Synthesizing Conjunctive Queries for Code Search

Chengpeng Wang
The Hong Kong University of Science and Technology, China

Peisen Yao
Zhejiang University, Hangzhou, China

Wensheng Tang
The Hong Kong University of Science and Technology, China

Gang Fan
Ant Group, Shenzhen, China

Charles Zhang
The Hong Kong University of Science and Technology, China

Abstract

This paper presents SQUID, a new conjunctive query synthesis algorithm for searching code with target patterns. Given positive and negative examples along with a natural language description, SQUID analyzes the relations derived from the examples by a Datalog-based program analyzer and synthesizes a conjunctive query expressing the search intent. The synthesized query can be further used to search for desired grammatical constructs in the editor. To achieve high efficiency, we prune the huge search space by removing unnecessary relations and enumerating query candidates via refinement. We also introduce two quantitative metrics for query prioritization to select the queries from multiple candidates, yielding desired queries for code search. We have evaluated SQUID on over thirty code search tasks. It is shown that SQUID successfully synthesizes the conjunctive queries for all the tasks, taking only 2.56 seconds on average.

2012 ACM Subject Classification Software and its engineering → Automatic programming; Human-centered computing → User interface programming

Keywords and phrases Query Synthesis, Multi-modal Program Synthesis, Code Search

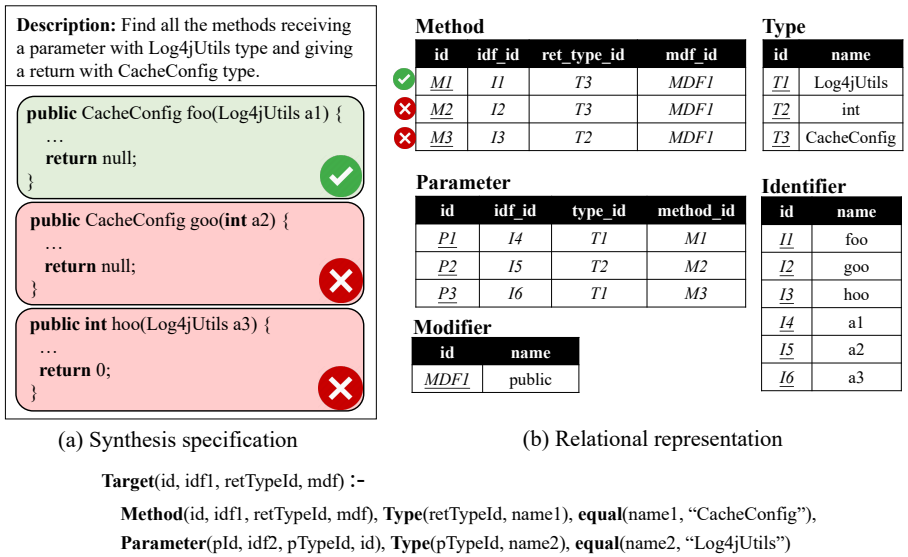
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.

Acknowledgements We thank the anonymous reviewers, Xiao Xiao, and Xiaoheng Xie for their helpful comments. Peisen Yao is the corresponding author.

1 Introduction

Developers often need to search their code for target patterns in various scenarios, such as API understanding [29], code refactoring [56], and program repair [47]. According to recent studies [34, 30], existing efforts have to compromise between ease of use and capability. Most mainstream IDEs [23] only support string match or structural search of restrictive grammatical constructs although complex user interactions are not required. Besides, static program analyzers, such as Datalog-based program analyzers [44, 35, 3], provide deep program facts for users to explore advanced patterns, while users have to customize the analyzers to meet their needs [12]. For example, if users want to explore code patterns with the Datalog-based program analyzer CODEQL [3], they have to learn the query language to access the derived relational representation. However, there always exists a non-trivial gap between a user's search intent and a customized query supporting code search. A large number of underlying complex relations can make query writing involve strenuous efforts, especially in formalizing search intents and debugging queries, which hinders the usability of CODEQL in the scenarios of code search.


© Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang;
licensed under Creative Commons License CC-BY 4.0



(a) Synthesis specification

(b) Relational representation

(c) Conjunctive query

Figure 1 A motivating example[1]

Our Goal. We aim to propose a query synthesizer to unleash the power of a Datalog-based program analyzer for code search. To show the search intent, a user can specify a synthesis specification consisting of positive examples, negative examples, and a natural language description. Specifically, positive and negative examples indicate desired and non-desired grammatical constructs, respectively, while the natural language description shows the search intent by a sentence. Our synthesizer is expected to generate a query separating positive examples from negative ones, which can support code search in the editor. In this work, we focus on conjunctive queries, which have been recognized as queries of an important form to support search tasks [17].

Consider a usage scenario: Find all the methods receiving a parameter with Log4jUtils type and giving a return with CacheConfig type. The user can provide the synthesis specification shown in Figure 1(a). With the relational representation in Figure 1(b) derived from the examples, our synthesizer would synthesize the conjunctive query in Figure 1(c) to express the search intent. In particular, our synthesis specification is easy to provide. The users can often copy desired grammatical constructs from an editor as positive examples [34] and then mutate them to form negative ones. Meanwhile, they can express their need to search code with a brief sentence as the description. Thus, an effective and efficient synthesizer enables the users to express the search intent from a high-level perspective, serving as a user-friendly interface for code search.

Challenges. Nevertheless, it is far from trivial to synthesize a conjunctive query for code search. First, a Datalog-based program analyzer can generate many relations with multiple attributes as the relational representation. For example, CodeQL exposes over a hundred relations to users for query writing [45]. The various choices of selecting relations and enforcing conditions on attributes induce a dramatically huge search space in the synthesis, which can involve both the comparisons between attributes and string constraints, posing a significant challenge to achieving high efficiency. Second, there often exist multiple query candidates that separate positive examples from negative ones, while several candidates can

[1] We show five relations as examples, while a Datalog-based analyzer can derive over a hundred relations.

suffer from the over-fitting problem, failing to express the search intent with no bias [53]. An ineffective query candidate selection would mislead the synthesizer into returning wrong queries and further cause the failure of code search.

Existing Effort. There are three major lines of existing effort. The first line of the studies utilizes input-output examples to synthesize queries in various forms, such as analytic SQL queries [58, 13] and relational queries [42, 46]. Although the queries often have expressive syntax, the synthesizers only take a few relations as input, not facing hundreds of relations as ours. The second line of approaches is the component-based synthesis technique [14, 38], which leverages type signatures to enumerate well-typed programs. However, a significant number of comparable attribute pairs still induce an explosively huge search space even if we adopt the techniques by guiding the search with the schema. The third line of the studies derives program sketches from natural language descriptions via semantic parsing [28] and prioritizes feasible solutions with probability models [54, 4]. Unfortunately, the ambiguity of natural languages and the inadequacy of the training process can make a semantic parser ineffective and eventually miss optimal solutions [40]. It is also worth noting that existing techniques do not attempt to select a feasible solution that maximizes or minimizes a specific metric. Although several inductive logic learning-based techniques adopt heuristic priority functions to accelerate the synthesis [46], they do not guarantee the optimality of the synthesized queries, and thus can not resolve the query candidate selection in our problem.

Our Solution. Our key idea comes from three critical observations. First, only a few relations contribute to separating positive examples from negative ones. For example, the methods in Figure 1(a) have the same modifier, indicating that the relation Modifier is unnecessary. Second, adding an attribute comparison expression or a string constraint to the condition of a conjunctive query yields a stronger restriction on grammatical constructs. If a query misses a positive example, we cannot obtain a query candidate by strengthening the query. Third, a desired query tends to constrain grammatical constructs mentioned in the natural language description sufficiently. For the instance in Figure 1, the query extracting the methods with the return type CacheConfig is a query candidate but not a desired one, as it does not pose any restriction on parameters.

Based on the observations, we realize that it is possible to narrow down necessary relations and avoid their infeasible compositions to prune the search space, and meanwhile, select query candidates with the guidance of the natural language description. According to the insight, we present a multi-modal synthesis algorithm Squid with three stages:

- To narrow down the relations, we introduce the notion of the *dummy relations* to depict the relations unnecessary for the synthesis and propose the *representation reduction* to exclude dummy relations, which prunes the search space effectively.
- To avoid infeasible compositions of relations, we perform the *bounded refinement* to enumerate the queries, skipping the unnecessary search for the queries that exclude a positive example. Particularly, the string constraints are synthesized by computing the longest common substrings, which is achieved efficiently in the refinement.
- To select desired queries, we establish the dual quantitative metrics, namely *named entity coverage* and *structural complexity*, and select query candidates by optimizing them as the objectives, which creates more opportunities for returning desired queries.

We implement Squid and evaluate it on 31 code search tasks. It successfully synthesizes desired queries for each task in 2.56 seconds on average. Besides, the representation reduction and the bounded refinement are crucial to its efficiency. Skipping either of them would increase the average time cost to around 8 seconds, and several tasks cannot be finished

within one minute. Meanwhile, dual quantitative metrics play critical roles in the selection. Applying only one metric would make 12 or 7 tasks fail due to the synthesized non-desired queries. We also state and prove the soundness, completeness, and optimality of SQUID. If there exist query candidates for a given synthesis specification, SQUID always returns query candidates optimizing two proposed metrics to express the search intent. To summarize, our work makes the following key contributions:

- We propose a multi-modal conjunctive query synthesis problem. An effective and efficient solution can serve as a user-friendly interface of a Datalog-based analyzer for code search.
- We design an efficient algorithm SQUID for an instance of our synthesis problem, which automates the code search tasks in real-world scenarios.
- We implement SQUID as a tool and evaluate it upon 31 code search tasks, showing that SQUID synthesizes the desired queries successfully and efficiently.

2 Overview

This section demonstrates a motivating example and briefs the key idea of our approach.

2.1 Motivating Example

Suppose a developer wants to avoid the security issue caused by log4j library [41]. He or she may examine the methods that receive a Log4jUtils object as a parameter and return a CacheConfig object. One choice is to leverage the built-in search tools of the IDEs to search the code lines containing Log4jUtils or CacheConfig, while the string matching-based search cannot filter grammatical constructs according to their kinds. Although several IDEs enable the structural search [23], their non-extensible templates only support searching for grammatical constructs of restrictive kinds. Another alternative is to write a query depicting the target pattern and evaluate it with a Datalog-based program analyzer, such as CODEQL [3]. However, it not only involves great laborious efforts in query language learning but also creates the burden of query writing and debugging.

To improve the usability and capability of code search, we aim to synthesize a query for a Datalog-based program analyzer. As shown in Figure 1(a), a user can specify the synthesis specification to indicate the search intent. Specifically, the positive and negative examples show the desired and undesired grammatical constructs, respectively, while the natural language description demonstrates the search intent in a sentence. Based on a Datalog-based program analyzer, we can convert the examples to a set of relations as the relational representation along with positive and negative tuples, which are shown in Figure 1(b). For example, the first tuple in the relation Method is the positive tuple indicating the method foo in Figure 1(a), which is a positive example. If we automatically synthesize the conjunctive query in Figure 1(c), the user does not need to delve into the relations and, instead specifies the synthesis specification from a high-level perspective.

2.2 Synthesizing Conjunctive Queries

The query synthesizer should effectively generate the desired queries that express the search intent correctly. However, it is stunningly challenging to obtain an effective and efficient synthesizer. First, we have to tackle a great number of the relations and their attributes when we choose relevant relations and enforce correct constraints upon them, which can involve both comparisons over attributes and string constraints. Second, the non-uniqueness of query

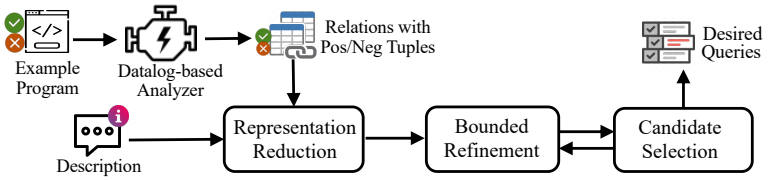


Figure 2 The overview of SQUID

candidates creates the obstacle of selecting proper candidates. Any improper selection would return a non-desired query, leading to code search failure. To address the challenges, we propose a new multi-modal synthesis algorithm SQUID. As shown in Figure 2, SQUID consists of three phases, which come from the following three ideas.

Idea 1: Removing dummy relations. Although there are many relations potentially used in the synthesis, we can identify a class of relations, named *dummy relations*, as unnecessary ones and then discard them safely. Specifically, a relation is dummy if it cannot separate a positive tuple from a negative one. As an example, the methods in Figure 1(a) have the same modifier, which is shown by the same values of the foreign keys mdf_id of the relation Method in Figure 1(b). This indicates that the relation Modifier has no impact on excluding negative tuples and thus can be discarded to prune the search space. Based on this insight, we propose the **representation reduction** to remove the dummy relations, narrowing down the necessary relations for the synthesis.

