https://doi.org/10.1016/j.jss.2020.110568

Enhancing Example-Based Code Search with Functional Semantics[☆]

Zhengzhao Chen[a], Renhe Jiang[a], Zejun Zhang[a], Yu Pei[b], Minxue Pan[a], Tian Zhang[a], Xuandong Li[a]

[a]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[b]*Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China*

Abstract

As the quality and quantity of open source code increase, effective and efficient search for code implementing certain semantics, or semantics-based code search, has become an emerging need for software developers to retrieve and reuse existing source code. Previous techniques in semantics-based code search encode the semantics of loop-free Java code snippets as constraints and utilize an SMT solver to find encoded snippets that match an input/output (IO) query. We present in this article the Quebio approach to semantics-based search for Java methods. Quebio advances the state-of-the-art by supporting important language features like invocation to library APIs and enabling the search to handle more data types like array/List, Set, and Map. Compared with existing approaches, Quebio also integrates a customized keyword-based search that uses as the input a textual, behavioral summary of the desired methods to quickly prune the methods to be checked against the IO examples. To evaluate the effectiveness and efficiency of Quebio, we constructed a repository of 14,792 methods from 723 open source Java projects hosted on GitHub and applied the approach to resolve 47 queries extracted from StackOverflow. Quebio was able to find methods correctly implementing the specified IO behaviors for 43 of the queries, significantly outperforming the existing semantics-based code search techniques. The average search time with Quebio was about 213 seconds for each query.

Keywords: semantics-based code search, symbolic analysis, SMT solver

[☆]This work is supported by National Natural Science Foundation (Grant Nos. 61690204, 61632015 and 61972193) of China, and partly supported by the Hong Kong RGC General Research Fund (GRF) under grant PolyU 152703/16E and PolyU 152002/18E, and the Hong Kong Polytechnic University internal fund 1-ZVJ1 and G-YBXU.

Email addresses: `141220018@smail.nju.edu.cn` (Zhengzhao Chen), `131220105@smail.nju.edu.cn` (Renhe Jiang), `MG1733092@smail.nju.edu.cn` (Zejun Zhang), `csypei@comp.polyu.edu.hk` (Yu Pei), `mxp@nju.edu.cn` (Minxue Pan), `ztluck@nju.edu.cn` (Tian Zhang), `lxd@nju.edu.cn` (Xuandong Li)

1. Introduction

As open source code getting more prevalent and more frequently used in constructing new software systems [1], searching for code that can be reused, with or without modifications, to solve the programming tasks at hand, i.e., code search, is becoming an inevitable activity in software development. A recent case study shows that on average each developer conducts five search sessions with 12 total queries for source code every workday [2].

To search for code, programmers can turn to general-purpose search engines like Google or source-code-hosting platforms like SourceForge and GitHub, which index source code and software projects based on their textual information or documentation to support keyword-based code search. Since no semantic information is taken into account when deciding whether a piece of code should be considered as a match during the search, keyword-based search often produce results that are irrelevant. To improve the search result quality, various semantics-based code search techniques have been developed in the past few years [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], among which techniques for example-based code search have yielded promising results [3, 19, 20, 21]. In example-based code search, a query typically contains a set of examples specifying the desired input/output (IO) behaviors of the target code. While such IO examples provide just incomplete functional specifications, they are lightweight—therefore easy to write and comprehend—and allow for partial matching of code: A piece of source code constitutes a match as long as it supports the specified IO behaviors, while how the code behaves beyond the examples is unimportant, which helps the search to focus on the core functionalities of code.

Satsy [19, 20, 21] is one of the state-of-the-art techniques for example-based code search. Code search using Satsy is conducted in two phases. During an offline encoding phase, Satsy gathers a collection of code snippets and encodes the semantics of each code snippet into a logical formula concerning its input and output variables. During an online search, Satsy binds concrete values from IO examples to compatible variables in each formula to construct a constraint, and checks the satisfiability of the constraint using a solver: the corresponding code snippet implements an IO behavior if the constraint is satisfiable, and the snippet is considered a match for the search if it implements all the IO behaviors specified in a query. Although Satsy has been applied to search for Java code in previous studies [19, 20, 21], its usefulness in daily code search activities is limited, as it handles only loop-free code snippets manipulating data of char, int, boolean, and String types.

In this paper, we propose a new approach, named Quebio, to example-based search for Java methods. Taken a group of IO examples as the input, Quebio searches for Java methods that implement the specified IO behaviors. Similar to Satsy, the application of Quebio also involves two phases: an offline symbolic encoding phase and an online method search phase. However, symbolic encoding in Quebio is able to support more language features like invocation to library APIs, which enables Quebio to handle more data types like array/List, Set, and

Map during the search. Method search with Quebio also involves a customized keyword-based search to quickly prune the methods to be checked against the IO examples, if a textual summary of the expected method behaviors is provided as part of the query. The new features enable Quebio to be used in a wider range of scenarios and be more efficient.

To evaluate the effectiveness and efficiency of Quebio, we built a local repository of 14,792 methods from 723 open source projects hosted on GitHub, constructed 47 queries based on questions from StackOverflow, and applied the approach to resolve the queries. Quebio managed to find methods correctly implementing the specified IO behaviors for 43 of the queries, among which 27 are most likely beyond the capability of Satsy, and the average search time with Quebio was about 213 seconds for each query. Such results suggest that Quebio significantly advances the state-of-the-art in semantics-based code search.

This work extends our previous research reported in [22]. The main differences between this work and the previous one include the following. First, our previous technique applies a set of predefined program transformation rules to encode the semantics of a method. However, it supports only a very limited number of Java language features and often produces large formulas with many temporary variables. In this work, we employ a revised version of the Symbolic PathFinder (SPF) symbolic execution engine [23] to encode Java methods, which enables us to handle more language features and leads to formulas with reduced size and complexity. Second, to further improve the efficiency of the online method search, a customized keyword-based search is introduced to quickly filter out less relevant methods so that fewer methods have to be checked by the solver. Third, we conduct a more comprehensive study to evaluate the effectiveness and efficiency of Quebio.

The primary contributions of our work are as follows:

- An approach, named Quebio, to semantics-based search for Java method using IO examples; Compared with the state-of-the-art approach Satsy, Quebio supports more language features (like loops and invocations to library APIs) and more data types (like array/List, Set, and Map), leading to improved applicability of the approach, and it employs a customized keyword-based search to quickly filter out methods that are less likely matches of the query under question, leading to improved search efficiency.

- An implementation of the approach that supports automated encoding of Java methods and efficient search for methods with expected IO behaviors;

- A large-scale evaluation of the approach based on 14,792 Java methods from real-world projects and 47 queries extracted from StackOverflow. Quebio was able to find correct methods for 43 queries, with search time for each query averaging to around 213 seconds.

The rest of the paper is organized as follows. Section 2 uses an example to illustrate how Quebio can be used to effectively find Java methods of interest. Section 3 gives an overview of the Quebio approach and explains how it works

ID	INPUT	OUTPUT
E1	[1]	[1]
E2	[1,2]	[1,2]
E3	[2,1]	[1,2]
E4	[1,3,4,2]	[1,2,3,4]
E5	[12,34,8,65,22]	[8,12,22,34,65]

Table 1: Example INPUTS and OUTPUTS for a method that sorts an array of integers in increasing order. Quebio uses comma-separated values surrounded by a pair of square brackets to represent an array or a list. IDS are not part of the input for Quebio, but are added here for easy reference.

in detail. Section 4 reports on the empirical experiments we conduct to evaluate Quebio and the corresponding results. Section 5 reviews recent works related to Quebio and Section 6 concludes the paper.

2. An Illustrative Example

In this section, we use an example to demonstrate from a user's perspective how the Quebio approach can be used to find Java methods that implement behaviors specified in IO examples.

To sort a given array of integer values in increasing order is a simple, yet common, task in software development. While the Java language already provides an efficient API (namely `Arrays.sort()`) to do the sorting, programmers from different backgrounds still frequently ask questions about how to implement it. In fact, a question about "sort an array in Java"[1] was posted in 2012 on StackOverflow—a popular Q&A website for programmers—and 15 answers were provided by the community to the question, and the thread has been viewed more than 700K times ever since.

The task can be partially specified using the straightforward IO examples listed in Table 1. Using the examples and an optional list of keywords "int array sort" as the input[2], Quebio was able to successfully find in about 12 minutes a list of 560 Java methods that at least support some of the specified IO behaviors, and the first of the returned methods actually correctly implements the desired functionality.

