conjunctions. When there are multiple candidate queries with the maximal entity coverage, we select the one with the lowest structural complexity, which induces the weakest querying condition. For example, `query2` is the target query fitting the specification in Fig. 1. Although we can enforce the modifier to be "public" by refinement, this does not increase the entity coverage but increases the structural complexity.

Table 1. Description of querying tasks

ID	Description	(#P,#N)	(#C,#A)	Kind
1	Float variables of which the identifier contains "cash"	(3, 1)	(4, 4)	Var
2	Cast expressions from double-type to float-type	(1, 2)	(6, 7)	Expr
3	Expressions comparing long int with int	(1, 2)	(3, 6)	Expr
4	Cast expressions casting long to int	(2, 1)	(6, 7)	Expr
5	Expressions comparing a variable and Boolean literal	(1, 3)	(4, 5)	Expr
6	New expressions of ArrayList	(1, 1)	(3, 3)	Expr
7	Logical-and expressions with a literal as an operand	(2, 2)	(4, 5)	Expr
8	The import of LocalTime	(2, 1)	(3, 4)	Stmt
9	The import of the classes in log4j	(1, 1)	(2, 2)	Stmt
10	Labeled statements	(2, 2)	(1, 0)	Stmt
11	If-statements with a Boolean literal as a condition	(2, 1)	(2, 1)	Stmt
12	For-statements with a Boolean literal as the condition	(2, 1)	(2, 1)	Stmt
13	Public methods with void return type	(2, 1)	(5, 6)	Method
14	Methods receiving a parameter with Log4jUtils type	(2, 1)	(4, 4)	Method
15	Classes with a login method	(2, 1)	(3, 3)	Class
16	Classes containing a field with float type	(1, 1)	(4, 4)	Class

3 Evaluation

We implement CodeSpider to synthesize the queries for a query-based program analyzer in Ant Group, while CodeSpider is general enough to synthesize queries in other code query languages [1]. We select the querying tasks in [9] as a benchmark and specify the query specification manually. As shown in the column (#P, #N) of Table 1, the numbers of positive and negative examples are both no larger than 3. Then we measure the time cost of CodeSpider and examine whether the synthesized queries are the target queries.

CodeSpider successfully synthesizes all the target queries for the sixteen querying tasks. The numbers of components and APIs in the synthesized query are shown in the column (#C, #A) in Table 1. Actually, CodeSpider follows the principle of Occam's Razor. The queries cover the maximal numbers of entities in the NL description, mitigating the over-fitting problem caused by code examples. Also, it tends to select the simplest queries without unnecessary conditions, which have the lowest structural complexity, so that the queries have better generalization capability. Moreover, CodeSpider obtains high efficiency. Fig. 4 shows that it finishes any synthesis in ten seconds, and most of the tasks can be handled in less than four seconds. The average time cost is only 3.35 seconds, evidencing its high efficiency.

References

[1] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[2] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. 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*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 599–612. https://doi.org/10.1145/3009837.3009851

[3] Jenny Rose Finkel, Trond Grenager, and Christopher D. Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *ACL 2005, 43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, 25-30 June 2005, University of Michigan, USA*, Kevin Knight, Hwee Tou Ng, and Kemal Oflazer (Eds.). The Association for Computer Linguistics, 363–370. https://doi.org/10.3115/1219840.1219885

[4] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-directed program synthesis for RESTful APIs. In *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 122–136. https://doi.org/10.1145/3519939.3523450

[5] IntelliJ IDEA. 2022. Structural search and replace. https://www.jetbrains.com/help/idea/structural-search-and-replace.html

[6] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for fold: synthesis-aided API discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 205:1–205:27. https://doi.org/10.1145/3428273

[7] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. 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, 690–701. https://doi.org/10.1145/2950290.2950341

[8] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. ARBITRAR: User-Guided API Misuse Detection. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1400–1415. https://doi.org/10.1109/SP40001.2021.00090

[9] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code using Query-by-Example. In *UIST ’21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, Jeffrey Nichols, Ranjitha Kumar, and Michael Nebeling (Eds.). ACM, 84–99. https://doi.org/10.1145/3472749.3474737

[10] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 800–810. https://doi.org/10.1145/3180155.3180194

[11] Charles Zhang and Hans-Arno Jacobsen. 2004. PRISM is research in aSpect mining. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 20–21. https://doi.org/10.1145/1028664.1028676