Idea 2: Enumerating query candidates via refinement. According to the query syntax, the constraints, including attribute comparisons and string constraints, pose restrictions on grammatical constructs. This implies that we cannot obtain a query candidate by refining the query that excludes a positive tuple. In Figure 1, we may obtain a query that enforces both the parameter and the return value of a method have the same type. Obviously, the query excludes the method foo, which is a positive tuple, and thus, we should stop strengthening the restrictions on grammatical constructs. Based on this insight, we introduce the technique of the **bounded refinement**, which adds conditions for the query enumeration and discards the queries that exclude any positive tuples. Thus, we can avoid enumerating infeasible compositions of relations, which further prunes the search space effectively.

Idea 3: Dual quantitative metrics for selection. Desired queries not only separate positive tuples from negative ones but also tend to cover as many program-related named entities as possible. In Figure 1, we may obtain a query candidate that restricts the return type to be a CacheConfig object. However, it does not pose any restriction on the parameters and leaves the named entity "parameter" uncovered, showing that it does not express the intent sufficiently. Meanwhile, Occam's razor [5] implies that desired queries should be as simple as possible. Hence, we introduce the *named entity coverage* and the *structural complexity* as the dual quantitative metrics, and perform the **candidate selection** to identify desired queries by optimizing the metrics. Finally, we blend the selection with the refinement and terminate the enumeration when unexplored candidates cannot be better than the current ones, further avoiding the unnecessary enumerative search.

3 Problem Formulation

This section first presents the program relational representation (§ 3.1) and then introduces the conjunctive queries for code search (§ 3.2). Lastly, we state the multi-modal conjunctive query synthesis problem and brief the roadmap of technical sections (§ 3.3).

3.1 Program Relational Representation

First of all, we formally define the concept of the *relation* as the preliminary.

▶ **Definition 3.1.** (Relation) A relation $R(a_1, \cdots, a_n)$ is a set of tuples $(t_1, \cdots, t_n)$, where n is the arity of R. For each $1 \leq i \leq n$, a_i is the attribute of the relation.

A relation is structured data that stores the details of different aspects of an entity. Concretely, a Datalog-based program analyzer encodes the program properties with a set of relations in a specific schema [3, 34, 37]. In what follows, we define the *relational representation* of a program.

▶ **Definition 3.2.** (Relational Representation) Given a program, its relational representation $\mathcal{R}$ is a set of relations over the following schema Γ. Specifically, Γ maps a relation symbol R to an n-tuple of pairs, where each element in the tuple $\Gamma(R)$ has one of the following form:

- (id, R): The attribute id is the primary key of the relation R.
- (a, R'): The foreign key a in the relation R, referencing the primary key of $R' \in dom(\Gamma)$.
- (a, STR): The attribute a has a string value indicating the textual information.

Particularly, we say Γ as the language schema. Without introducing the ambiguity, we use $(a, \cdot) \in \Gamma(R)$ to indicate that $(a, \cdot)$ is an element of the tuple $\Gamma(R)$.

▶ **Example 3.1.** Figure 1(b) shows five relations as examples, where the values of the primary keys are italic and underlined, and the foreign keys are italic. Based on the first tuple in the relation Method, we can track the identifier, the return type, and the modifier of the method foo based on the foreign keys. Similarly, we can identify the identifier and the type of each parameter based on the relation Parameter.

Essentially, the relational representation of a program encodes the program properties with relations, which depicts the relationship of grammatical constructs in the program. In reality, various relations can be derived with Datalog-based program analyzers, such as ReferenceType and VarPointsTo provided by Doop [44], depicting the type information and points-to facts, respectively. Due to the space limit, we only show five relations in Figure 1(b).

3.2 Conjunctive Queries

To simplify the presentation, we formulate the conjunctive queries as the relational algebra expressions [1] in the rest of the paper. In what follows, we first brief several relational algebra operations and then introduce the conjunctive queries for code search.

▶ **Definition 3.3.** (Relational Algebra Operations) In a relational algebra, the selection, projection, Cartesian product, and rename operations are defined as follows:

- $\sigma_\Theta(R) := \{(t_1, \ldots, t_n) \in R \mid [a_i \mapsto t_i \mid 1 \leq i \leq n] \models \Theta\}$ is the selection of a relation R with the selection condition Θ over its attributes.
- $\Pi_{\mathbf{a}'}(R) := \{(\mathbf{t}.a'_1, \ldots, \mathbf{t}.a'_k) \mid \mathbf{t} \in R\}$ is the projection of a relation R upon a tuple of attributes $\mathbf{a}'$, denoted by $\Pi_{\mathbf{a}'}(R)$. Here $\mathbf{a}' = (a'_1, \ldots, a'_k)$. $\mathbf{t}.a$ is the value of the attribute a in the tuple $\mathbf{t}$.
- $R_1 \times R_2 := \{(t^1_1, \ldots, t^1_{n_1}, t^2_1, \ldots, t^2_{n_2}) \mid (t^1_1, \ldots, t^1_{n_1}) \in R_1, (t^2_1, \ldots, t^2_{n_2}) \in R_2\}$ is the Cartesian product of two relations R_1 and R_2.
- The rename of a relation R, denoted by $\rho_A(R)$, yields the same relation named A.

The relational algebra operations enable us to manipulate the relational representation to search desired grammatical constructs. Specifically, we often need to search specific grammatical constructs via string match and enforce them to satisfy several constraints simultaneously. Now we formalize the conjunctive queries to express the code search intent.

▶ **Definition 3.4.** (Conjunctive Query) Given the relational representation $\mathcal{R}$, a conjunctive query R_Q is a relational algebra expression of the form $\Pi_{(A_i.*)}(\sigma_\Theta(\rho_{A_1}(R_1) \times \ldots \times \rho_{A_m}(R_m)))$, where $R_i \in \mathcal{R}$, $\Theta := \phi_1 \wedge \cdots \wedge \phi_n$, and each $\phi_i (1 \leq i \leq n)$ occurring in the selection condition Θ is an atomic condition in the following two forms:

- An atomic equality formula $A_j.\mathsf{id} = A_k.a$, where a is the foreign key and $(a, R_j) \in \Gamma(R_k)$.
- A string constraint $p(A_k.a, \ell)$ over the string attribute $A_k.a$, where ℓ is a string literal, and $p \in \{\mathsf{equal}, \mathsf{suffix}, \mathsf{prefix}, \mathsf{contain}\}$.

In particular, $\Pi_{(A_i.*)}$ indicates the projection upon all the attributes of the relation A_i.

The form of the conjunctive queries in Definition 3.4 depicts the user intent from two aspects. First, the atomic equality formulas encode the relationship between the grammatical constructs. Second, the four string predicates support the common scenarios of string matching-based code search. We do not focus on synthesizing more expressive string constraints, which is the orthogonal direction of program synthesis [10, 27, 36]. Based on the conjunctive queries, we can simultaneously perform the string matching-based search and filter the constructs with various relations in the program relational representation.

▶ **Example 3.2.** We can formalize the query in Figure 1(c) as the relational algebra expression:

$$\Pi_{(A_1.*)}(\sigma_\Theta(\rho_{A_1}(\mathsf{Method}) \times \rho_{A_2}(\mathsf{Type}) \times \rho_{A_3}(\mathsf{Parameter}) \times \rho_{A_4}(\mathsf{Type})))$$

where the selection condition Θ in the selection operation is as follows:

$$\begin{aligned}\Theta :=& (A_1.\mathsf{id} = A_3.\mathsf{method\_id}) \wedge (A_1.\mathsf{ret\_type\_id} = A_2.\mathsf{id}) \wedge (A_3.\mathsf{type\_id} = A_4.\mathsf{id}) \wedge \\ & \mathsf{equal}(A_2.\mathsf{name}, \text{“CacheConfig”}) \wedge \mathsf{equal}(A_4.\mathsf{name}, \text{“Log4jUtils”})\end{aligned}$$

The conjunctive queries can be instantiated with various flavors [17, 1, 7]. The *select-from-where queries* are the instantiations of the conjunctive queries in SQL. Besides, a simple Datalog program can also express the conjunctive query with a single Datalog rule. In our paper, we formulate a conjunctive query as a relational algebra expression. Our implementation actually synthesize the conjunctive queries as Datalog programs, which are evaluated by a Datalog solver over the program relational representation for code search.

3.3 Multi-modal Conjunctive Query Synthesis Problem

To generate the conjunctive query for code search, the users need to specify their intent as the specification. In our work, we follow the premise of the recent studies on the multi-modal program synthesis [10, 4] that multiple modalities of information can go arm in arm with each other, serving as the informative specification for the synthesis. Specifically, the users provide a natural language sentence to describe the target code pattern and provide several grammatical constructs as positive and negative examples. Notably, the multi-modal synthesis specification is easy to provide. When the users want to explore a specific code pattern, they can describe the pattern briefly in a natural language and provide several examples from their editors instead of delving into the details of the underlying relations and their attributes, enabling users to generate a query for code search in a declarative manner.

Based on the multi-modal synthesis specification, the positive and negative examples are converted to the tuples in a specific relation. For example, Figure 1 shows a set of relations as the relational representation of the examples. To formalize our problem better, we define the notion of the *relation partition* as follows.

▶ **Definition 3.5.** (Relation Partition) The relation partition of $R^* \in \mathcal{R}$ is a pair of two relations (R_p^*, R_n^*) satisfying $R^* = R_p^* \cup R_n^*$ and $R_p^* \cap R_n^* = \emptyset$. The relation partition is non-trivial if and only if $R_p^* \neq \emptyset$ and $R_n^* \neq \emptyset$. We say the tuples in R_p^* and R_n^* are positive and negative tuples, respectively.

▶ **Example 3.3.** As shown in Figure 1, we can construct the relation partition (R_p^*, R_n^*), where $R_p^* = \{(\mathsf{M1},\ \mathsf{I1},\ \mathsf{T3},\ \mathsf{MDF1})\}$ and $R_n^* = \{(\mathsf{M2},\ \mathsf{I2},\ \mathsf{T3},\ \mathsf{MDF1}), (\mathsf{M3},\ \mathsf{I3},\ \mathsf{T2},\ \mathsf{MDF1})\}$. Obviously, R_p^* and R_n^* are disjoint, and $R_p^* \cup R_n^*$ is exactly the relation Method.

The positive and negative tuples essentially depict the positive and negative examples, respectively. Based on the program's relational representation, the examples specified by the users can determine the positive and negative tuples, which can be achieved in various manners. Specifically, users can select a grammatical construct in the IDEs or use a code sample in a specific coding standard [51] as a positive example and remove several sub-patterns from a positive example by mutation to construct negative ones. Such positive and negative examples further constitute a sample code snippet, from which a Datalog-based analyzer derives a set of relations as the relational representation. In this paper, we omit the details of positive/negative tuple generation and formulate a *multi-modal conjunctive query synthesis (MMCQS) problem* as follows.

> Given a relational representation $\mathcal{R}$, a relation partition (R_p^*, R_n^*) of $R^* \in \mathcal{R}$, and a natural language description s, we aim to synthesize a conjunctive query R_Q containing the positive tuples in R_p^* and excluding the negative tuples in R_n^*.

▶ **Example 3.4.** Figure 1(a) shows the multi-modal synthesis specification, which consists of a positive example, two negative examples, and a natural language description s as "Find all the methods receiving a Log4jUtils-type parameter and giving a CacheConfig-type return". Leveraging a Datalog-based analyzer, we obtain the relational representation and the relation partition of Method, which are shown in Figure 1(b). To automate the code search, we expect to synthesize the query in Fig 1(c) or Example 3.2 according to the relational representation, the relation partition, and the natural language sentence.

To promote the code search, we propose an efficient synthesis algorithm Squid for the MMCQS problem, which is our main technical contribution. As explained in § 1, it is challenging to solve the MMCQS problem efficiently, which involves mitigating the huge search space and selecting queries from multiple candidates. In the following two sections, we formalize the conjunctive query synthesis from a graph perspective (§ 4), and illustrate the technical details of our synthesis algorithm Squid (§ 5), which prunes the search space and selects desired queries effectively.

4 Conjunctive Query Synthesis: A Graph Perspective

This section presents a graph perspective of our conjunctive query synthesis problem. Specifically, we introduce two graph representations of the language schema and the conjunctive queries, named the schema graph (§ 4.1) and the query graph (§ 4.2), respectively, which reduces the conjunctive query synthesis to the query graph enumeration. Lastly, we summarize the section and highlight the technical challenges from a graph perspective (§ 4.3).

4.1 Schema Graph

According to Definition 3.2, a relation in the language schema has three kinds of attributes, namely a unique primary key, foreign keys, and string attributes. Obviously, the selection condition Θ in R_Q should only compare the foreign key of a relation with its referenced primary key or constrain the string attributes with string predicates. To depict the possible ways of constraining the attributes, we define the concept of the *schema graph* as follows.