Figure 1 shows a method named `insertion` from the search results. The method takes an array as the input, implements insertion sort to rearrange the elements in the input array in the expected order, and returns the sorted array upon completion. Method `insertion` supports all the behaviors specified by the examples listed in Table 1. Particularly, IO examples E1, E2, and E3 are handled by the method along paths $p_1 = (2, 3, 12)$, $p_2 = (2, 3, 4, 5, 6, 10, 3, 12)$, and $p_3 = (2, 3, 4, 5, 6, 7, 8, 6, 10, 3, 12)$, respectively, where each execution path of the method is denoted using a sequence of line numbers.

[1] https://stackoverflow.com/questions/8938235/sort-an-array-in-java

[2] Query Q4 in Table 3.

```java
int[] insertion(int[] array){
    int n = array.length;
    for (int j = 1; j < n; j++) {
        int key = array[j];
        int i = j - 1;
        while (i >= 0 && array[i] > key) {
            array[i + 1] = array[i];
            i--;
        }
        array[i + 1] = key;
    }
    return array;
}
```

Figure 1: A Java method `insertion` that implements insertion sort to rearrange the elements in an integer array in increasing order. The method is returned by Quebio as a match when IO examples from Table 1 are used for the query.

Two things are particularly worth noting about the example. First, the implementation of method `insertion` involves nested loops, object member access, and array element access[3], none of which is supported by existing techniques in semantics-based code search like Satsy. Second, 6,202 methods were found with signatures compatible with the specified IO examples in this search. If all of them are to be examined by the SMT solver, Quebio can only check less than 3% of the methods in 20 minutes. Thanks to the customized keyword-based search, Quebio only needs to check 705 methods that are the most likely matches using a solver and it successfully finishes the search within less than 12 minutes.

3. The Quebio Technique

In this section, we will explain how Quebio finds methods implementing certain desired functionality based on a group of IO examples and an optional list of keywords.

3.1. Overview

Figure 2 shows an overview of Quebio. There are two phases in applying Quebio to do a code search. During offline symbolic encoding, Quebio takes a collection of Java methods and the specifications of a set of library APIs as the input, and constructs a logical formula to encode the semantics of each possible execution path of each method (Section 3.2). The methods and their corresponding logical formulas are stored into a repository, which will serve as the method pool during the online search phase. Online method search is driven by a group of IO examples specifying the expected IO behaviors of interested

[3]Array element access is handled by Quebio in the same way as how method invocations are processed.

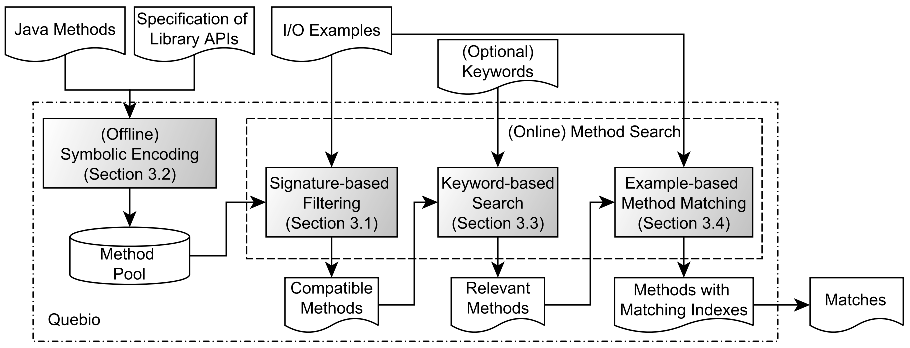
Java Methods
Specification of Library APIs
I/O Examples
(Optional) Keywords
(Offline) Symbolic Encoding (Section 3.2)
(Online) Method Search
Signature-based Filtering (Section 3.1)
Keyword-based Search (Section 3.3)
Example-based Method Matching (Section 3.4)
Method Pool
Compatible Methods
Relevant Methods
Methods with Matching Indexes
Matches
Quebio

Figure 2: Overview of The Approach

methods and an optional list of keywords summarizing the semantics of those methods. During the search, Quebio first gathers methods with signatures compatible with the types of values in the given IO examples from the pool, and then conducts a customized keyword-based search to quickly narrow down the set of possible matches (Section 3.3). Methods returned by the keyword-based search from the repository are then checked against each IO example from the query to decide how likely the expected IO behaviors are supported by each method. Only methods with their likelihood of supporting all the specified behaviors above a threshold are reported to the user (Section 3.4). The rest of this section elaborates on the steps in each phase.

3.2. Symbolic Encoding

This section describes Quebio's symbolic-execution-based approach to encoding the semantics of Java methods.

In our previous work [22], we proposed a program-transformation-based approach to the symbolic encoding of Java methods. In that approach, we build the control-flow graph (CFG) of a method, extract execution paths from the CFG, encode each path into 3-address intermediate representation, and finally translate the path into a logical formula using syntax-directed translation [24]. The approach, however, tends to produce formulas with a large number of temporary symbols and redundant assertions, leading to longer processing time during offline symbolic encoding as well as longer running time and higher failure rate during constraint solving of the online method search. We therefore design Quebio to integrate the SPF symbolic execution engine [23] for the symbolic encoding of Java methods in this work.

A symbolic-execution [25] engine generates and explores execution paths of a given program in a symbolic way. Along each path, the engine computes the values of program variables as expressions on the symbolic values of the inputs and gathers conditions under which the path will actually be executed: the expressions and conditions can be combined into a logical formula to capture the semantics of the program along the path. SPF builds on the top of the Java

PathFinder (JPF) model checker. It introduces a customized bytecode instruction factory to replace the concrete execution semantics of Java programs with symbolic execution, and it attaches a field `attribute` to each variable for storing the symbolic value of the variable. To collect the logical formulas for the paths of a method, Quebio monitors the execution of every **`return`** instruction in that method. If the instruction executes successfully, which means a feasible path is found, Quebio takes the path condition from the current execution context and conjuncts it with the expressions for the symbolic values of the method's output parameters as well as return values to get the formula encoding the semantics of the path.

A generic symbolic execution engine, however, is insufficient for the symbolic-encoding of methods in the context of example-based code search for two main reasons, both having to do with how calls to library APIs are handled during symbolic execution. First, some library method implementations mix code in Java and other programming languages like C, C++, and Assembly, while adding support for the symbolic execution of those code to an existing generic Java symbolic-execution engine can be extremely challenging. Second, the implementation of certain library APIs can be long and highly complex. Relying only on symbolic execution to discover the semantics of such APIs will most likely result in symbolic expressions and path conditions that are beyond the processing power of existing decision procedures. As a tradeoff between usability and applicability, we design Quebio to make use of existing specifications for library APIs during symbolic encoding. In particular, when encountering an invocation to a library API with specification, Quebio will treat the invocation as a blackbox and instantiate the corresponding specification based on the actual parameters used, instead of analyzing the API method implementation to derive the semantics of the invocation.

Next, we explain how we specify library APIs and how we revise SPR to utilize the specifications in symbolic encoding. While using SPR for encoding the semantics of code snippets is not new [21], we extend the existing work to support also the invocation of library APIs, enabling symbolic encoding to handle a wider range of Java code.

3.2.1. Specifications of Library APIs

In this section, we describe how we specify the semantics of library APIs to be used in symbolic encoding. Given an API method m with n $(0 \leq n)$ parameters, we manually craft a formula $\mathcal{F}_m$ that reflects the pre- and postconditions of m as described in the associated documentations. In the formula, we use `pi` and `pi`$*$ to denote the value of the i-th parameter of m $(0 \leq i < n)$ at method entry and exit, respectively, and use `r` to denote the return value of m, if any. Note that, if m is an instance method, `p0` is the receiver object (denoted using **`this`** in Java) upon which m is invoked. Also note that, Quebio handles both array element access and query of String length in the same way as it processes method invocations.

For example, Figure 3 gives the documentation and the specification, in the input format for the Z3 SMT solver [26], of method `add` from class `java`

```
/** Inserts the specified element at the
*   specified position in this list. Shifts
*   the element currently at that position
*   (if any) and any subsequent elements to
*   the right (adds one to their indices).
*   @param index index at which the
      specified element is to be inserted
*   @param element element to be inserted */
public void add(int index, E element)
```

```
(implies (and (and
  (= t1 (seq.len p0))
  (>= p1 0))
  (<= p1 t1))
  (and (and (and
  (= t2 (seq.extract p0 0 p1))
  (= t3 (seq.unit p2)))
  (= t4 (seq.extract p0 p1 (- t1 p1))))
  (= p0* (seq.++ t2 t3 t4))))
```

Figure 3: The documentation (left) and logical formula (right) for method `add` from class `java.util.ArrayList`.

`.util.ArrayList`. In the example, an arraylist is modelled as a Z3 sequence, `seq.len`, `seq.extract`, `seq.unit`, and `seq.++` are Z3 operations to get the size of a sequence, to retrieve a sub-sequence, to construct a sequence with a single element, and to get the concatenation of a list of sequences, respectively, and `tj` s are temporary variables introduced to facilitate the reference to various aspects of the parameters and the results of applying those operations ($1 \leq j \leq 4$). Note that, since the method modifies only the receiver object `p0`, but not the other two parameters `p1` or `p2`, and it returns no value, neither of `p1*`, `p2*`, or `r` appears in the specification. Here the formula specifies that, if the first parameter is a valid index for the receiver list, the method will insert the second parameter into the list at the position indicated by the first parameter, which is exactly what the method should do according to its documentation.

3.2.2. Invoke Instructions

To enable the incorporation of API method specifications into the logical formulas that encode the semantics of Java methods, we modify the interpretation of method invocation instructions, namely *invokevirtual*, *invokestatic*, *invokespecial*, and *invokeinterface*, in SPF. The new interpretation can be described using the algorithm shown in Figure 4.

Upon the invocation to a library method in symbolic encoding, Quebio first gets the current path condition *pc* (Line 4) and the method *mi* to invoke (Line 5) from the execution context. If the specification of *mi* is available (Line 6), Quebio updates the symbolic state of the program and the current path condition accordingly based on the specification (Lines 7–26). Particularly, it first gets the specification of *mi* (Line 7), gathers the symbolic values of the parameters to be used to invoke *mi* (Lines 9–11), and creates new symbolic variables to denote the values of parameters, on which method *mi* has side-effect, after the invocation (Lines 14–16). The symbolic values of the parameters before and after the invocation as well as that for the method return value (created on Line 19, if necessary) are then used to instantiate the specification of *mi* and the instantiation result is conjuncted with the current path condition to produce the updated condition (Line 20). Afterwards, Quebio suspends the symbolic execution of SPF (Line 22) and installs a callback method to be invoked

```
 1  class INVOKE... extends ...INVOKE...{
 2    Instruction execute(...){
 3      ...
 4      PathCondition pc = getPathCondition();
 5      MethodInfo mi = getMethodToInvoke();
 6      if(specRepo.containsSpecFor(mi)){
 7        SMTSpec spec = specRepo.getSpecFor(mi);
 8        int paramNum = mi.getparamNum();
 9        Expression[] param = new Expression[paramNum];
10        for(int i = 0; i < paramNum; i++){
11          param[i] = getParamSymbolicValue(mi, i);
12        Expression[] _param = new Expression[paramNum];
13        for(int i = 0; i < paramNum; i++){
14          if(mi.modifiesParam(i)){
15            _param[i] = param[i].nextVersion();
16            setParamSymbolicValue(mi, i, _param[i]);
17          }
18        }
19        Expression _return = mi.hasReturnValue() ? createSymbolicVariable() : null;
20        updatePathCondition(spec.instantiate(param, _return, _param));
21
22        suspendSymbolicExecution();
23        addMethodExitListener(mi, () -> {
24            resumeSymbolicExecution();
25            if(mi.hasReturn()) setReturnSymbolicValue(_return);
26          });
27      }
28      return super.execute(...);
29    }
30  }
```

Figure 4: Interpretation of the *invoke* instruction family. Invocations to instructions like *invokevirtual*, *invokestatic*, *invokespecial*, and *invokeinterface* are interpreted in similar ways.

when method mi returns (Line 23). The callback resumes symbolic execution (Line 24) and sets $\_result$ as the attribute of mi's return value (Line 25), if applicable. Whether a specification is available for mi or not, SPF next invokes the method concretely as usual.

For example, field `length` of a String object is modeled in Quebio as a method with the same name that can be called on String objects, and the semantics of the method in SMT is specified as `(= r (seq.len (element p0)))`, where `r` denotes the return value of the method and `p0` denotes the receiver String object. Given such a specification, symbolic execution of the statement on Line 2 of Figure 1 will lead to 1) a clause `(= stringlength_1_INT (seq.len (element a_1_ARRAY)))` being conjuncted to the current path condition and 2) the symbolic value of `n` being set to `stringlength_1_INT`. In the clause, `stringlength_1_INT` and `a_1_ARRAY` are the new symbolic variables generated by Quebio for the return value and the single parameter of the method invocation, respectively. Here, the name of each symbolic variable contains the name of its corresponding program variable, a unique version number, and its type name, concatenated using underscores.

3.3. Keyword-Based Search

Quebio looks for methods of interest in two steps: a customized keyword-based search and an example-based method matching. If the query contains a list of keywords, Quebio first calculates a textual similarity between the keywords and the summary of each method from the pool. Next, only methods with positive textual similarity will go through the second step and have their constructed logical formulas checked against the IO examples for possible matches. In such a design, the customized keyword-based search is employed to quickly narrow down the set of possible matches, so that the more memory- and computation-intensive example-based method matching only needs to be applied to a smaller amount of methods, leading to improved search efficiency.

To support this keyword-based search, during the offline encoding process, Quebio produces a natural language description for each method m by 1) treating the source code of m as a piece of text, 2) removing Java keywords from that text while preserving m's name as well as its variable names and types, 3) replacing the name of each method invoked by m with the summary of that method extracted from the Java documentation, and 4) removing stop-words like "to" and "the" from the text. Based on the description, Quebio generates a summary for the method. Specifically, it calculates the TF-IDF [27] of the words appearing in the description, sorts the words in decreasing order of frequency, and selects the 5 most frequent words as the summary of the method.

During keyword-based search, Quebio calculates the cosine similarity [28] between the query keywords and the method summary. Only methods with positive similarity values will be used as candidates in the following step of example-based method matching.

3.4. Example-based method matching

During example-based method matching, Quebio computes a matching index for each method survived the keyword-based search to indicate how likely the method's behaviors comply with the IO examples: the greater the matching index, the more likely the method supports the specified IO behaviors.

In this step, we model a method M as a triple $\langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, where $\widetilde{I}$ is the list of M's input variables, $\widetilde{O}$ the list of M's output variables, and $\widetilde{P}$ the list of paths extracted from M via symbolic encoding (Section 3.2). An IO example is modelled as a pair $\langle \widetilde{\sigma}, \widetilde{\omega} \rangle$, where $\widetilde{\sigma}$ is the list of input values and $\widetilde{\omega}$ is the list of output values. Given a variable or a value v, function τ maps v to its type. Note that function τ can be extended in a natural way so that it works also on a list of variables or values. Function ζ maps a path $p \in \widetilde{P}$ to the formula constructed for p. Given a method $M = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$ and an IO example $(\widetilde{\sigma}, \widetilde{\omega})$, we use functions α and β to bind values from $\widetilde{\sigma}$ and $\widetilde{\omega}$ to variables from $\widetilde{I}$ and $\widetilde{O}$, respectively, when their types match. In other words, we require $\tau(\widetilde{\sigma}) = \tau(\alpha(\widetilde{\sigma}))$ and $\tau(\widetilde{\omega}) = \tau(\beta(\widetilde{\omega}))$.

Next, we first explain, given a specific pair of binding functions α and β, how Quebio decides whether an IO behavior is supported by method M, then describe how the decisions based on different pairs of binding functions are summarized to produce the matching index for M.

3.4.1. Method Matching Based on Given Binding Functions

Once binding functions α and β are fixed, the question of whether the IO example $(\widetilde{\sigma}, \widetilde{\omega})$ specifies a feasible behavior of method M equates to whether formula

$$\exists p \in \widetilde{P} : \zeta(p)[\widetilde{\sigma}/\alpha(\widetilde{\sigma}), \widetilde{\omega}/\beta(\widetilde{\omega})] \tag{1}$$

is satisfiable or not [29, 30], where $\zeta(p)[\widetilde{\sigma}/\alpha(\widetilde{\sigma}), \widetilde{\omega}/\beta(\widetilde{\omega})]$ represents the result of replacing formal parameters $\alpha(\widetilde{\sigma})$ and $\beta(\widetilde{\omega})$ in $\zeta(p)$ with concrete values $\widetilde{\sigma}$ and $\widetilde{\omega}$ from IO examples, respectively. Quebio follows Procedure 1 to decide on that. In particular, Quebio constructs two constraints for each extracted path p of M (Line 1): *preCond* is constructed by only binding each value from $\widetilde{\sigma}$ to its corresponding variable from $\alpha(\widetilde{\sigma})$ in $\zeta(p)$ (Line 2), which captures whether $\widetilde{\sigma}$ constitutes an acceptable input for M along path p; *prepostCond* is constructed by further binding each value from $\widetilde{\omega}$ to its corresponding variable from $\beta(\widetilde{\omega})$ (Line 3), which captures not only whether $\widetilde{\sigma}$ constitutes an acceptable input for M along path p, but also whether the execution of M along p produces the expected output values. The procedure returns `"sat"` if *prepostCond* is indeed satisfiable (Lines 4–6). Otherwise, if all the input variables of M have been assigned values ($|\widetilde{I}| = |\widetilde{\sigma}|$, Line 7) and method M will execute along path p upon input variables $\widetilde{\sigma}$ ($\textsc{solve}(preCond) =$ `"sat"`, Line 7), whether *prepostCond* is satisfiable determines whether the IO example is supported by M along path p (Line 8). If no path from $\widetilde{P}$ can conclusively decide whether the IO example is supported or not, the procedure returns `"unknown"`. In general, Procedure 1 may return `"unknown"` for two reasons: First, the actual execution path for handling the example input may not be included in $\widetilde{P}$; Second, while the actual execution path is included in $\widetilde{P}$, the SMT solver employed is not able to determine, within the given time budget, whether the specified IO behavior is feasible for the path or not.