▶ **Definition 4.1.** (Schema Graph) The schema graph G_Γ of a language schema Γ is (N_Γ, E_Γ):

- The set N_Γ contains the relation symbols in the schema or the string type STR as the nodes of the schema graph, i.e., $N_\Gamma := dom(\Gamma) \cup \{\mathsf{STR}\}$.
- The set E_Γ contains an edge (n_1, n_2, a) if and only if either of the conditions holds:
  - $n_1, n_2 \in dom(\Gamma)$ and $(a, n_2) \in \Gamma(n_1)$: The relation n_1 has a foreign key named a referencing the primary key of the relation n_2.
  - $n_1 \in dom(\Gamma)$ and $(a, \mathsf{STR}) \in \Gamma(n_1)$: a is the string attribute of the relation n_1.

▶ **Example 4.1.** Consider the relations in the relational representation shown in Figure 1(b). We can construct the schema graph in Figure 3(a). The edge from Method to Modifier labeled with mdf_id shows that the attribute mdf_id of Method is a foreign key referencing the primary key of Modifier. Similarly, the edge from Type to STR labeled with name shows that the attribute name in Type is a string attribute.

Noting that there can exist multiple edges with different labels between two nodes in the schema graph, which indicate that a relation take multiple foreign keys referencing the same relation or string attributes as the attributes. Essentially, the schema graph encodes the available relations with its node set and depicts the valid forms of the atomic formulas appearing in the selection condition with its edge set. Although we can compare the attributes of any relations flexibly, a solution to our problem must take the valid form of the atomic formulas as its selection condition, comparing the foreign keys with the referenced primary keys or examining the string attributes of the relations appearing in a Cartesian product.

4.2 Query Graph

As formulated in Definition 3.4, there are two key components in the conjunctive query, namely the Cartesian product and the selection condition. Leveraging the schema graph, we can represent the components with nodes and edges on the graph, which uniquely determines a conjunctive query. Formally, we introduce the notion of the *query graph* as follows.

▶ **Definition 4.2.** (Query Graph) Given a conjunctive query Q, its query graph G_Q is (N_Q, E_Q, Φ_Q):

- The set N_Q contains (R_i, A_i) or (p, ℓ) as a node in the query graph. $R_i \in \mathcal{R}$ is a relation and A_i is the unique relation identifier. p and ℓ are the string predicate and literal, respectively.
- The set E_Q contains (n_1, n_2, a) as an edge, corresponding to the equality atomic formula $A_j.a = A_k.\mathsf{id}$ in the selection condition, where $n_1 = (R_j, A_j)$ and $n_2 = (R_k, A_k)$.
- The mapping Φ_Q maps a 3-tuple (R_j, A_j, a) to a node (p, ℓ), indicating the edge from (R_j, A_j) to (p, ℓ) with the label a, which corresponds to the string constraint $p(A_j.a, \ell)$.

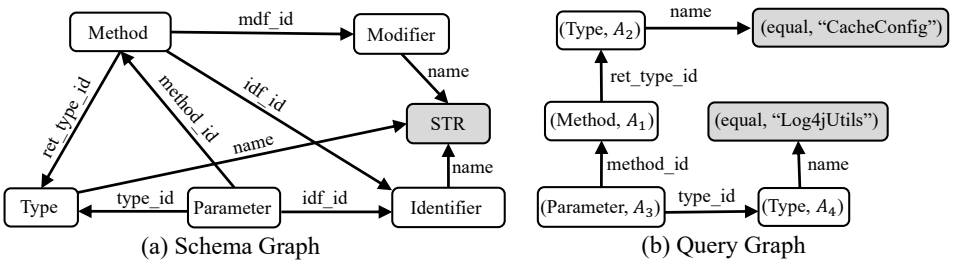


Figure 3 The examples of the schema graph and the query graph

▶ **Example 4.2.** Figure 3(b) shows an query graph of R_Q in Example 3.2. The white nodes show the four relations appearing in the Cartesian product, while the gray nodes indicate the string predicates and literals in the selection condition. The edges depict two kinds of atomic conditions. For example, the edge from $(\mathsf{Parameter}, A_3)$ to (Method, A_1) labeled with $\mathsf{method\_id}$ indicates the equality constraint $A_3.\mathsf{method\_id} = A_1.\mathsf{id}$. Meanwhile, the edge induced by $\Phi_Q(\mathsf{Type},\ A_4,\ \mathsf{name}) = (\mathsf{equal}, \text{“}\mathsf{Log4jUtils}\text{”})$ indicates the string constraint $\mathsf{equal}(A_4.\mathsf{name}, \text{“}\mathsf{Log4jUtils}\text{”})$.

Essentially, a conjunctive query and a query graph are allotropes. That is, there exists a bijection κ mapping a conjunctive query R_Q to a query graph G_Q such that $G_Q = \kappa(R_Q)$ and $R_Q = \kappa^{-1}(G_Q)$. Thus, we can enumerate the conjunctive queries by enumerating the query graphs. Besides, the schema graph restricts the form of the selection condition over the attributes. If an edge with the label a connects (R_j, A_j) and (R_k, A_k) in a query graph, there should exist an edge labeled with a connecting the relations R_j and R_k in the schema graph. A similar argument also holds for the edge of the query graph indicated by the mapping Φ_Q. Therefore, we can reduce our conjunctive query synthesis to a search problem, of which the search space is characterized as the set of query graphs.

4.3 Summary

Leveraging the schema graph, we have reduced the conjunctive query synthesis process to query graph enumeration. To obtain the desired queries, we only need to select the nodes and edges from the schema graph and create the node (p, ℓ) with proper string predicates and literals for constructing a query graph G_Q, which should satisfy that the induced query $\kappa^{-1}(G_Q)$ separates positive tuples from negative ones.

Obtaining the desired queries with high efficiency is a non-trivial problem. The schema graph can be overwhelming, containing over a hundred nodes and edges, which leads to an enormous number of choices for relations occurring in the query graph. Even for a given set of relations, the flexibility of instantiating equality constraints and string constraints over their attributes can induce a large number of selection choices of edges in a query graph, which exacerbates the search space explosion problem. Additionally, the existence of multiple query candidates necessitates the effective selection of candidates, which is crucial for conducting a code search task. In the next section, we will detail our synthesis algorithm that addresses these challenges, resulting in high efficiency and effectiveness for code search.

5 Synthesis Algorithm

This section presents our synthesis algorithm Squid to solve the MMCQS problem. Squid takes as input the relational representation $\mathcal{R}$ of an example program, a relation partition

(R_p^*, R_n^*) of $R^* \in \mathcal{R}$, and a natural language description s. It generates the query candidates separating positive tuples $\mathbf{t}_p \in R_p^*$ from negative ones $\mathbf{t}_n \in R_n^*$, which can be further selected and then used for code search. To address the challenges in § 4.3, SQUID works with the following three stages:

- To tackle a large number of relations, SQUID relies on the notion of dummy relations and conducts the representation reduction based on the positive and negative tuples, which effectively narrow down the relations that can appear in the query graph (§ 5.1).
- To avoid unnecessary enumeration of edges in query graphs, we propose the bounded refinement by enumerating the query graphs based on the schema graph, which essentially appends equality constraints and string constraints inductively (§ 5.2).
- To select query candidates, SQUID identifies the named entities in the natural language description s for the prioritization, and blends the selection with the refinement to collect the desired queries (§ 5.3).

We also formulate the soundness, completeness, and optimality of our algorithm (§ 5.4). For better illustration, we use the synthesis instance shown in Figure 1 throughout this section.

5.1 Representation Reduction

To tackle the large search space, we first propose the representation reduction to narrow down the relations possibly used in the query. Specifically, we introduce the notion of the dummy relations to determine the characteristics of unnecessary relations (§ 5.1.1) and propose the algorithm of removing dummy relations for the representation reduction (§ 5.1.2).

5.1.1 Dummy Relations

There exists a class of relations, named *dummy relations*, which cannot involve in distinguishing positive and negative tuples, such as the relation Modifier in Figure 1. Before defining them, we first introduce the *undirected relation path* and the *activated relation*.

▶ **Definition 5.1.** (Undirected Relation Path) An undirected relation path from R_0 to R_{k+1} in the schema graph $G_\Gamma = (N_\Gamma, E_\Gamma)$ is $p : R_0 \hookrightarrow_{(a_0,d_0)} \cdots \hookrightarrow_{(a_k,d_k)} R_{k+1}$, where $R_i \in dom(\Gamma)$. Here, $d_i = 1$ if and only if $(R_i, R_{i+1}, a_i) \in E_\Gamma$, and $d_i = -1$ if and only if $(R_{i+1}, R_i, a_i) \in E_\Gamma$.

▶ **Definition 5.2.** (Activated Relation) Given a tuple $\mathbf{t}_0 \in R_0$ and an undirected relation path $p : R_0 \hookrightarrow_{(a_0,d_0)} \cdots \hookrightarrow_{(a_k,d_k)} R_{k+1}$, the activated relation of $\mathbf{t}_0$ along p is

$$\mathcal{I}(\mathbf{t}_0, p) = \{\mathbf{t}_{k+1} \mid \mathbf{t}_{i+1} \in R_{i+1},\ \mathsf{ite}(d_i = 1,\ \mathbf{t}_i.a_i = \mathbf{t}_{i+1}.\mathsf{id},\ \mathbf{t}_i.id = \mathbf{t}_{i+1}.a_i),\ 0 \le i \le k\}$$

▶ **Example 5.1.** In in Figure 1, the path p_1 : Method $\hookrightarrow_{(\mathsf{method\_id},-1)}$ Parameter $\hookrightarrow_{(\mathsf{type\_id},\ 1)}$ Type, and $\mathbf{t}_p = (\mathsf{M1},\ \mathsf{I1},\ \mathsf{T3},\ \mathsf{MDF1}) \in R_p^*$. We have $\mathbf{t}_1 = (\mathsf{P1},\ \mathsf{I4},\ \mathsf{T1},\ \mathsf{M1}) \in$ Parameter, and $\mathbf{t}_2 = (\mathsf{T1}, \mathsf{Log4jUtils}) \in$ Type, By inspecting other tuples, we have $\mathcal{I}(\mathbf{t}_p, p_1) = \{(\mathsf{T1}, \mathsf{Log4jUtils})\}$.

Intuitively, an undirected relation path can depict the restriction upon the relations in the Cartesian product of the query. The activated relation actually contains the tuples enforcing the primary key of $\mathbf{t}_0$ to appear in a selected tuple. Therefore, it is possible to narrow down the relations for the synthesis by inspecting the activated relations of positive and negative tuples along each undirected relation path. According to the intuition, we formally introduce and define the notion of the *dummy relations* as follows.

▶ **Definition 5.3.** (Dummy Relation) Given a relation partition (R_p^*, R_n^*) of $R^* \in \mathcal{R}$, a relation $R \in \mathcal{R}$ is dummy if for every undirected relation path p from R^* to R, either of the conditions is satisfied: (1) There exists $\mathbf{t}_p \in R_p^*$ such that $\mathcal{I}(\mathbf{t}_p, p) = \emptyset$; (2) $\mathcal{I}(\mathbf{t}_p, p) = \mathcal{I}(\mathbf{t}_n, p)$ for any $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$.

Definition 5.3 formalizes two characteristics of relations unnecessary the synthesis. First, the empty activated relation of a positive tuple $\mathbf{t}_p$ indicates the absence of the tuples in the relation R, making the tuple $\mathbf{t}_p$ appear. Second, the relation R cannot contribute to separating positive tuples from negative ones if several tuples in R make all the positive and negative tuples appear simultaneously. Thus, such relations can be discarded safely.

▶ **Example 5.2.** Example 5.1 shows that $\mathcal{I}(\mathbf{t}_p, p_1) \neq \emptyset$. Similarly, $\mathcal{I}(\mathbf{t}_n, p_1) = \{(\mathsf{T2}, \mathsf{int})\} \neq \mathcal{I}(\mathbf{t}_p, p_1)$ when $\mathbf{t}_n = (\mathsf{M2},\ \mathsf{I2},\ \mathsf{T4},\ \mathsf{MDF1}) \in R_n^*$. Hence, Type is not dummy. Besides, any undirected relation path p_2 from Method to Modifier has the form

$$\mathsf{Method}[\hookrightarrow_{(\mathsf{mdf\_id},+1)} \mathsf{Modifier} \hookrightarrow_{(\mathsf{mdf\_id},-1)} \mathsf{Method}]^* \hookrightarrow_{(\mathsf{mdf\_id},+1)} \mathsf{Modifier}$$

where $[\cdot]^*$ indicates the repeated subpath. We find that $\mathcal{I}(\mathbf{t}_p, p_2) = \mathcal{I}(\mathbf{t}_n, p_2) = \{\mathsf{MDF1}, \mathsf{public}\}$ for any $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$. Thus, the relation Modifier is a dummy relation.

5.1.2 Removing Dummy Relations

Based on Definition 5.3, identifying dummy relations involves two technical parts. First, we should collect all the undirected relation paths from R^* to each relation. Second, we need to compute $\mathcal{I}(\mathbf{t}_p, p)$ and $\mathcal{I}(\mathbf{t}_n, p)$ for each undirected relation path p and positive/negative tuple. However, the schema graph can contain a large and even infinite number of undirected relation paths from R^*. Any cycle induces the infinity of the path number, making it tricky to examine the conditions in Definition 5.3. Fortunately, we realize that the number of cycles in a path does not affect the activated relation, which is stated in the following property.