Consider IO example E3 for method `insertion` from Section 2. When both the input array $[2, 1]$ and the output array $[1, 2]$ are bound to parameter `array`, constraint *preCond* that Quebio constructs for path $p3$ is the conjunction of $p3$'s path condition and formula "`(= (element array_1_ARRAY) (seq.++ (seq.unit 2) (seq.unit 1)))`", while constraint *prepostCond* constructed for the same path is the conjunction of *preCond* and formula "`(= (element array_12_ARRAY) (seq.++ (seq.unit 1) (seq.unit 2)))`", where `array_1_ARRAY` and `array_12_ARRAY` are the symbolic variables introduced during symbolic encoding to represent the values of parameter `array`. Since *prepostCond* is satisfiable, Procedure 1 will return `"sat"`.

3.4.2. Calculation of matching index

The matching index between method M and a group of IO examples summarizes the path matching results derived from different binding relations between method parameters and the values in provided IO examples. Procedure 2 describes how Quebio calculates the summary. For each valid pair of functions $\langle\alpha, \beta\rangle$ that binds concrete IO values to method parameters (Line 2), Quebio first calculates the numbers of paths of M that support some IO example (N_{sat}),

Procedure 1 pathMatching

Input: Method $M = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, IO example $\langle \widetilde{\sigma}, \widetilde{\omega} \rangle$, and two functions α and β
Output: Matching result from {`"sat"`, `"unsat"`, `"unknown"`}

```
 1: for each $p \in \widetilde{P}$ do
 2:     $preCond \leftarrow \zeta(p)[\widetilde{\sigma}/\alpha(\widetilde{\sigma})]$
 3:     $prepostCond \leftarrow \zeta(p)[\widetilde{\sigma}/\alpha(\widetilde{\sigma}), \widetilde{\omega}/\beta(\widetilde{\omega})]$
 4:     if SOLVE($prepostCond$) = "sat" then
 5:         return "sat";
 6:     end if
 7:     if $|\widetilde{I}| = |\widetilde{\sigma}| \wedge$ SOLVE($preCond$) = "sat" then
 8:         return SOLVE($prepostCond$)
 9:     end if
10: end for
11: return "unknown";
```

that does not support any IO example (N_{uns}), and that we do not know whether they support any IO example or not (N_{unk}) (Lines 4–10). Then, if there is at least one IO example is supported (Line 11), Quebio calculates a matching index between M and the IO examples w.r.t. the current binding functions α and β (Line 12). The overall matching index between method M and a group of IO examples is defined as the largest matching index observed across all $\langle \alpha, \beta \rangle$ pairs (Line 13). Methods with matching indexes greater than a threshold value (0.5 by default) are reported to users in decreasing order of their matching index values. Methods with matching indexes smaller than the threshold are discarded.

3.5. Implementation

We have developed the Quebio approach into a tool, also named Quebio. Internally, Quebio encodes the semantics of discovered paths from a method in the Z3 input format [31], which extends the SMT-LIB2 standard [32], and it invokes the Z3 SMT solver [26] to determine whether a particular path supports an IO example. Based on our empirical experience, we set the timeout for each invocation to Z3 to 2 seconds, and treat all invocations not finished within that time limit as returning `"unknown"`. To strike a good balance between effectiveness and efficiency, Quebio stops the encoding of a method if 50 different feasible paths have already been discovered or 5 minutes have passed.

We devised a simple domain specific language (DSL) to specify the IO examples as part of the input queries to Quebio. Table 2 lists the grammar of the language in the ANTLR [4] syntax. A query may contain one or more examples, and each example is a mapping from a group of input values to another group of output values. For simplicity reasons, the types of values are directly

[4] https://github.com/antlr/antlr4

Procedure 2 MATCHINGINDEX

```
Input: Method $M = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, IO Examples $\{\langle \widetilde{\sigma}_i, \widetilde{\omega}_i \rangle\}_{i=1}^{m}$
Output: Matching index between $M$ and the examples
 1: $index \leftarrow 0$
 2: for each pair $\langle \alpha, \beta \rangle$ of functions binding $\widetilde{\sigma}$ and $\widetilde{\omega}$ to $\widetilde{I}$ and $\widetilde{O}$ do
 3:     $N_{sat} \leftarrow 0$, $N_{uns} \leftarrow 0$, $N_{unk} \leftarrow 0$
 4:     for each $\langle \widetilde{\sigma}_i, \widetilde{\omega}_i \rangle \in \{\langle \widetilde{\sigma}_i, \widetilde{\omega}_i \rangle\}_{i=1}^{m}$ do
 5:         $state \leftarrow$ PATHMATCHING($M$, $\langle \widetilde{\sigma}_i, \widetilde{\omega}_i \rangle$, $\alpha$, $\beta$)
 6:         if $state =$ "sat" then $N_{sat}$++
 7:         else if $state =$ "unsat" then $N_{uns}$++
 8:         else $N_{unk}$++
 9:         end if
10:     end for
11:     if $N_{sat} \neq 0$ then
12:         $curIndex \leftarrow (N_{sat} + 0.5N_{unk})/(N_{sat} + N_{uns} + N_{unk})$
13:         $index \leftarrow$ MAX($index$, $curIndex$)
14:     end if
15: end for
16: return $index$
```

Table 2: A simple domain specific language for specifying the IO examples.

examples	:	example(','example)*
input	:	value(','value)*
output	:	value(','value)*
example	:	'('input'->'output')'
value	:	primitive | container
primitive	:	INT | FLOAT | STRING | BOOLEAN
container	:	list | set | map
list	:	'['primitive(','primitive)*']'
set	:	'{'primitive(','primitive)*'}'
pair	:	primitive ':' primitive
map	:	'{'pair(','pair)*'}'
INT	:	[-]?[0-9]+
FLOAT	:	[-]?[0-9]+[.][0-9]+
STRING	:	'"'[␣-!#-~]*'"'
BOOLEAN	:	'True' | 'False'

inferred from the syntax of the values. The DSL currently supports 7 kinds of data types, including **`int`**, **`float`**, **`boolean`**, `String`, array/`List`, `Set`, and `Map`. For instance, `([3,4,2]->[2,3,4]),([7,6]->[6,7])` specifies two more IO examples that can be used as the input to resolve the query described in Section 2 with Quebio.

In view that the tasks of checking whether a path supports an IO example are independent of each other, Quebio employs multiple (4 by default) threads to conduct such checks in parallel.

4. Evaluation

We design and conduct experiments to address the following research questions:

RQ1: How *effective* is Quebio in example-based code search?

RQ2: How *efficient* is Quebio in example-based code search?

RQ3: How does the customized keyword-based search affect the effectiveness and efficiency of Quebio?

In RQ1 and RQ2, we evaluate the effectiveness and efficiency of Quebio from a user's perspective. One key difference between Quebio and existing example-based code search techniques is in that Quebio conducts a tailored keyword-based search before launching the more precise, but also more expensive, semantics-based search. We therefore investigate in RQ3 the impact of that keyword-based search on the overall effectiveness and efficiency of Quebio.

4.1. Experimental Subjects

We collect in total 47 queries from the popular Q&A website StackOverflow for programmers as our experimental subjects in three phases. Table 3 shows, for each of the 47 queries, its ID, a short DESCRIPTION, and the list of KEYWORDS as part of the input to Quebio. In phase one, we review 1000 most frequently asked questions with tag **java** on StackOverflow and gather 35 queries (Q1–Q35) that 1) ask about the implementation of certain functionalities expressible using IO examples and 2) concern only data types supported by Quebio. In phase two, to facilitate our comparison between Quebio and Satsy [21], we make sure all the 8 subject queries that were also extracted from StackOverflow and used in [21] are studied in our experiments. Since 2 of those 8 queries are already included in phase one (Q12 and Q20), we add the remaining 6 (Q36–Q41). In phase three, to make our selection of subject queries more representative of code search tasks programmers would perform in real-world scenarios, we construct more queries based on 6 questions selected from StackOverflow using the same two criteria as mentioned above but with few answers (Q42–Q47). For each query, we construct 5 IO examples to be used to drive the search with Quebio.

4.2. Measures

We distinguish two types of methods among those returned by Quebio: Methods that accept all the IO examples and have the matching index equal to 1 are called *valid* results, while methods that are manually confirmed to have indeed solved the underlying questions are called *correct* results. Moreover, we assume users are interested in finding *any* method that implements the expected behaviors correctly, and they will not check the remaining returned methods once a correct implementation is found. Correspondingly, we assess the effectiveness of Quebio on a particular query in terms of the following measures:

#R: the number of methods returned;

Table 3: Subject queries used in the experiments.

ID	DESCRIPTION	KEYWORD
Q1	Get the separate digits of an int number	separate digits number
Q2	Reverse an int value without using array	reverse int value
Q3	Concatenate int values	concatenate int value
Q4	Sort integers in increasing order	int array sort
Q5	Convert letters in a string to a number	convert string to int
Q6	Round a double to an int	round double int
Q7	Divide two integers to produce a double	divide integer
Q8	Round a double to 2 decimal places	round double places
Q9	Check if an int is prime more efficiently	check int prime
Q10	Pad an integer with zeros on the left	pad integer
Q11	Convert a String to an int	convert string to int
Q12	Check if a String is numeric	string numeric
Q13	Count the occurrence of a char in a String	count char in string
Q14	Reverse a String	string reverse
Q15	Check whether a String is not null and not empty	string null empty
Q16	Check String for palindrome	string palindrome
Q17	Concatenate two strings	string concatenate
Q18	Evaluate a math expression in String form	string evaluate
Q19	Convert an ArrayList to a String	convert list to string
Q20	Remove the last character from a String	string remove last
Q21	Test if an array contains a given value	array contain
Q22	Concatenate two arrays	array concatenate
Q23	Reverse an int array	array reverse
Q24	Remove objects from an array	array remove
Q25	Add new elements to an array	array add
Q26	Find the max value in an array of primitives	array max
Q27	Find the min value in an array of primitives	array min
Q28	Find the index of maximum from slice of an array	array max index
Q29	Sort an array	array sort
Q30	Find duplicates in an Array	array find duplicate
Q31	Sort an Array in decreasing order	array sort descending
Q32	Find the index of an element in an array	array find index
Q33	Remove repeated elements from ArrayList	array remove repeated
Q34	Intersect ArrayLists	arraylist intersection
Q35	Union of ArrayLists	arraylist union
Q36	Check if one string contains another, case insensitive	check string contain another
Q37	Capitalize the first letter of a string	capitalize first letter string
Q38	Determine if a number is positive	determine number positive
Q39	Trim the file extension from a file name	trim file extension name
Q40	Turn a string into a char	turn string to char character
Q41	Check if one string is a rotation of another	check string rotation another
Q42	Largest element in two array	largest common element array
Q43	Compare integer within a range	compare integer largest max range
Q44	Same numbers from two int arrays	common number two integer array
Q45	If the sum of any two elements equals another	check array sum two element
Q46	Search elements from a list	search arraylist match element list
Q47	Remove values greater than 100	list integer remove number grater

#V: the number of returned methods that are valid;
#C: the number of returned methods that are correct;
Rk: the top rank of correct methods in the result list.

We record the following measures regarding the efficiency of Quebio during each query as well:

#Mc: the number of methods from the pool with IO parameters that are compatible with the specified IO examples;
#Ms: the number of methods that survived keyword-based search and therefore are to be examined by method matching;

#Me: the number of candidate methods actually examined by method matching within the given time limit;

T_{kb}: the time used by Quebio on keyword-based search;

T_{eb}: the time used by Quebio on example-based method matching;

T: the total time used by Quebio to produce the results; Since the compatibility between IO examples and method signatures can be checked instantly, we have $T = T_{kb} + T_{eb}$.

All times are in seconds.

To facilitate the evaluation of the impact of the customized keyword-based search, we also count, for each query, 1) the number #Cu of correct methods that are only returned when keyword-based search is enabled or disabled, i.e., the number of returned correct methods that are unique in each case, and 2) the method survival rate in keyword-based search, which is calculated as the ratio of the number of methods surviving keyword-based seach to the number of methods with compatible parameters, i.e., #Ms/#Mc.

Note that, when keyword-based search is disabled, the number of methods from the pool with IO parameters that are compatible with the specified IO examples will be the same as the number of methods that are to be examined by example-based method matching, i.e., #Mc = #Ms.

4.3. Construction of The Method Pool

We construct the pool of methods used in the experiments by gathering relevant methods from repositories of open source Java projects hosted on the GitHub software development platform. Due to the sheer number of such repositories, it is prohibitively expensive for us to download all of them blindly. We therefore rely on the search functionality provided by the GitHub platform to narrow down the repositories that may be relevant to the subject queries, and then only download the source code of those repositories. In particular, we first use the search tool provided by GitHub to look for relevant repositories based on the keywords from the queries, as shown in column KEYWORD of Table 3. Then we gather for each query the first 5 repositories as well as the repositories containing the first 15 code snippets from the results. Next, to diversify the methods pool, we search for repositories related with container types such as `LinkedList`, `TreeSet`, and `HashMap`. We collect totally 723 distinct Java repositories in this way.

We check out the latest revisions of all these repositories and gather in total 6,487,974 methods. From these methods, we select the ones that Quebio should be able to process as our subjects using the following criteria: (1) the method is not an abstract or native method; (2) the method concerns only primitive, String, and/or container data types like array/List, Set, and Map; (3) the method has at least one input and one output variable; (4) the method compiles successfully. We are left with 24,898 methods satisfying these criteria. We manage to encode 14,792 of them in 160 hours, producing logical formulas for 246,986 paths from those methods via symbolic execution. The box plot in Figure 5 relates the number of paths extracted from a method and the time

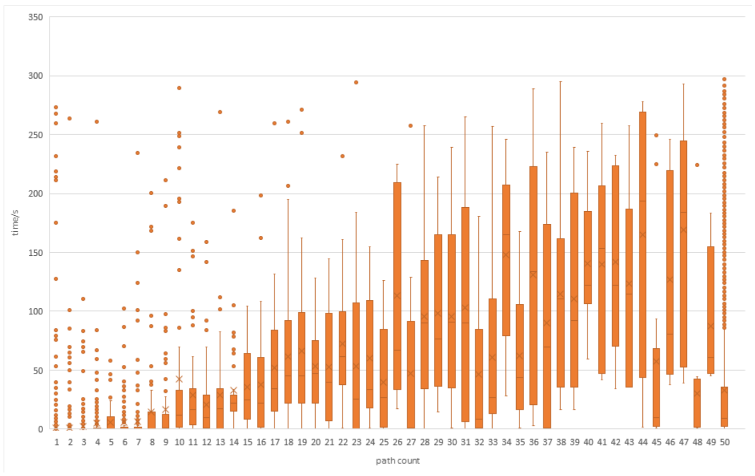

Figure 5: Relation between the numbers of paths extracted from the methods and the time required to encode the methods.

required to encode the method: each box at position x shows the distribution of the encoding time across all methods with x extracted paths. We can see from the figure that, as expected, the encoding time is proportional to the number of paths extracted from each method in general.

The reasons why the other 10,106 methods were not encoded into our local repository are grouped into 6 classes, which are shown in column REASON of Table 4. Columns #M and %M of the table show the number and the percentage of methods that failed to be encoded for a particular reason: 2,287 methods involve variables of *unsupported data types* like multidimensional arrays and nested containers; 3,457 methods contain invocations to *unsupported library methods*, i.e., library methods with no logical formulas encoding their semantics; The encoding of 2,303 methods terminated prematurely due to *Z3 errors* raised during the resolution of path constraints. Such errors often occur when the method has bit operations or the path constraint is incomplete; The encoding of 134 methods was not able to find any feasible path within the 5 minute time frame allocated for the task; 752 methods were not encoded because *runtime exceptions* like `NullPointerException` and `ArrayIndexOutOfBoundsException` were thrown by the Java PathFinder; Encoding of the *others* failed silently without generating any error message. Unsupported data type, unsupported library method, and complex constrains are the major reasons for failures in method encoding.

4.4. Experimental Protocol

Each experiment then involves running Quebio, with or without enabling the customized keyword-based search, to look for interested methods.

To answer RQ1 and RQ2, we feed the IO examples and keywords of each query to Quebio to search for matching methods from the pool. Valid results can easily be identified by checking whether their matching index is equal to 1.