▶ **Property 5.1.** Given any $\mathbf{t}_0 \in R_0$, we have $\mathcal{I}(\mathbf{t}_0, p) = \mathcal{I}(\mathbf{t}_0, p')$ for p and p' as follows:

$$p : R_0 \hookrightarrow_{(a_0,d_0)} \cdots R_l\ [\hookrightarrow_{(a'_l,d'_l)} \cdots R_{l+t} \hookrightarrow_{(a'_{l+t},d'_{l+t})} R_l]^+ \hookrightarrow_{(a_l,d_l)} \cdots R_{\mathsf{k}} \hookrightarrow_{(a_k,d_k)} R_{\mathsf{k+1}}$$
$$p' : R_0 \hookrightarrow_{(a_0,d_0)} \cdots R_l \hookrightarrow_{(a'_l,d'_l)} \cdots R_{l+t} \hookrightarrow_{(a'_{l+t},d'_{l+t})} R_l \hookrightarrow_{(a_l,d_l)} \cdots R_{\mathsf{k}} \hookrightarrow_{(a_k,d_k)} R_{\mathsf{k+1}}$$

Here, $[\cdot]^+$ indicates the cycle occurring at least one time.

Property 5.1 holds trivially according to Definition 5.2. For each undirected relation path p containing a cycle, the constraints over the tuples in $\mathcal{I}(\mathbf{t}_0, p)$ are the same as the ones over the tuples in $\mathcal{I}(\mathbf{t}_0, p')$. Thus, we can just examine the finite number of undirected relation paths in which a cycle appears at most one time.

Establish upon the above concepts and property, Algorithm 1 shows the details of the representation reduction by removing dummy relations. Initially, we construct the schema graph G_Γ according to Definition 4.1 (line 3). Then we compute the undirected relation paths from R^* to R in G_Γ, where any cycle repeats at most once (lines 4–5). Specifically, the function AcyclicPath collects all the acyclic undirected relation paths from R^* to R in the schema graph G_Γ, while the function augmentPathWithCycle augments each acyclic path by appending each cycle at most one time. For each undirected relation path p, we compute $\mathcal{I}(\mathbf{t}_p, p)$ and $\mathcal{I}(\mathbf{t}_n, p)$ according to Definition 5.2 for each $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$, respectively. Therefore, we can identify R as a non-dummy relation if both the conditions in Definition 5.3 are violated (lines 6–11). Finally, we obtain the reduced relational representation $\mathcal{R}'$ that excludes all the dummy relations (line 12).

Algorithm 1 Removing dummy relations for representation reduction

```
1  Procedure reduce(Γ, R, R*_p, R*_n):
2      R' ← ∅;
3      G_Γ ← SchemaGraph(Γ) ;
4      foreach R ∈ R :
5          P ← augmentPathWithCycle(AcyclicPath(R*, R, G_Γ));
6          foreach p ∈ P, t_p ∈ R*_p, t_n ∈ R*_n :
7              if I(t_p, p) ≠ I(t_n, p) :
8                  R' ← R' ∪ {R}; break ;
9          foreach p ∈ P, t_p ∈ R*_p :
10             if I(t_p, p) = ∅ :
11                 R' ← R' \ {R}; break ;
12     return R';
```

▶ **Example 5.3.** Consider the undirected relation paths from Method to Modifier in Figure 3(a). Algorithm 1 collects the acyclic path p_3 : Method $\hookrightarrow_{(\text{mdf\_id},+1)}$ Modifier and augments it to form the path p_4 : Method $\hookrightarrow_{(\text{mdf\_id},+1)}$ Modifier $\hookrightarrow_{(\text{mdf\_id},-1)}$ Method $\hookrightarrow_{(\text{mdf\_id},+1)}$ Modifier. Based on the activated relations $\mathcal{I}(\mathbf{t}, p_3)$ and $\mathcal{I}(\mathbf{t}, p_4)$ for each tuple $\mathbf{t}$ in Method, we can find that $\mathcal{I}(\mathbf{t}_p, p) = \mathcal{I}(\mathbf{t}_n, p)$ for every $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$. Thus, Modifier is a dummy relation.

Essentially, our representation reduction analyzes the example program upon its relational representation. The activated relations provide sufficient clues to identifying unnecessary relations, i.e., the dummy ones. As the first step of the synthesis, the representation reduction narrows down the relations used in the conjunctive query. Furthermore, in the enumeration of the edges of a query graph, i.e., the sets E_Q and Φ_Q, we only need to focus on the attributes in the non-dummy relations in the reduced relational representation, which prunes the search space significantly. Lastly, we formulate the soundness of the representation reduction as follows, which can further ensure the completeness of our synthesis algorithm in § 5.4. We provide a detailed proof in [49].

▶ **Theorem 5.1.** (Soundness of Representation Reduction) If an instance of the MMCQS problem has a solution, there must be a conjunctive query R_Q, of which the Cartesian product only consists of non-dummy relations, such that R_Q is also a solution.

Proof. Assume that there does not exist a solution, only manipulating non-dummy relations. The assumption implies that there is a conjunctive query R'_Q satisfying the following conditions:

- R'_Q is the solution of the MMCQS problem instance;
- A dummy relation R appears in the Cartesian product of R'_Q.

Denote $R'_Q = \Pi_{(A_1.*)}(\sigma_{\Theta'}(\rho_{A_1}(R_1) \times \ldots \times \rho_{A_{m'}}(R_{m'})))$, where $\Theta' = \phi_e^{(1)} \wedge \cdots \wedge \phi_e^{(n'_e)} \wedge \phi_s^{(1)} \wedge \cdots \wedge \phi_s^{(n'_s)}$. Without the loss of generality, we assume that the dummy relation R appears in the last few renaming expressions in the selection condition and equality/string constraints, i.e., $R_{m+1} = R_{m+2} = \cdots = R_{m'} = R$, and $\phi_e^{(i)}$ and $\phi_s^{(j)}$ constrain the attributes of R, where $n_e < i \leq n'_e$ and $n_s < j \leq n'_s$. Then we can construct the query candidate R_Q by removing all the occurrences of the relation R from the Cartesian product and the selection condition. That is, $R_Q = \Pi_{(A_1.*)}(\sigma_{\Theta}(\rho_{A_1}(R_1) \times \ldots \times \rho_{A_m}(R_m)))$, where $\Theta = \phi_e^{(1)} \wedge \cdots \wedge \phi_e^{(n_e)} \wedge \phi_s^{(1)} \wedge \cdots \wedge \phi_s^{(n_s)}$. As long as we prove that R_Q is also a solution, we can repeat the process of eliminating dummy relations iteratively until the query does not contain any dummy relations, which

conflicts with our assumption at the beginning. Hence, we only need to prove that the above conjunctive query R_Q is a solution, which finally proves the theorem.

- First, the selection condition Θ in the conjunctive query R_Q is weaker than the selection condition Θ', as Θ excludes several atomic formulas from Θ'.
- Second, R_1 is renamed to A_1 and cannot be a dummy relation, which means that R_1 still appears in the Cartesian product of R_Q.
- Third, if the eliminated equality constraints $\phi_e^{(i)}(n_e < i \leq n'_e)$ do not induce any undirected relation path from R_1 to R, the selection conditions Θ and Θ' pose the same restrictions on the tuples in R_1, indicating that $R_Q = R'_Q$.
- Fourth, if the eliminated equality constraints $\phi_e^{(i)}(n_e < i \leq n'_e)$ induce a undirected relation path p from R_1 to R, we need to discuss two cases as follows:
  - Case 1: $\mathcal{I}(\mathbf{t}_p, p) = \emptyset$ for a specific tuple $\mathbf{t}_p \in R_p^* \subset R_1 = R^*$. According to Definition 3.4, we can obtain that $\mathbf{t}_p \notin \Pi_{(A_1.*)}\sigma_{\Theta'}(\rho_{A_1}(R_1) \times \ldots \times \rho_{A_{m'}}(R_{m'}))$ Thus, R'_Q cannot contain $\mathbf{t}_p$, which conflicts with the fact that R'_Q is a solution of the problem instance.
  - Case 2: $\mathcal{I}(\mathbf{t}_p, p) = \mathcal{I}(\mathbf{t}_n, p)$ for each $\mathbf{t}_p \in R_p^* \subset R_1 = R^*$ and $\mathbf{t}_n \in R_n^* \subset R_1 = R^*$. In this case, no matter how $\phi_e^{(i)}(n_e < i \leq n'_e)$ constrain the tuples in R^*, i.e., A_1 or R_1, all the positive and negative tuples in R^* belong to $\Pi_{(A_1.*)}\sigma_{\Theta'}(\rho_{A_1}(R_1) \times \ldots \times \rho_{A_{m'}}(R_{m'}))$ simultaneously. This also conflicts with the fact that R'_Q is a solution to the problem instance.

Therefore, we can always construct a conjunctive query with non-dummy relations if the problem instance has a solution. ◀

5.2 Bounded Refinement

Based on the reduced relational representation $\mathcal{R}'$, we can enumerate query candidates by selecting proper nodes corresponding to non-dummy relations in $\mathcal{R}'$ and edges connecting such nodes. However, the search space is potentially unbounded. The relations can occur in a query multiple times, i.e., a node in the schema graph can be selected more than one time. Meanwhile, the literal in a string constraint can be instantiated flexibly, which increases the difficulty of enumerating a query graph with proper instantiation of Φ_Q. To achieve high efficiency, we propose the bounded refinement to expand query graphs on demand and strengthen the query with the strongest string constraints. Specifically, we first introduce the notions of the bounded query and the refinable query (§ 5.2.1), and then present the details of enumerating the query graphs (§ 5.2.2).

5.2.1 Bounded Query and Refinable Query

As shown in Example 3.2, a relation can appear multiple times in the Cartesian product of a conjunctive query, inducing an unbounded search space in the synthesis. The unboundedness of the search space poses the great challenge of enumerating the query candidates efficiently. However, we realize that the conjunctive query for a code search task often involves only a few relations, each of which appears quite a few times. Thus, it is feasible to bound the maximal multiplicity of the relation in the query and conduct the bounded enumeration. Formally, we introduce the notion of the *(m,k)-bounded query* as follows.

▶ **Definition 5.4.** ((m, k)-Bounded Query) An (m, k)-bounded query is a conjunctive query with m relations such that (1) each relation appears at most k times; (2) there is a relation appearing exactly k times.

▶ **Example 5.4.** The conjunctive query $\Pi_{(A_1.*)}(\sigma_{\mathbf{true}}(\rho_{A_1}(\mathsf{Method})))$ is a $(1,1)$-bounded query. Similarly, the conjunctive query in Example 3.2 is a $(4,2)$-bounded query.

Intuitively, we can enumerate the (m,k)-bounded queries by selecting non-dummy relations at most k times, forming a query graph with m nodes. When constructing the sets E_Q and Φ_Q for the query graph enumeration, we only need to concentration on the attributes of the selected relations. However, not all the query graphs are worth enumerating. If R_Q excludes a positive tuple, there is no need to add more nodes and edges to its query graph, as it would induce a stronger selection condition, making the new query still exclude the positive tuple. Formally, we formulate the notion of the *refinable query* as follows.

▶ **Definition 5.5.** (Refinable Query) A conjunctive query R_Q is a refinable query if and only if for any $\mathbf{t}_p \in R_p^*$ we have $\mathbf{t}_p \in R_Q$, i.e., $R_p^* \subseteq R_Q$.

▶ **Example 5.5.** Consider $R_Q := \Pi_{(A_1.*)}(\sigma_\Theta(\rho_{A_1}(\mathsf{Method}) \times \rho_{A_2}(\mathsf{Type}) \times \rho_{A_3}(\mathsf{Parameter})))$, where the selection condition Θ in the selection operation is as follows:

$$\Theta := (A_1.\mathsf{id} = A_3.\mathsf{method\_id}) \wedge (A_1.\mathsf{ret\_type\_id} = A_2.\mathsf{id}) \wedge \mathsf{equal}(A_2.\mathsf{name}, \text{“CacheConfig”})$$

It is a refinable query as $R_p^* \subseteq R_Q = \{(\mathsf{M1},\ \mathsf{I1},\ \mathsf{T3},\ \mathsf{MDF1}), (\mathsf{M2},\ \mathsf{I2},\ \mathsf{T3},\ \mathsf{MDF1})\}$.

Essentially, a refinable query is the over-approximation of the positive tuples. When it excludes all the negative tuples, the query is exactly a query candidate. Thus, we can collect the query candidates by refining the refinable queries in the bounded enumeration.