Table 4: Reasons why Quebio failed to encode some methods. For each REASON, the number of methods whose encoding failed due to the reason (#M) and the percentage of those methods (%M).

REASON	#M	%M
Unsupported data type	2,287	23
Unsupported library method	3,457	34
Z3 error	2,303	23
Timeout	134	1
Runtime exception	752	7
Others	1,173	12
Total	10,106	100

To determine the correct methods in the results, two of the authors manually examine all the methods returned by Quebio. They first mark the methods that, they believe, implement the expected behaviors as correct *independently*, and then discuss each method where their independent marks differ. We conservatively only mark a method as correct if both authors reach a consensus on that in the end.

To answer RQ3, we run Quebio again on the same set of subject queries but with keyword-based search disabled—we refer to Quebio running in this mode as Quebio-. We compare the results produced by Quebio and Quebio- in terms of the measures described in Section 4.2.

All the experiments are conducted on a laptop computer with Intel Core i7-6700 CPU (3.4GHz) and 16G RAM running Ubuntu 16.04. The limit on the amount of time to spend on each experiment, whether keyword-based search is enabled or not, is set to 20 minutes, which is in alignment with the settings adopted in [19].

4.5. Experimental Results

We report on the experimental results and answer the research questions in this section.

4.5.1. RQ1: Effectiveness.

Table 5 lists, in column Quebio, for each query the number of methods returned (#R), the number of returned methods that are valid (#V), the number of returned methods that are correct (#C), and the top rank of correct methods in the result list (Rk).

Quebio was able to return result methods for all queries but Q19, and it typically returns quite a few methods for each of those queries. Such results suggest that there do exist many methods in open source software repositories implementing functionalities similar to what programmers often ask about. Quebio found over 100 matching methods for queries like Q4 and Q23. One reason for so many result methods is that the desired functionality is often (re)implemented in various open source projects. For example, many methods in the repository sort elements in a list, therefore Quebio was able to find 560 likely methods for query Q4. Another reason is related to the fact that some

Table 5: Experimental results.

ID	Quebio											Quebio-							
	#R	#V	#C	Rk	#Mc	#Ms	#Me	T	T_{kb}	T_{eb}	#Cu	#R	#V	#C	Rk	#Mc	#Me	T	#Cu
Q1	2	1	1	1	3263	57	57	61.7	10.5	51.2	0	77	71	1	6	3263	3263	609.7	0
Q2	158	40	12	1	5077	257	257	109.9	12.4	97.5	10	1230	388	2	8	5077	2195	1200.0	0
Q3	38	4	0	0	2873	76	76	65.2	10.1	55.1	0	133	21	0	0	2873	235	1200.0	0
Q4	560	27	36	1	6202	705	705	714.4	8.1	706.3	36	83	0	0	0	6202	174	1200.0	0
Q5	22	12	4	1	838	75	75	65.5	4.9	60.6	4	41	19	0	0	838	87	1200.0	0
Q6	17	0	7	10	158	24	24	9.0	5.3	3.7	1	111	4	6	105	158	137	1200.0	0
Q7	2	0	2	1	25	2	2	4.1	4.0	0.1	0	17	0	2	1	25	25	1200.0	0
Q8	15	11	2	1	549	80	80	4.7	2.3	2.4	0	77	51	2	1	549	252	1200.0	0
Q9	39	11	2	1	1173	44	44	13.4	4.5	8.9	2	146	56	0	0	1173	149	1200.0	0
Q10	1	1	1	1	39	2	2	3.5	3.4	0.1	0	17	11	1	1	39	34	1200.0	0
Q11	46	12	4	1	838	75	75	55.7	5.0	50.7	4	22	3	0	0	838	45	1200.0	0
Q12	189	76	1	76	711	193	193	61.0	1.8	59.2	1	99	43	0	0	711	103	1200.0	0
Q13	72	29	7	1	244	109	109	140.9	3.3	137.6	6	28	14	1	15	244	38	1200.0	0
Q14	111	28	1	30	755	159	159	29.5	2.0	27.5	1	103	20	0	0	755	118	1200.0	0
Q15	198	97	18	1	711	204	204	67.1	1.8	65.3	12	87	40	6	12	711	93	1200.0	0
Q16	191	77	2	64	711	195	195	64.0	1.8	62.2	2	76	35	0	0	711	77	1200.0	0
Q17	21	18	4	6	213	24	24	6.0	1.0	5.0	3	70	60	1	37	213	96	1200.0	0
Q18	68	8	0	0	838	164	164	191.7	5.0	186.7	0	11	1	0	0	838	27	1200.0	0
Q19	0	0	0	0	17	17	17	7.9	4.0	3.9	0	0	0	0	0	17	17	868.2	0
Q20	287	129	9	1	755	306	306	65.6	1.9	63.7	7	96	19	2	3	755	98	1200.0	0
Q21	40	20	23	1	347	43	43	35.9	2.7	33.2	22	14	6	1	6	347	15	1200.0	0
Q22	39	39	2	28	969	48	48	20.2	4.2	16.0	2	21	20	0	0	969	56	1200.0	0
Q23	555	13	32	1	6202	685	685	567.3	8.3	559.0	32	12	0	0	0	6202	29	1200.0	0
Q24	66	11	15	1	1317	100	100	184.7	4.2	180.5	15	4	0	0	0	1317	12	1200.0	0
Q25	33	12	1	2	3047	393	393	450.8	5.9	444.9	1	0	0	0	0	3047	12	1200.0	0
Q26	375	94	29	13	2873	730	730	1162.5	10.2	1152.3	28	9	2	1	1	2873	22	1200.0	0
Q27	380	98	20	15	2873	539	539	709.1	10.2	698.9	20	9	3	0	0	2873	21	1200.0	0
Q28	478	193	3	1	2873	805	805	1052.9	10.2	1042.7	3	4	1	0	0	2873	17	1200.0	0
Q29	559	19	37	1	6202	695	695	574.5	8.3	566.2	37	6	0	0	0	6202	16	1200.0	0
Q30	67	24	10	13	692	71	71	88.7	3.8	84.9	10	9	2	0	0	692	15	1200.0	0
Q31	558	10	9	1	6202	695	695	605.4	8.2	597.2	9	6	0	0	0	6202	15	1200.0	0
Q32	143	76	8	1	1317	178	178	209.2	7.3	201.9	8	4	2	0	0	1317	7	1200.0	0
Q33	620	9	4	74	6202	883	883	819.8	8.3	811.5	4	4	0	0	0	6202	12	1200.0	0
Q34	48	36	4	8	969	53	53	25.5	4.2	21.3	4	15	9	0	0	969	47	1200.0	0
Q35	40	39	2	38	969	47	47	18.7	4.1	14.6	2	15	15	0	0	969	49	1200.0	0
Q36	32	12	3	7	224	41	41	15.4	1.2	14.2	0	199	95	3	3	224	224	172.1	0
Q37	38	29	2	1	755	167	167	41.3	2.1	39.2	0	196	181	2	3	755	775	271.9	0
Q38	11	3	0	0	1173	12	12	8.6	4.9	3.7	0	1147	480	0	0	1173	1173	394.7	0
Q39	11	10	2	3	755	116	116	14.2	2.1	12.1	0	189	184	2	6	755	755	267.9	0
Q40	53	52	3	8	838	259	259	313.7	5.4	308.3	0	129	120	3	1	838	838	980.1	0
Q41	32	13	1	10	224	39	39	16.0	0.9	15.1	0	201	101	1	8	224	224	174.7	0
Q42	10	1	1	1	316	20	20	79.7	5.1	74.6	0	164	70	1	70	316	316	553.6	0
Q43	102	34	1	29	1996	188	188	152.2	8.7	143.5	1	571	166	0	0	1996	1009	1200.0	0
Q44	97	50	4	1	971	103	103	66.9	4.2	62.7	1	551	268	3	75	971	864	1200.0	0
Q45	115	41	1	10	693	133	133	148.7	3.8	144.9	1	271	112	0	0	693	324	1200.0	0
Q46	25	14	8	3	971	26	26	27.2	4.3	22.9	7	129	56	1	33	971	222	1200.0	0
Q47	416	5	1	4	6206	696	696	872.1	8.2	863.9	1	60	0	0	0	6206	130	1200.0	0
Tot.	6980	1538	339	-	84166	10533	10533	10022.1	244.0	9778.1	297	6463	2749	42	-	84166	14482	49892.9	0
Avg.	149	33	7	11	1791	224	224	213.2	5.2	208.0	6	138	58	0.89	20	1791	308	1061.6	0

of the IO examples used in the queries can often be interpreted in different ways, leading to partially matched methods in the results. For instance, the IO examples used to characterize query Q23 are `([1]->[1])`, `([1,2]->[2,1])`, `([1,1]->[1,1])`, `([1,3,2]->[2,3,1])`, `([1,3,2,4]->[4,2 ,3,1])`, while an *array copying* method supports the first two examples and a *sorting* method supports the first three.

For 44 out of the 47 queries, the returned methods contained at least one valid method. Overall, the numbers of valid methods are considerably smaller than those of returned methods for most queries, which is as expected, since it is more difficult for methods to support all, rather than just some, of the IO examples. Quebio returned quite a number of valid methods for queries like Q2 because it allows the values of certain parameters of a method to be decided by the solver during matching, which significantly increases the chance for IO examples to be supported by methods in unexpected ways. For example, `(123->321)` is an IO example we use for query Q2. However, a method "**`int`** `add(`

`int a,int b){return a+b;}`" in our local repository will be returned as a valid match since, when we assign 123 to a, the solver can find a model where b has value 198 to satisfy $a + b = 321$. Such over-flexibility is intrinsic to example-based code search since no rules regarding how the input and output values should be used by the methods are imposed by the queries, but its negative impact can be partially relieved by the optional keyword-based search in Quebio: Although method `add` will be a valid match for query Q2, its similarity with the keywords of the query is low, therefore the method will rank lower than many other valid methods. In fact, the first valid method for query Q2 is named `reverse` in our experiments and it indeed correctly implements the functionality as expected by Query Q2. For a few queries, e.g., Q10, Q22, Q35, Q39, and Q40, almost all returned methods are valid. Consider query Q10 for example. The IO examples used for the query include `(0,2->"00")`, `(1,2->"01")`, `(12,4->"0012")`, `(23,2->"23")`, and `(23,1->"23")`. These examples are so representative and restrictive that most methods either accept or reject all of them, which explains why we only find one likely method that is also valid.

Quebio managed to return correct methods for 43 queries, and the total number of correct methods is much smaller than that of valid methods, indicating that IO examples are indeed weak specifications. A closer look at the search results reveal that 241 of the correct methods are also valid. This leaves us with 1,297 valid but incorrect methods, which we refer to as false-positives, and 98 correct but invalid methods, which we refer to as false-negatives. In general, both false-positives and false-negatives are inevitable with example-based code search: A valid method may be incorrect if it does not satisfy the intended specification for the query which the IO examples fail to convey; A correct method may have a matching index smaller than 1 if its path to support an IO example is not extracted during symbolic encoding or the constraint solver fails to confirm the satisfiability of that path within the given time or using the allocated resources. To reduce the number of false-positives, we need to devise new, easy-to-use ways to provide auxiliary information to the search process to complement the weak specifications, or we may generate new inputs and outputs for the valid methods based on dynamic analysis and provide users with the extra behaviors to help them identify correct methods easier. To reduce the number of false-negatives, we may check the conformance between an IO example and a method by actually running the method on the given input values and comparing the actual outputs with the expected ones.

The top-ranked correct methods appear in position 1 for 22 queries, between positions 2 and 10 for 11 queries, and after position 10 for another 10 queries, which suggests that the ranking mechanism in Quebio is overall effective.

Quebio found only valid methods but no correct methods for 4 queries. We manually crafted correct implementations for all those 4 methods, constructed logical formulas to encode 145 of their paths into the method pool using Quebio, and conducted the same search again. Quebio was able to find all the manually written methods and return them as valid matches for the queries. Such results suggest the previous failures were most likely due to a lack of correct matches in our local repository of methods, rather than Quebio's inability to find such matches when they do exist.

Quebio returned 6,980 methods for 46 out of the 47 queries, among which 339 methods were correct implementations for 43 queries, averaging to 7.9 correct methods for each query.

4.5.2. RQ2: Efficiency.

Table 5 also lists, in column Quebio, for each query the number of methods from the pool with IO parameters that are compatible with the specified IO examples (#Mc), the number of methods that survived keyword-based search and are to be examined by method matching (#Ms), the number of candidate methods actually examined by method matching within the given time limit (#Me), the total time in seconds used by Quebio to produce the results (T), and the breakdown of that to the time spent on keyword-based search (T_{kb}) and example-based method matching (T_{eb}). In total, Quebio spent 9,986 seconds to resolve all the 47 queries and return the 6,980 result methods and the 339 correct matches. On average, it took Quebio 213 seconds, 1.4 seconds, and 29.5 seconds to resolve a query, to produce a result method, and to return a correct implementation, respectively.

Figure 6 shows the distribution of the total search time T for processing each query. A bar at x-coordinate a with height n indicates that, for n queries, the total search time was between $a - 100$ and a seconds. The distribution is clearly skewed to the left, indicating that the search time for most queries is limited. Particularly, only less than 100 seconds were spent on the process of 29 queries, and the search time was less than 600 seconds, or 10 minutes, for 41 queries.

Compared with keyword-based search, example-based method matching took considerably more time. Particularly, keyword-based search took between 10 and 13 seconds to finish only on 6 queries, while on all the other 41 cases, the search finished within 10 seconds. In contrast, it took Quebio longer than 60 seconds to finish example-based method matching on 25 queries, and the time was even long than 600 seconds on 6 of the queries. In total, the time spent on keyword-based search accounts for less than 2.5% of the overall search time on all the queries, which suggests example-based method matching to be the main target when improving the efficiency of the search.

The scatter chart in Figure 7 shows the relation between the number of methods examined in example-based method matching and the total time spent in each search. The figure suggests that the search time is in proportion to the number of methods examined, which suggests that a meaningful way to improve the efficiency of Quebio is to be more selective and apply the method matching step to a more restricted set of candidate methods. The optional keyword-based search step of Quebio was introduced exactly for this purpose, we study its impact on the effectiveness and efficiency of Quebio in RQ3.

On average, Quebio spent 213 seconds, 1.4 seconds, and 29.5 seconds to handle a query, to produce a result method, and to return a correct implementation, respectively.

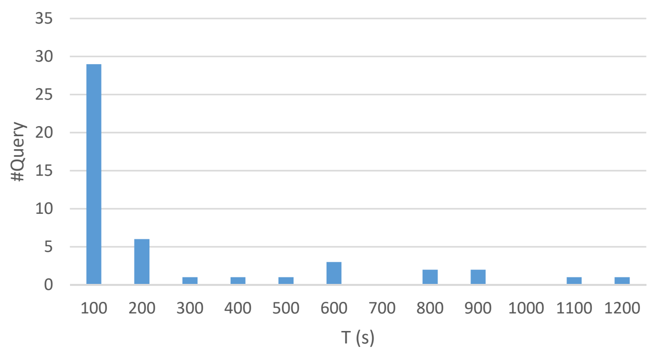

Figure 6: Distribution of the total time (T) for processing each query.

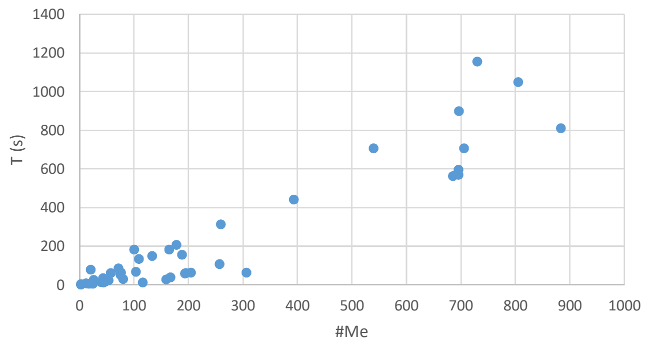

Figure 7: Relation between the overall search time (T) and the number of methods examined during example-based method matching (#Me).

4.5.3. RQ3: Usefulness of keyword-based search.

Table 5 also lists for each query the same set of measures, except #Ms, for the effectiveness and efficiency of Quebio-. To facilitate the comparison between Quebio and Quebio-, the numbers of correct methods exclusively returned by Quebio in each mode (#Cu) are provided in the same table as well.