5.2.2 Enumerating Query Candidates via Refinement

We denote the sets of (m,k)-bounded refinable queries and query candidates by $\mathcal{S}_R(m,k)$ and $\mathcal{S}_C(m,k)$, respectively, and set a multiplicity bound K to bound the multiplicity of a relation. To conduct a bounded enumeration, we have to compute the set $\mathcal{S}_C(m,k)$ for $k \leq K$, which can be achieved by examining whether the queries in $\mathcal{S}_R(m,k)$ are query candidates. Obviously, exhaustive enumeration is impossible as the search space of (m,k)-bounded queries is exponential to m and k. To avoid unnecessary enumeration, we leverage the structure of $\mathcal{S}_C(m,k)$, which is formulated in the following property.

▶ **Property 5.2.** For every refinable query $R_Q \in \mathcal{S}_C(m,k)$ and $\kappa(R_Q) := (N_Q, E_Q, \Phi_Q)$, there exists a query graph G_Q^1 or G_Q^2 such that

- $N_Q = N_Q^1 \cup \{(R, A_i)\}$, $E_Q^1 \subseteq E_Q$, and $\Phi_Q^1 \subseteq \Phi_Q$, where R appears in G_Q^1 exactly $(k-1)$ times. Here, $G_Q^1 = (N_Q^1, E_Q^1, \Phi_Q^1)$ and $\kappa^{-1}(G_Q^1) \in \mathcal{S}_R(m-1, k-1)$.
- $N_Q = N_Q^2 \cup \{(R, A_i)\}$, $E_Q^2 \subseteq E_Q$, and $\Phi_Q^2 \subseteq \Phi_Q$, where R appears in G_Q^2 fewer than k times. Here, $G_Q^2 = (N_Q^2, E_Q^2, \Phi_Q^2)$ and $\kappa^{-1}(G_Q^2) \in \mathcal{S}_R(m-1, k)$.

Property 5.2 shows that the sets of the nodes and edges in a query graph of a refinable query are subsumed by the ones of a refinable query with fewer relations, which permit us to enumerate the query candidates by computing $\mathcal{S}_R(m,k)$ and $\mathcal{S}_C(m,k)$ inductively. Technically, we achieve the enumerative search via the bounded refinement. Assuming that we have $\mathcal{S}_R(m',k')$ and $\mathcal{S}_C(m',k')$ for all $m' < m$ and $k' \leq k$. Algorithm 2 computes the sets $\mathcal{S}_R(m,k)$ and $\mathcal{S}_C(m,k)$. The technical details of the refinement are as follows:

- For the base case, where $m = 1$ and $k = 1$, we construct an empty query graph (line 3). For a general case, we merge the sets of the query graphs induced by the refinable queries in $\mathcal{S}_R(m-1,k)$ and $\mathcal{S}_R(m-1,k-1)$ (lines 4–7). Hence, we obtain a set of query graphs W to maintain all the refinable queries with m relations.

Algorithm 2 Enumerating query candidates via refinement

```
1  Procedure refine(S_R, S_C, R*_p, R*_n, m, k, R'):
2      if m = 1 and k = 1 :
3          W ← {(∅, ∅, ⊥)} ;
4      if m > 1 :
5          W ← {κ(R_Q) | R_Q ∈ S_R(m − 1, k)};
6          if k > 1 :
7              W ← W ∪ {κ(R_Q) | R_Q ∈ S_R(m − 1, k − 1)};
8
9      S_R(m, k) ← ∅;
10     while W is not empty do
11         G_Q ← pop(W); V ← ∅;
12         foreach R ∈ R' :
13             if multiplicity(G_Q, R) < k and κ^{-1}(G_Q) ∈ S_R(m − 1, k) :
14                 V ← V ∪ expand(G_Q, R);
15             if multiplicity(G_Q, R) = (k − 1) and κ^{-1}(G_Q) ∈ S_R(m − 1, k − 1) :
16                 V ← V ∪ expand(G_Q, R) ;
17
18         foreach G_Q : (N_Q, E_Q, Φ_Q) ∈ V and R*_p ⊆ κ^{-1}(G_Q) :
19             S_R(m, k) ← S_R(m, k) ∪ κ^{-1}(G_Q) ;
20             foreach (R_i, A_i) ∈ N_Q and (a, STR) ∈ Γ(R_i) :
21                 (p, ℓ) ← synLCS(G_Q, R_i, A_i, a, R*_p);
22                 G'_Q ← (N_Q, E_Q, Φ_Q[(R_i, A_i, a) ↦ (p, ℓ)]) ;
23                 S_R(m, k) ← S_R(m, k) ∪ κ^{-1}(G'_Q);
24
25     S_C(m, k) ← {R_Q | R_Q ∈ S_R(m, k), R*_p = R_Q};
```

- For each query graph $G_Q \in W$, we leverage the function expand wraps a specific relation R as a new node and add new edges, producing a set of query graphs containing the relation R (lines 12–16). Such query graphs are maintained in the set V.
- For each query graph $G_Q \in V$ that induces a refinable query, we add the induced refinable query to the set $\mathcal{S}_R(m, k)$ (line 19) and attempt to synthesize a string constraint (lines 20–23). Specifically, the function synLCS examines the values of a string attribute a in the positive tuples to compute the longest common substring ℓ and the strongest string predicate p (line 21), where all the values of a in the positive tuples satisfy the string constraints induced by p and ℓ. By updating Φ_Q, we add a new edge to G_Q and produce a new query graph inducing a refinable query (lines 22–23).
- Finally, we check whether a (m, k)-bounded refinable query is a query candidate or not, which yields the set $\mathcal{S}_C(m, k)$ (line 25). The sets $\mathcal{S}_R(m, k)$ and $\mathcal{S}_C(m, k)$ are exactly the sets of (m, k)-bounded refinable queries and query candidates.

Notably, we construct the string constraints on demands to strengthen the selection condition of refinable queries, which is achieved by the function synLCS at line 21. The reduction from string constraint synthesis to the LCS computation enable us to leverage existing algorithms, such as general suffix automaton [33], to compute the string literal ℓ and select a string predicate p efficiently, which promote the efficiency of query refinement.

▶ **Example 5.6.** Figure 4 shows the part of the refinement for the instance in Example 3.4. G^1_Q only contains the relation Method. After adding the nodes and edges, we construct the query graphs of refinable queries with more relations and atomic constraints, such as G^2_Q, G^3_Q, and G^4_Q. Particularly, we identify the query candidate $\kappa^{-1}(G^4_Q)$, i.e., R_Q in Example 3.2.

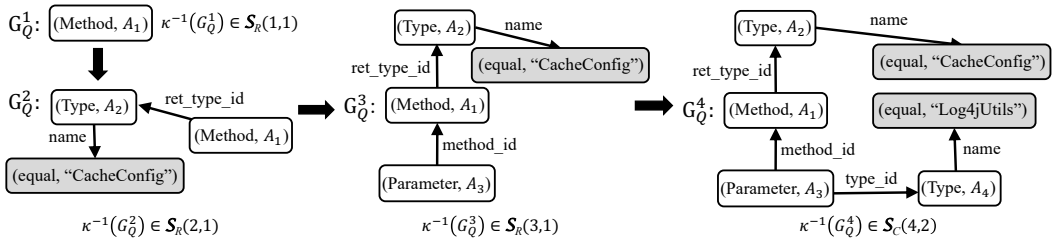


Figure 4 The example of the bounded refinement

5.3 Candidate Selection

Based on the bounded refinement, we collect the query candidates in which each relation appears no more than K times, where K is the multiplicity bound. However, not all the query candidates are the desired ones. In this section, we introduce dual quantitative metrics to prioritize queries (§ 5.3.1) and select query candidates during the refinement (§ 5.3.2).

5.3.1 Dual Quantitative Metrics

Desired queries are expected to express the search intent correctly, covering as many grammatical concepts in the natural language as possible. Also, they should be as simple as possible according to Occam's razor. Based on the intuitions, we formalize the following two metrics and then define the total order to prioritize the query candidates.

▶ **Definition 5.6.** (Named-Entity Coverage) Given a function h mapping an attribute of a relation to a set of words in natural language, the named entity coverage of a conjunctive query R_Q with respect to a natural language description s is

$$\alpha_h(R_Q, s) = \frac{1}{|N(s)|} \cdot \Big| \bigcup_{(R_i, a_j) \in \mathcal{A}(\Theta)} h(R_i, a_j) \cap N(s) \Big|$$

Here, $\mathcal{A}(\Theta)$ contains the relations and their attributes appearing in the selection condition Θ while $N(s)$ is the set of the named entities in the description s.

▶ **Definition 5.7.** (Structural Complexity) Let $\delta(\Theta)$ be the number of the atomic formulas in the selection condition Θ. The structural complexity of a conjunctive query R_Q is

$$\beta(Q) = m + \delta(\Theta)$$

To compute the named entity coverage, we instantiate the function h manually and obtain the set $N(s)$ from the natural language description s based on the named entity recognition techniques [31]. The computation does not introduce much overhead, as the natural language description exhibits a fairly small length in our scenarios. Meanwhile, measuring the structural complexity is quite straightforward. The quantities m and $\delta(\Theta)$ can be totally determined by the sizes of the sets N_Q, E_Q and Φ_Q in a query graph. Thus, computing the two quantities does not introduce much overhead during the enumeration.

▶ **Example 5.7.** Assume that we instantiate the function h as follows:

$$h(\mathsf{Parameter}, \mathsf{id}) = \{\mathsf{parameter}\},\ h(\mathsf{Method}, \mathsf{id}) = \{\mathsf{method}\},\ h(\mathsf{Method},\ \mathsf{idf\_id}) = \{\mathsf{identifier}\}$$

$$h(\mathsf{Method}, \mathsf{ret\_type\_id}) = \{\mathsf{return}, \mathsf{type}\},\ h(\mathsf{Method}, \mathsf{mdf\_id}) = \{\mathsf{modifier}\}$$

Consider the natural language description s in Example 3.4 and the conjunctive query R_Q in Example 3.2, of which the query graph is shown in Figure 3(b). We can obtain a set of named entities $N(s) = \{\mathsf{method}, \mathsf{type}, \mathsf{parameter}, \mathsf{return}\}$. Therefore, we have $\alpha_h(R_Q, s) = 1$. According to $m = 4$ and $\delta(\Theta) = 5$, its structural complexity is $\beta(R_Q) = 4 + \delta(\Theta) = 9$.

Intuitively, the selection condition of a query is more likely to conform to the user intent if the query has a higher named entity coverage. Besides, the simpler form query can have better generalization power among the query candidates covering the same number of named entities. Based on Occam's razor, we should choose the simplest query from the candidates that maximizes the named entity coverage. Thus, we propose the *total order* of conjunctive queries as follows.

▶ **Definition 5.8.** (Total Order) Given the function h in Definition 5.6, we have $R_Q^2 \preceq_s R_Q^1$ if and only if they satisfy one of the following conditions:

$$\alpha_h(R_Q^1, s) \geq \alpha_h(R_Q^2, s) \quad \text{or}$$

$$\alpha_h(R_Q^1, s) = \alpha_h(R_Q^2, s),\ \beta(R_Q^1) \leq \beta(R_Q^2)$$

▶ **Example 5.8.** Consider the following query candidates for the instance in Example 3.4.

$$R_Q^{c1} := \Pi_{(A_1.*)}(\sigma_{A_1.\mathsf{name}} = \text{“}\mathsf{foo}\text{”}(\rho_{A_1}(\mathsf{Method})))$$

$$R_Q^{c2} := \Pi_{(A_1.*)}(\sigma_{\Theta_2}(\rho_{A_1}(\mathsf{Method}) \times \rho_{A_2}(\mathsf{Type}) \times \rho_{A_3}(\mathsf{Parameter}) \times \rho_{A_4}(\mathsf{Type})))$$

Here, $\Theta_2 := \Theta \wedge (A_1.\mathsf{name} = \text{“}\mathsf{foo}\text{”})$ and Θ is shown in Example 3.2. Given the function h shown in Example 5.7, we can obtain that $\alpha_h(R_Q^{c1}, s) = \frac{1}{4}$, $\alpha_h(R_Q^{c2}, s) = 1$, $\beta(R_Q^{c1}) = 2$, and $\beta(R_Q^{c2}) = 10$. According to Example 5.7, we have $R_Q^{c1} \preceq_s R_Q^{c2} \preceq_s R_Q$.

The total order is an adaption of Occam's razor for our synthesis problem. Without the named entity coverage, we would select the query candidates with the lowest structural complexity, such as R_Q^{c1} in Example 5.8, even if they do not constrain the relationship of several grammatical constructs as expected. Based on the total order, we can select the query candidates by solving the dual-objective optimization problem, which finally yields the desired queries for code search.

5.3.2 Blending Selection with Refinement

Based on Definition 5.8, we propose Algorithm 3 that blends the candidate selection with the bounded refinement, which is more likely to obtain the desired queries for a code search task. First, we obtain the schema graph and remove the dummy relations via the representation reduction (line 2). We then compute the upper bound of the named entity coverage, which is denoted by $\widetilde{\alpha}$ (line 3). After the initialization of α_{max}, β_{min}, and $\mathcal{S}_Q$ (line 4), we conduct the bounded refinement and select the query candidates in each round (lines 5–14). Obviously, there are at most $K \cdot |\mathcal{R}'|$ relations in a query for a given multiplicity bound K (line 5), and a relation can only appear at most $min(K, m)$ times in a query with m relations (line 8). In each round, we fuse the refinement and selection as follows:

- Enumerate (m, k)-bounded refinable queries and query candidates with Algorithm 2, strengthening the selection conditions of the refinable queries in previous rounds (line 9).
- Compute $\alpha_h(R_Q, s)$ and $\beta(R_Q)$ for each (m, k)-bounded query candidate R_Q and update the selected candidate set $\mathcal{S}_Q$, α_{max}, and β_{min} (lines 10–11). Particularly, α_{max} and β_{min} are updated to identify the largest candidates with respect to the total order.

Algorithm 3 Blending selection with refinement

```
1  Procedure synthesize(Γ, R, R_p^*, R_n^*, s, K):
2      R' ← reduce(Γ, R, R_p^*, R_n^*) ;
3      α̃ ← (1/|N(s)|)|{w ∈ h(R,a) ∩ N(s) | ∃R ∈ R', T ∈ R' ∪ {STR} : (a,T) ∈ Γ(R)}|;
4      α_max ← MIN_INT; β_min ← MAX_INT; S_Q ← ∅;
5      foreach 1 ≤ m ≤ K · |R'| :
6          if S_R(m−1, k−1) = ∅ and S_R(m−1, k) = ∅ :
7              continue ;
8          foreach 1 ≤ k ≤ min(K, m) :
9              refine(S_R, S_C, R_p^*, R_n^*, m, k, R');
10             foreach R_Q ∈ S_C(m, k) :
11                 (α_max, β_min, S_Q) ← update (R_Q, α_max, β_min, S_Q);
12             β̃ = min({β(R_Q) | R_Q ∈ S_R(m,k) ∪ S_R(m,k−1)});
13             if α_max = α̃ and β_min ≤ β̃ :
14                 return S_Q;
15     return S_Q;
```

- Terminate the iteration in advance and return the set $\mathcal{S}_Q$ if α_{max} reaches the upper bound of the named entity coverage, i.e., $\widetilde{\alpha}$, and the queries to be refined in the next round do not have lower structural complexities than β_{min} (lines 13–14).

The refinement strengthens the selection conditions to exclude all the negative tuples. Specifically, we explore the bounded search space containing the query graphs of refinable queries, avoiding the unnecessary enumerative search effectively. In real-world code search tasks, the selection condition is often involved different kinds of grammatical constructs, making each relation appear often appear in the conjunction query one or two times. Therefore, we set the multiplicity bound K to 2 for real-world code search tasks in practice, of which the effectiveness will be evidenced by our evaluation.

The natural language description benefits our synthesis process from two aspects. First, the selected queries in $\mathcal{S}_Q$ are the largest queries under the total order, and thus they are more likely to conform to the user's search intent than other query candidates. Second, we terminate the enumerative search if the named entity coverage cannot increase with a smaller structural complexity, avoiding unnecessary enumerative search of bounded query candidates for the efficiency improvement.

▶ **Example 5.9.** Consider R_Q^{c1}, R_Q^{c2} and R_Q in Example 5.8. We obtain the query candidate R_Q^{c1} when $(m, k) = (1, 1)$, and discover the candidates R_Q and R_Q^{c2} when $(m, k) = (4, 2)$. Based on Definition 5.8, we select and maintain the query candidate R_Q in $\mathcal{S}_Q$. Also, we find $\alpha_{max} = \alpha_h(R_Q, s) = 1$ reaches $\widetilde{\alpha}$, indicating that the candidates in the subsequent rounds cannot yield a larger named entity coverage with lower structural complexity. Algorithm 3 terminates and returns the set $\mathcal{S}_Q = \{R_Q\}$.

5.4 Summary

Our synthesis algorithm SQUID is an instantiation of a new synthesis paradigm of the multi-modal synthesis, which reduces the synthesis problem to a multi-target optimization problem. We now formulate and prove the soundness, completeness, and optimality of SQUID with three theorems as follows, which are proved in [49].

▶ **Theorem 5.2.** (Soundness) For any $R_Q \in \mathcal{S}_Q$, where $\mathcal{S}_Q$ is returned by Algorithm 3, R_Q must contain all the positive tuples in R_p^* and exclude the negative tuples in R_n^*.

Proof. To prove the soundness of SQUID, we only need to prove that for any $R_Q \in \mathcal{S}_C(m,k)$, R_Q must contain all the positive tuples in R_p^* and exclude all the negative tuples in R_n^*. According to line 25 in Algorithm 2, we only add a conjunctive query R_Q to the set $\mathcal{S}_C(m,k)$ if R_Q only contains the positive tuples in R_p^*. Therefore, R_Q is a query candidate. The soundness of our algorithm is proved. ◀

▶ **Theorem 5.3.** (Completeness) If an MMCQS problem instance has an (m,k)-bounded query as its solution and $k \leq K$, the set $\mathcal{S}_Q$ returned by Algorithm 3 is not empty.

Proof. Assume that there exists an (m,k)-bounded query R_Q as the solution of the MMCQS problem instance. According to Theorem 5.1, we can construct another query candidate R'_Q such that

- No dummy relation appears in the Cartesian product of R'_Q.
- The query graph of R'_Q is the subgraph of the query graph of R_Q.

Meanwhile, Algorithm 3 invokes Algorithm 2 inductively to compute all the (m,k)-bounded queries and query candidates, where $k \leq K$. Also, we notice that R'_Q must belong to $\mathcal{S}_R(m,k)$ and $\mathcal{S}_C(m,k)$ for some (m,k), which implies that $\mathcal{S}_C$ can not be empty. Hence, the completeness of SQUID is proved. ◀

▶ **Theorem 5.4.** (Optimality) Denote $I = \{(m,k) \mid 1 \leq m \leq K \cdot |\mathcal{R}'|, 1 \leq k \leq min(K,m)\}$ and $\widetilde{\mathcal{S}} = \bigcup_{(m,k)\in I} \mathcal{S}_C(m,k)$. The returned query set $\mathcal{S}_Q$ of Algorithm 3 satisfies:

- $R'_Q \preceq_s R_Q$ for every $R_Q \in \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}}$.
- There do not exist $R_Q \in \widetilde{\mathcal{S}} \setminus \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}}$ such that $R'_Q \preceq_s R_Q$ and $R_Q \not\preceq_s R'_Q$.

Proof. We first introduce a set I' to contain all the pairs (m,k) iterated in the executed rounds of Algorithm 3. Denote $\widetilde{\mathcal{S}'} = \bigcup_{(m,k)\in I'} \mathcal{S}_C(m,k)$. According to the functionality of the method update invoked at line 11 in Algorithm 3, it has selected all the largest query candidates based on the total order in Definition 5.8. Hence, we can obtain that

- (P1) $R'_Q \preceq_s R_Q$ for any $R_Q \in \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}'}$
- (P2) There do not exist $R_Q \in \widetilde{\mathcal{S}'} \setminus \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}'}$ such that $R'_Q \preceq_s R_Q$ and $R_Q \not\preceq_s R'_Q$.

If all the pairs $(m,k) \in I$ are iterated, we have $\widetilde{\mathcal{S}} = \widetilde{\mathcal{S}'}$. The theorem holds trivially. If Algorithm 3 terminates from line 14, several pairs in I are not iterated, leaving several query candidates not enumerated. In what follows, we will prove the two properties in the theorem one by one for this case.

First, we try to prove $R'_Q \preceq_s R_Q$ for any $R_Q \in \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}}$, which comes to two cases. If $R'_Q \in \widetilde{\mathcal{S}'}$, we can obtain that $R'_Q \preceq_s R_Q$ according to (P1) above. Otherwise, there exists $(m,k) \in I \setminus I'$ such that $R'_Q \in \mathcal{S}_C(m,k)$. According to lines 13 and 14 in Algorithm 3, the query graph of R'_Q must have more relations or edges than the query graphs of specific queries in $\mathcal{S}_C(m-1,k-1)$ and $\mathcal{S}_C(m-1,k)$, which implies

$$\beta(R'_Q) > \min(\{\beta(R_Q) \mid R_Q \in \mathcal{S}_C(m-1,k-1) \cup \mathcal{S}_C(m-1,k)\})$$

More generally, we have

$$\min(\{\beta(R_Q) \mid R_Q \in \mathcal{S}_C(m,k)\}) > \min(\{\beta(R_Q) \mid R_Q \in \mathcal{S}_C(m-1,k-1) \cup \mathcal{S}_C(m-1,k)\})$$

Therefore, we have $\beta(R'_Q) > \beta^*_{min}$, where β^*_{min} is the value of β_{min} when Algorithm 3 terminates. Recap that β^*_{min} is the structural complexity of the queries in $\mathcal{S}_C$, while the

queries in $\mathcal{S}_C$ have reached $\widetilde{\alpha}$, i.e., the upper bound of the named entity coverage. Thus, we can easily obtain $R'_Q \preceq_s R_Q$ for any $R_Q \in \mathcal{S}_Q$ based on the total order in Definition 5.8.

Second, we try to prove that there does not exist $R_Q \in \widetilde{\mathcal{S}} \setminus \mathcal{S}_Q$ and $R'_Q \in \widetilde{\mathcal{S}}$ such that $R'_Q \preceq_s R_Q$ and $R_Q \not\preceq_s R'_Q$. If it does not hold, we can find $(m, k) \in I \setminus I'$ such that $R_Q \in \mathcal{S}_C(m, k)$ and $\beta(R_Q) < \beta^*_{min}$. However, $\beta(R_Q) > \widetilde{\beta}^* \geq \beta^*_{min}$, where $\widetilde{\beta}^*$ is the value of $\widetilde{\beta}$ when Algorithm 3 terminates. Contradiction!

Lastly, we have proved the optimality of our synthesis algorithm. ◀

6 Implementation

Established upon the industrial Datalog-based Java program analyzer in Ant Group, Squid synthesizes conjunctive queries to support code search tasks in Java programs. Noting that our approach is general enough to support the conjunctive query synthesis for any Datalog-based analyzer as long as the generated relations can be formulated by Definition 3.2. In what follows, we provide more implementation details of Squid.

Synthesis Input Configuration. We design a user interface to convenience the users to specify examples in a code snippet. Specifically, the users can copy a desired grammatical construct from their workspace as a positive example or write a positive example manually. By mutating a positive example, the users can create more positive and negative examples, eventually forming an example program. Then we convert the program to the relational representation, which consists of 173 relations with 1,093 attributes in total, and partitions a relation into two parts to induce positive and negative tuples. To extract the named entities from the natural language description, we leverage the named entity recognition [31] and construct the dictionary of entities to filter unnecessary named entities in the post-processing. Specifically, the dictionary contains 205 words, which are the keywords describing grammatical constructs in Java programs, such as "method", "parameter", and "return". Furthermore, we also instantiate the function h in Definition 5.6 to bridge the program relational representation with natural language words. We publish all the synthesis specifications and dictionary of entities online [48].

Synthesis Algorithm Design. Based on the language schema of Java, we construct the schema graph offline and persist it for synthesizing queries for a given synthesis specification. Instead of invoking the Datalog-based analyzer, we implement a query evaluator for conjunctive queries upon the relational representation to identify the refinable queries and query candidates, which can improve the efficiency of the query evaluation during the synthesis. In the bounded refinement, we set the multiplicity bound K to 2 by default to support code search tasks. To efficiently synthesize string constraints, we leverage the generalized suffix automaton [33] to identify the longest common substrings of a set of string values, which returns the string predicate p and the string literal ℓ with low time overhead. Currently, Squid utilizes four predicates for string match, while we can further extend it to support regex match by adopting existing regex synthesis techniques [27, 10] to Algorithm 2.

7 Evaluation

To quantify the effectiveness and efficiency of Squid, we conduct a comprehensive empirical evaluation and answer the following four research questions:

- **RQ1:** How effective is Squid in the conjunctive query synthesis for code search tasks?

- **RQ2:** How big are the benefits of the representation reduction and the bounded refinement in terms of efficiency?
- **RQ3:** Is the query candidate selection effective and necessary for the synthesis?
- **RQ4:** How does SQUID compare to other approaches that could be used in our problem?

Benchmark. There are no existing studies targeting our multi-modal synthesis problem, so we construct a new benchmark for evaluation, which consists of 31 code search tasks. As shown in Table 1, the tasks cover five kinds of grammatical constructs, namely variables, expressions, statements, methods, and classes. Specifically, 14 tasks are the variants of C++ search tasks in [34] or the query synthesis tasks in [46]. We also consider more advanced tasks deriving from real demands. For example, Task 2 originates from the coding standard of a technical unit in Ant Group, while Task 21 is often conducted when the developers check the usage of the log4j library to improve reliability. For each task, we specify examples in a program and a sentence as the natural language description. The program is fed to the commercial Datalog-based analyzer in Ant Group to generate the relational representation and the relation partition, while the natural language description is processed via the named entity recognition technique [31]. The columns **L** and **(P, N)** in Table 1 indicate the line numbers of the programs and the numbers of positive/negative tuples, respectively.

Experimental Setup. We conduct all the experiments on a Macbook Pro with a 2.6 GHz Intel® Core™ i7-9750H CPU and 16 GB physical memory.

7.1 Overall Effectiveness

To evaluate the effectiveness of SQUID, we run it upon the synthesis specification for each code search task, examining whether the synthesized queries express the intent correctly, and meanwhile, measure the time cost of synthesizing queries in each task.

In Table 1, the column $|\mathbf{G_Q}|$ indicates the numbers of the relations, equality constraints, and string constraints. The column **k** shows the maximal multiplicity of a relation in a synthesized query, while the column $|\mathbf{G'_\Gamma}|$ indicates the numbers of nodes and edges in the subgraph of the schema graph induced by $\mathcal{R}' \cup \{\mathsf{STR}\}$. The time cost of SQUID is shown in the column $\mathbf{T}_0$. According to the statistics, we can obtain two main findings. First, SQUID synthesizes the queries for all the tasks successfully. It manipulates more than three relations in Tasks 22 and 25, which are even non-trivial for a human to achieve. Second, SQUID synthesizes the queries with a quite low time cost. The average time cost is 2.56 seconds, while most of the tasks are finished in three seconds.

As mentioned in § 6, SQUID performs the bounded refinement with the multiplicity bound $K = 2$. In our benchmark, five code search tasks demand several relations appear two times. In practice, the searching condition can hardly relate to more than two grammatical constructs of the same kind, so our setting of the multiplicity bound K enables SQUID to synthesize queries for code search tasks in real-world scenarios. Meanwhile, we quantify the time cost of the synthesis in the cases of $K = 3$ and $K = 4$. Averagely, SQUID takes 2.71 seconds and 3.98 seconds under the two settings, respectively. Thus, the overhead increases gracefully when K increases, demonstrating the great potential of SQUID in efficiently synthesizing more sophisticated queries with a larger multiplicity bound.

7.2 Ablation Study on Efficiency

We evaluate two ablations of SQUID, namely SQUIDNRR and SQUIDNBR, to quantify the impact of the representation reduction and the bounded refinement on the efficiency.

Table 1 Experiment results of synthesizing conjunctive queries for code search tasks.

ID	Description	L	(P, N)	$\lvert G_Q \rvert$	k	$\lvert G'_\Gamma \rvert$	T_0(s)
1	Local variables with double type	10	(2, 2)	(2, 1, 1)	1	(10, 19)	2.36
2	Float variables of which the identifier contains “cash”	10	(3, 1)	(3, 2, 2)	1	(9, 17)	2.37
3	Public field variables of a class	6	(2, 1)	(2, 1, 1)	1	(10, 21)	2.30
4	Public field variables whose names use “cash” as suffixes	8	(3, 2)	(2, 2, 2)	1	(10, 23)	2.49
5	Arithmetic expressions using double-type operands	9	(2, 2)	(3, 2, 2)	1	(9, 29)	2.62
6	Cast expressions from double-type to float-type [46]	11	(1, 2)	(3, 2, 2)	2	(9, 23)	2.31
7	Arithmetic expressions only using literals as operands	19	(2, 3)	(3, 2, 1)	2	(9, 29)	2.76
8	Expressions comparing a variable and a boolean literal [34]	16	(2, 1)	(2, 1, 1)	1	(8, 29)	2.36
9	New expressions of ArrayList	8	(2, 1)	(2, 1, 1)	1	(10, 25)	2.20
10	Logical conjunctions with a boolean literal [34]	11	(3, 1)	(2, 1, 2)	1	(9, 29)	2.31
11	Float increment expression [46]	11	(1, 2)	(3, 2, 2)	1	(9, 23)	2.55
12	Expressions comparing two strings with “==” [34]	14	(2, 1)	(3, 2, 3)	1	(11, 46)	3.01
13	Expressions performing downcasting [46]	25	(2, 1)	(3, 2, 0)	1	(11, 39)	2.63
14	The import of LocalTime	7	(1, 1)	(1, 0, 2)	1	(9, 23)	2.17
15	The import of the classes in log4j	9	(3, 1)	(1, 0, 1)	1	(9, 22)	2.22
16	Labeled statements using “err” as the label [34]	17	(1, 1)	(1, 0, 1)	1	(10, 21)	2.18
17	If-statements with a boolean literal as a condition [34]	16	(2, 1)	(2, 1, 0)	1	(9, 17)	2.24
18	For-statements with a boolean literal as the condition [34]	15	(2, 1)	(2, 1, 0)	1	(10, 25)	2.31
19	Invocation of unsafe time function “localtime” [34]	9	(2, 1)	(2, 1, 1)	1	(10, 23)	2.23
20	Public methods with void return type [34]	10	(2, 1)	(3, 2, 2)	1	(11, 26)	2.36
21	Methods receiving a parameter with Log4jUtils type	11	(2, 1)	(3, 2, 1)	1	(9, 20)	2.45
22	Methods using a boolean parameter as a if-condition [46]	29	(2, 2)	(4, 3, 0)	1	(11, 26)	3.23
23	Methods creating a File object	14	(2, 1)	(3, 2, 1)	1	(12, 30)	2.38
24	Mutually recursive methods [34, 46]	20	(2, 2)	(2, 2, 0)	2	(11, 27)	2.42
25	Overriding methods of classes [46]	25	(2, 4)	(5, 5, 0)	2	(8, 22)	5.89
26	User classes with “login” methods	15	(2, 1)	(2, 1, 2)	1	(11, 28)	2.53
27	Classes containing a field with Log4jUtils type	20	(2, 1)	(3, 2, 1)	1	(12, 35)	2.42
28	Classes having a subclass	25	(3, 3)	(2, 1, 0)	2	(13, 33)	2.21
29	Classes implementing Comparable interface	16	(2, 1)	(2, 1, 1)	1	(13, 35)	2.58
30	Classes containing a static method	17	(2, 1)	(3, 2, 1)	1	(11, 28)	2.46
31	Java classes with main functions	16	(2, 1)	(2, 1, 1)	1	(10, 28)	2.35

- SquidNrr: This ablation of Squid does not perform the representation reduction but still leverages Algorithm 2 to conduct the bounded refinement.
- SquidNbr: The ablation performs the representation reduction as Squid does, while it enumerates all the query graphs and permits each relation to appear at most K times.

We measure the time cost of two ablations to quantify their efficiency. Specifically, we set the time budget for synthesizing queries for a single task to 30 seconds, as a synthesizer would have little practical value for the real-world code search if it ran out of the time budget.

Figure 5 shows the comparison of the time cost of Squid and the two ablations. First, the representation reduction can effectively reduce the time cost. Specifically, the average time cost of SquidNrr is 8.98 seconds, indicating that the representation reduction introduces a 71.49% reduction over the time cost. Second, the bounded refinement has a critical impact on the efficiency of Squid. Without the refinement, SquidNbr has to explore the huge search space induced by the non-dummy relations, making 14 out of 31 tasks cannot be finished within the time budget, such as Task 4, Task 5, etc. For the failed tasks, we do not show the time cost of SquidNbr in Figure 5. SquidNbr also takes much more time than Squid, consuming 7.89 seconds on average, even if it successfully synthesizes the queries.

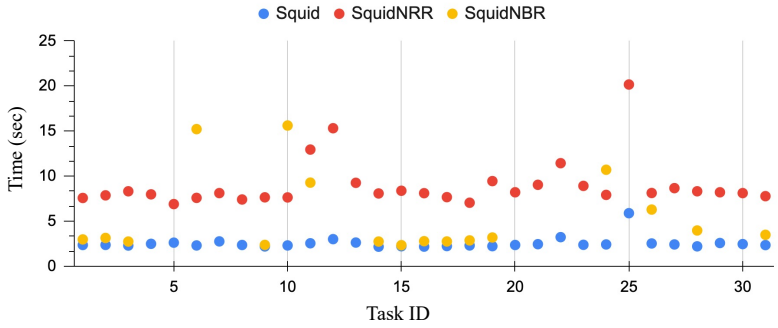
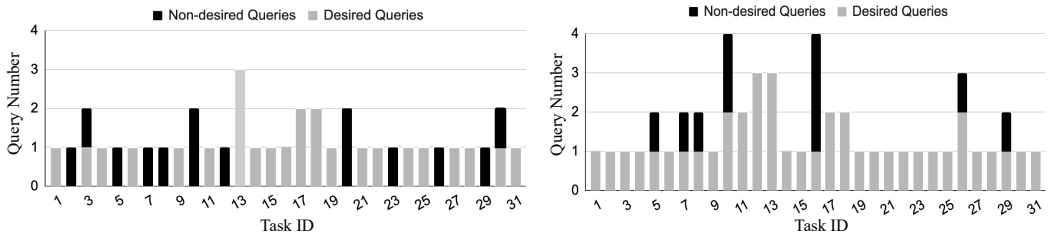


Figure 5 The time cost comparison of SQUID, SQUIDNRR, and SQUIDNBR



(a) structural complexity (b) named-entity coverage

Figure 6 The numbers of synthesized queries prioritized with different metrics

To investigate how the efficiency is improved, we further measure the size of the subgraph of the schema graph induced by $\mathcal{R}' \cup \{\mathsf{STR}\}$. Initially, the schema graph contains 174 nodes (including the node depicting STR) and 1,093 edges. As shown in the column $|\mathbf{G}'_\Gamma|$ of Table 1, the induced subgraph only contains around ten nodes and no more than fifty edges. Although SQUIDNRR prunes unnecessary relations by enumerating several bounded queries at the beginning of the refinement, it has to spend more time on the query enumeration than SQUID, which demonstrates the critical role of the bounded refinement in our synthesis. Besides, the running time of SQUIDNBR is similar to SQUID on several benchmarks, such as Task 1, Task 2, Task 3, Task 9, etc., while it takes much longer time than SQUID in other benchmarks. Although SQUIDNBR does not discard infeasible queries, it still benefits from representation reduction. When a desired query is of small size and the reduced program representation induces a small schema graph G'_Γ, SQUIDNBR can terminate to find an optimal query by enumerating a few candidates. However, if G_Q and G'_Γ are large, SQUIDNBR enumerates a large number of infeasible queries, which introduces significant overhead.

7.3 Impact of Selection

To measure the impact of the query candidate selection, we adapt each metric separately for candidate prioritization. Specifically, we alter Algorithm 3 and select the queries minimizing the structural complexity and maximizing the named entity coverage, respectively. We then count the returned queries and inspect whether they are desired ones or not.

Figure 6(a) shows the numbers of the synthesized queries with the structural complexity as the metric. As we can see, SQUID produces non-desired queries in 12 tasks, while the returned set of synthesized queries in 10 tasks do not contain any desired query. For Task 3 and Task 30, it provides the desired queries along with non-desired ones, which makes the users confused about how to select a proper one. Similar to R_Q^{c1} in Example 5.8, the non-desired queries are caused by the over-fitting of positive and negative examples. Although they have

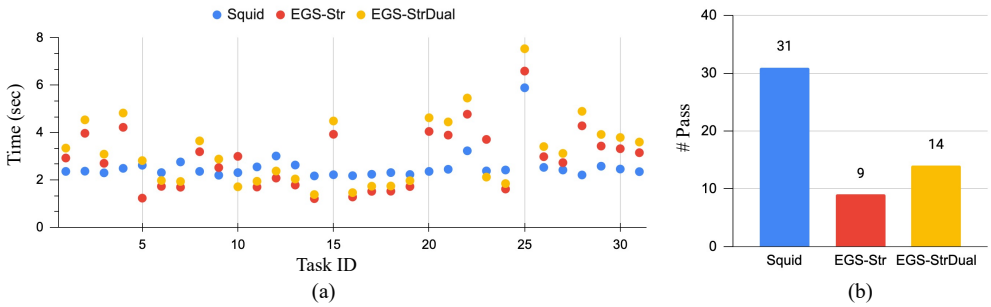


Figure 7 The time cost and the numbers of passed tasks of Squid, EGS-Str, and EGS-StrDual

the simplest form of the selection conditions, the relationship of grammatical constructs mentioned in the natural language description is not constrained, making synthesized queries not express the users' intent correctly.

Figure 6(b) shows the numbers of the queries maximizing the named entity coverage. Squid returns at least one non-desired query for seven tasks. Similar to R_Q^{c2} in Example 5.8, non-desired queries come from over-complicated selection conditions. Although the selected queries have the same named entity coverage, several queries contain more atomic formulas than the desired ones, posing stronger restrictions upon the code. Besides, the synthesized queries in several tasks, e.g., Task 11 and Task 26, have complex selection conditions although they are equivalent under the context of code search. However, such queries exhibit higher structural complexity, posing more difficulty in understanding them.

7.4 Comparison with Existing Techniques

To the best of our knowledge, no existing technique or implemented tool targets the same problem as Squid. To compare Squid with existing effort, we adapt the state-of-the-art Datalog synthesizer EGS [46] as our baseline. Originally, it synthesizes a conjunctive query to separate a positive tuple from all the negative ones and then group all the conjunctive queries as the final synthesis result, which can be theoretically a disjunctive query. However, EGS does not synthesize string constraints and only prioritizes feasible solutions based on their sizes, i.e., the structural complexity in our work. Thus, we construct two adaptions, namely EGS-Str and EGS-StrDual, to synthesize the queries under our problem setting.

- EGS-Str computes the longest substring of each string attribute in the positive tuple such that the string values of the attributes in negative ones do not contain it as the substring. We follow the priority function in EGS, which consists of the number of undesirable tuples eliminated per atomic constraint and the size of a query, to accelerate searching a query candidate with a small size. Finally, it obtains a query candidate for each positive tuple and groups the candidates as a result.
- EGS-StrDual further extends EGS-Str by considering the named entity coverage. Specifically, it prioritizes the refinable queries according to the three metrics, including the number of undesirable tuples eliminated per atomic constraint, the named entity coverage, and the size of a query. Other settings are the same as the ones of EGS-Str.

Figure 7 shows the results of the comparison. On average, EGS-Str and EGS-StrDual spend 2.85 and 3.18 seconds on a synthesis instance, respectively, while the average time cost of Squid is 2.56 seconds. Although EGS-Str and EGS-StrDual accelerate the searching

process with priority functions as the heuristic metrics, they have to process the positive tuples in each round, and thus, the number of positive tuples can increase the overhead.

Meanwhile, the two baselines only succeed synthesizing queries for 9 and 14 tasks, respectively. There are two root causes of their failures in synthesizing desired queries. First, they synthesize the query candidates for each positive query separately and, thus, are more prone to over-fitting problems than Squid. Second, the core algorithms of EGS and the two adaptions, which are the instantiations of inductive logic programming, can not guarantee the obtained solutions are optimal under the given metrics. As reported in [46], the query may not be of the minimal size if EGS leverages the number of undesirable tuples eliminated per atomic constraint to accelerate the searching process of a query candidate. In our problem, our dual quantitative metrics increase the difficulty of achieving the optimal solutions with the two adaptions, which causes the failures of the code search tasks.

7.5 Discussion

In what follows, we demonstrate the discussions on the limitations of Squid and several future works, which can further improve the practicality of our techniques.

Limitations. Although Squid is demonstrated to be effective for code search, it has two major limitations. First, Squid cannot synthesize the query where the multiplicity of a relation is larger than the multiplicity bound K. In other words, Theorem 5.3 actually ensures partial completeness. Although we may achieve completeness for all realizable instances by enumerating queries until obtaining a query candidate in Algorithm 3, Squid would fail to terminate for unrealizable instances. Second, Squid does not support synthesizing the queries with logical disjunctions. However, when a code search task involves the matching of multiple patterns, Squid would not discover the correct queries, which are out of the scope of the conjunctive queries.

Future Works. In the future, we will attempt to propose an efficient decision procedure to identify unrealizable instances. Equipped with the decision procedure, we can only enumerate queries for realizable instances and generalize the query refinement by discarding the multiplicity bound. Besides, it would be promising to generalize Squid for disjunctive query synthesis. One possible adaption is to divide positive tuples into proper clusters and synthesize a conjunctive part for each cluster separately, following existing studies such as EGS [46] and RhoSynth [15]. In addition, we aim to expand Squid to diverse program domains, such as serverless functions [50] and programs running on networking devices [57]. These use cases have gained significant attention in recent years, which can pose new challenges on code search where new approaches may be needed. Lastly, it would be meaningful to combine Squid with the techniques in the community of human-computer interaction [8] to unleash its benefit for practice use.

8 Related Work

Multi-modal Program Synthesis. There has been a vast amount of literature on the multi-modal synthesis [11, 4, 9, 10, 40, 16, 34]. For example, the LTL formula synthesizer LtlTalk [16] maximizes the objective function that measures the similarity between the natural language description and the LTL formula, and searches for the optimal solution that distinguishes the positive and negative examples. Squid bears similarities to LtlTalk in terms of the prioritization, while we use the named entities to avoid the failure of semantic parsing of a sentence. Another closely related work is a query synthesizer named Sporq [34].

Based on code examples and user feedback, SPORQ iterates its PBE-based synthesis engine to refine the queries, which demands verbose user interactions and a long time period. In contrast, SQUID automates the code search by solving a new multi-modal synthesis problem, which only requires the users to specify code examples and a natural language description. effectively relieving the user's burden in the searching process.

Component-based Synthesis. Several recent studies aim to compose several components (e.g., the classes and methods in the libraries) into programs that achieve target functionalities [25, 18, 24, 38, 20, 21]. Typically, SYPET [14] and APIPHANY [19] both use the Petri net to encode the type signature of each function, and collect the reachable paths to enumerate the well-typed sketches of the programs, which prunes the search space at the start of the synthesis. In our work, SQUID leverages the schema graph to guide the enumerative search, which share the similarity with existing studies. However, our enumerative search space does not consist of the reachable paths in the schema graph, and instead, contains different choices of selecting its nodes and edges. Besides, unlike prior efforts [14, 24, 20, 19], SQUID computes the activated relations and then discards unnecessary components, i.e., dummy relations, which distinguishes SQUID significantly from other component-based synthesizers.

Datalog Program Synthesis. There have been many existing efforts of synthesizing Datalog programs [2, 42, 39, 43, 32]. For example, ZAATAR [2] encodes the input-output examples and Datalog programs with SMT formulas, and synthesizes the candidate solution via constraint solving. Unlike constraint-based approaches, ALPS [42] and GENSYNTH [32] synthesize target Datalog programs via the enumerative search, which is similar to our synthesis algorithm. However, existing studies do not tackle a large number of relations in the synthesis [2, 42, 32] or pursue an optimal solution with respect to a natural language description. Meanwhile, they do not support the synthesis of string constraints, making their approaches incapable of string matching-based code search. In contrast, SQUID ensures soundness, completeness, and optimality simultaneously and synthesizes string constraints for string matching, showing its potential in assisting real-world code search tasks.

Datalog-based Program Analysis. The past few decades have witnessed the increasing popularity of Datalog-based program analysis [22, 52, 55, 3, 44]. For example, CODEQL encodes a program with a relational representation and exposes a query language for query writing [3]. Several analyzers target more advanced semantic reasoning. For example, the points-to and alias facts are depicted by two kinds of relations in DOOP [6], and meanwhile, pointer analysis algorithms are instantiated as Datalog rules [44]. Other properties, such as def-use relation and type information, can also be analyzed by existing analyzers [26, 37]. Our effort has shown the opportunity of unleashing the power of Datalog-based program analyzers seamlessly to support the code search automatically.

9 Conclusion

We propose an efficient synthesis algorithm SQUID for a multi-modal conjunctive query synthesis problem, which enables automatic code search using a Datalog-based program analyzer. SQUID reduces the search space via the representation reduction and the bounded refinement, and meanwhile, conducts the query candidate selection with dual quantitative metrics. It efficiently synthesizes the queries for 31 code search tasks with the guarantees of soundness, completeness, and optimality. Its theoretical and empirical results offer strong evidence of its practical value in assisting code search in real-world scenarios.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

2 Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 689–706. Springer, 2017. doi:10.1007/978-3-319-66158-2_44.

3 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.2.

4 Christopher Baik, Zhongjun Jin, Michael J. Cafarella, and H. V. Jagadish. Duoquest: A dual-specification system for expressive SQL queries. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2319–2329. ACM, 2020. doi:10.1145/3318464.3389776.

5 Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24(6):377–380, 1987. doi:10.1016/0020-0190(87)90114-1.

6 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009. doi:10.1145/1640089.1640108.

7 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977. doi:10.1145/800105.803397.

8 Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Commun. ACM*, 64(8):98–106, 2021. doi:10.1145/3469279.

9 Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 328–343. ACM, 2021. doi:10.1145/3453483.3454047.

10 Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 487–502. ACM, 2020. doi:10.1145/3385412.3385988.

11 Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 602–612. ACM, 2019. doi:10.1145/3338906.3338951.

12 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016. doi:10.1145/2970276.2970347.

13 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In Albert

Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436. ACM, 2017. doi:10.1145/3062341.3062351.

14 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017. doi:10.1145/3009837.3009851.

15 Pranav Garg and Srinivasan H. Sengamedu. Synthesizing code quality rules from examples. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi:10.1145/3563350.

16 Ivan Gavran, Eva Darulova, and Rupak Majumdar. Interactive synthesis of temporal specifications from examples and natural language. *Proc. ACM Program. Lang.*, 4(OOPSLA):201:1–201:26, 2020. doi:10.1145/3428269.

17 Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006. doi:10.1145/1131342.1131345.

18 Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61. ACM, 2011. doi:10.1145/1993498.1993505.

19 Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 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*, pages 122–136. ACM, 2022. doi:10.1145/3519939.3523450.

20 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020. doi:10.1145/3371080.

21 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013. doi:10.1145/2491956.2462192.

22 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *codeQuest:* scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006. doi:10.1007/11785477_2.

23 IntelliJ IDEA. Structural search and replace, https://www.jetbrains.com/help/idea/structural-search-and-replace.html, 2022. [Online; accessed 10-Nov-2022]. URL: https://www.jetbrains.com/help/idea/structural-search-and-replace.html.

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

25 Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010. doi:10.1145/1806799.1806833.

26 Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2005.

27 Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In Bernd Fischer and Ina Schaefer, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts*

and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016, pages 70–80. ACM, 2016. doi:10.1145/2993236.2993244.

28 Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 1294–1303. The Association for Computer Linguistics, 2013. URL: https://aclanthology.org/P13-1127/.

29 Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware code search for javascript frameworks. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 690–701. ACM, 2016. doi:10.1145/2950290.2950341.

30 Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. Opportunities and challenges in code search tools. *ACM Comput. Surv.*, 54(9):196:1–196:40, 2022. doi:10.1145/3480027.

31 Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60. The Association for Computer Linguistics, 2014. doi:10.3115/v1/p14-5010.

32 Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. GENSYNTH: synthesizing datalog programs without language bias. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6444–6453. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16799.

33 Mehryar Mohri, Pedro J. Moreno, and Eugene Weinstein. General suffix automaton construction algorithm and space bounds. *Theor. Comput. Sci.*, 410(37):3553–3562, 2009. doi:10.1016/j.tcs.2009.03.034.

34 Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In Jeffrey Nichols, Ranjitha Kumar, and Michael Nebeling, editors, *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, pages 84–99. ACM, 2021. doi:10.1145/3472749.3474737.

35 Mayur Naik. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*, 2011.

36 Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA):139:1–139:29, 2019. doi:10.1145/3360565.

37 Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=WQc075jmBmf.

38 Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286. ACM, 2012. doi:10.1145/2254064.2254098.

39 Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL):62:1–62:27, 2020. doi:10.1145/3371130.

40 Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint*

Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pages 792–800. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/117.

41 Logging Services. Apache log4j security vulnerabilities , https://logging.apache.org/log4j/2.x/security.html, 2021. [Online; accessed 10-Nov-2022]. URL: https://logging.apache.org/log4j/2.x/security.html.

42 Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 515–527. ACM, 2018. doi:10.1145/3236024.3236034.

43 Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6117–6124. ijcai.org, 2019. doi:10.24963/ijcai.2019/847.

44 Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer, 2010. doi:10.1007/978-3-642-24206-9_14.

45 CodeQL. CodeQL for Java. https://codeql.github.com/docs/codeql-language-guides/codeql-for-java/, 2022. [Online; accessed 10-Nov-2022].

46 Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-guided synthesis of relational queries. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1110–1125. ACM, 2021. doi:10.1145/3453483.3454098.

47 Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in C. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 752–762. ACM, 2017. doi:10.1145/3106237.3106300.

48 Squid. SquidData. https://github.com/SquidData/SquidData, 2022. [Online; accessed 10-Nov-2022].

49 Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang. Synthesizing conjunctive queries for code search. *CoRR*, abs/2305.04316, 2023. URL: https://arxiv.org/abs/2305.04316, arXiv:2305.04316, doi:arXiv.2305.04316.

50 Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 493–509, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3542929.3563471.

51 Brendon J Wilson. Java coding convention, 2000.

52 Xiuheng Wu, Chenguang Zhu, and Yi Li. DIFFBASE: a differential factbase for effective software evolution management. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 503–515. ACM, 2021. doi:10.1145/3468264.3468605.

53 Yingfei Xiong and Bo Wang. L2S: A framework for synthesizing the most probable program under a specification. *ACM Trans. Softw. Eng. Methodol.*, 31(3):34:1–34:45, 2022. doi:10.1145/3487570.

54 Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017. doi:10.1145/3133887.

55 Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 590–604. IEEE Computer Society, 2014. doi:10.1109/SP.2014.44.

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

57 Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Meeting slos in cross-platform nfv. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 509–523, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3386367.3431292.

58 Xiangyu Zhou, Rastislav Bodík, Alvin Cheung, and Chenglong Wang. Synthesizing analytical SQL queries from computation demonstration. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 168–182. ACM, 2022. doi:10.1145/3519939.3523712.