Compared with Quebio, Quebio- returned a similar number of result methods (6,463 vs. 6,980), which contain more methods that fully support the given IO examples (2,749 vs. 1,538) but considerably fewer methods that indeed correctly implement the desired functionalities (42 vs. 339). Particularly, we make the following three observations about the two modes. First, Quebio- was not able to return more correct methods than Quebio for any query: When Quebio was not able to return any correct method for a query, neither can Quebio-; Out of the 43 queries for which Quebio found correct methods, Quebio- found the same amount of correct methods for 10 queries, found some, but fewer, correct methods for another 10 queries, and found no correct method for the remaining 23 queries. Second, when both Quebio and Quebio- are suceesul, i.e., both of them found at least one correct method, all the correct methods returned by

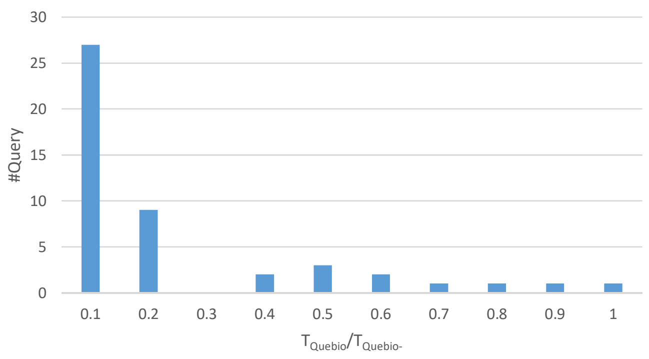

Figure 8: Distribution of the ratio between Quebio's search time and that of Quebio- ($T_{Quebio}/T_{Quebio-}$).

Quebio- are also found by Quebio: Quebio- produced no unique correct method for any query (#Cu is always 0 with Quebio-). Third, when both Quebio and Quebio- are successful, the top ranks of correct methods with Quebio were better, i.e., smaller, on all but 4 queries. Overall, the result quality of Quebio improves significantly when keyword-based search is performed before example-based method matching.

Keyword-based search also helps to improve the efficiency of method search. Figure 8 depicts the distribution of the ratio between the total search time with Quebio and that with Quebio-. A bar at x-coordinate a with height n indicates that, for n queries, the ratio of the total search time with Quebio to that with Quebio- was between $a - 0.1$ and a. Again, the distribution is clearly skewed to the left, indicating that the search time with Quebio is only a small portion of that with Quebio- for most queries. For example, for 27 queries only less than 10% of the time is needed when they are handled by Quebio rather than Quebio-, and for 41 queries less than half of the time is needed when Quebio rather than Quebio- is used to carry out the search. The total search time for the 47 queries with Quebio is merely 20% of that with Quebio-.

Quebio needs considerably shorter time to complete all the searches because keyword-based search can greatly reduce the number of methods to be examined during example-based method matching. Figure 9 shows the distribution of the method survival rate (#Ms/#Mc) across all queries. A bar at x-coordinate a with height n indicates that, for n queries, the ratio of the number of methods surviving keyword-based seach, and therefore need to be matched against the given IO examples, to the number of methods with compatible parameters was between $a - 0.1$ and a. It turns out that keyword-based search was able to filter out more than half of the compatible methods for all queries but Q19, which naturally will lead to reduced search time.

On the other hand, without the filtering via keyword-based search, Quebio- has to check the semantics of each compatible method, which may not even be feasible when the time budget allocated for the search is limited. In fact,

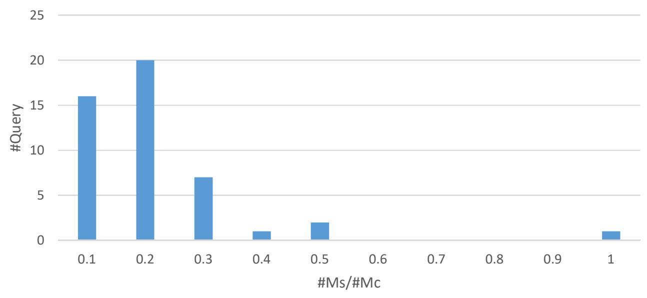

Figure 9: Distribution of the method survival rate (#Ms/#Mc) in keyword-based search with Quebio.

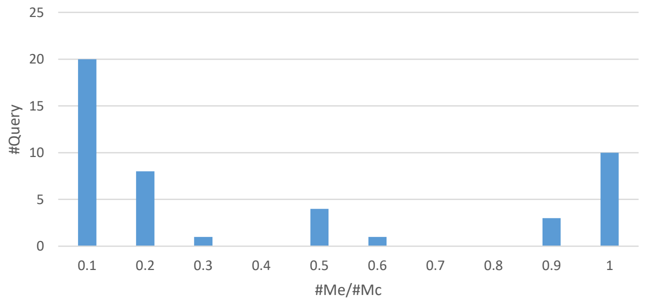

Figure 10: Distribution of the examination rate (#Me/#Mc) in example-based method matching with Quebio-.

Quebio- was *not* able to check all the compatible methods when processing most, 38 to be precise, of the subject queries. Figure 10 depicts the distribution of the examination rate (#Me/#Mc) across all queries in the experiments with Quebio-. A bar at x-coordinate a with height n indicates that, for n queries, the ratio of the number of methods examined during example-based method matching to the number of methods with compatible parameters was between $a - 0.1$ and a. Quebio- only managed to examine less than 60% of all the compatible methods within 20 minutes for 34 queries, and the examination rate was even below 10% for 20 queries. The overall low examination rate with Quebio- explains why so many correct methods were missed when keyworkd-based search was disabled.

Compared with the case where keyword-based search is disabled, Quebio with keyword-based search finds 8 times more correct methods (339 vs. 42) and correct methods to 23 more queries, while using only 20% of the time.

4.6. Comparison with Satsy

For each of the eight queries used to evaluate Satsy [21] and each of the three techniques, namely Quebio (Quebio), the round-robin mode of Satsy (Satsy RR), and the ranked mode of Satsy (Satsy Ranked), Table 6 lists the number of

Table 6: Comparison with Satsy

ID	Quebio		Satsy RR		Satsy Ranked	
	D@10	C	D@10	C	D@10	C
Q12	0	100	4.7	1.7	1	1
Q20	8	1	4.7	1.9	1	1
Q36	1	7	5.3	1.2	5.7	2.4
Q37	2	1	6.3	1	1	1
Q38	0	0	4	1	4	2.5
Q39	2	3	6.7	1	1	1
Q40	1	8	8.3	1	6.7	1.5
Q41	1	10	2.7	2.1	7.1	1.5
Average	1.9	13	4.3	1.1	2.8	1.2

desirable results among the top 10 that get returned (D@10) and the top rank of the desirable results (Rk). All the measures of Satsy are from its previous evaluation as reported in [21], and they reflect the average situation across multiple attempts on the same query using various examples.

Note that a direct comparison between these values will be inappropriate, as the two sets of experiments apply different criteria to determine whether a result is desirable or not. On the one hand, authors of [21] were interested in returned code snippets that are *useful* for solving the corresponding question, while we are more concerned with returned methods that *correctly* implement the functionality specified by the IO examples. On the other hand, it remains unknown to us what specific criteria were applied to decide the usefulness of code snippets, and it is also not clear how much effort would be needed to adjust the useful code snippets to suit a target programming task, while the correct methods returned by Quebio are most likely directly reusable. We therefore refrain from quantitatively comparing the effectiveness of Satsy and Quebio.

Qualitatively, both Quebio and Satsy can return useful results for the queries. Quebio was able to find correct implementations for 7 of the 8 queries, among which 6 queries have at least one correct method ranked among the top 10 results and 3 queries have multiple correct methods ranked among the top 10. Quebio found no matching method for query Q38, since comparing the number with zero using the greater-than operator is enough to "determine if a number is positive", and it is unnecessary to be implemented as a method. Satsy was able to find useful code snippets for all the 8 queries and rank them close to the top of the result list in either mode.

Since the Satsy tool is not publicly available for download, we are not able to apply Satsy to the other 39 queries used in our experiments and study the actual results. Nevertheless, we manually analyzed the queries and the desired functionalities and determined that most likely Satsy, in its implementation as described in [21], cannot successfully resolve 27 of the queries: 21 queries concern data types like array and List, which are not supported by Satsy, while 6 other queries demand implementations involving loops or other language features not supported by Satsy. Table 7 lists the IDs of the queries that, according to our

Table 7: Queries that, according to our analysis, can or cannot be resolved by Satsy.

RESOLVABLE?	REASON	QUERY
Yes		Q3, Q5, Q6, Q7, Q8, Q11, Q12, Q14, Q15, Q16, Q17, Q20, Q36, Q37, Q38, Q39, Q40, Q41, Q43, Q45
No	Unsurported data types	Q4, Q19, Q21, Q22, Q23, Q24, Q25, Q26, Q27, Q28, Q29, Q30, Q31, Q32, Q33, Q34, Q35, Q42, Q44, Q46, Q47
	Unsurported language constructs	Q1, Q2, Q9, Q10, Q13, Q18

analysis, Satsy can or cannot resolve.

4.7. Threats to Validity

We discuss in this section potential threats to the validity of our findings and how we mitigate them.

Threats to **construct validity** concerns whether the measures we use in the experiments reflect real-world situations. In this work, we measure the effectiveness of Quebio by looking at the number of returned methods that are correct implementations of the queries under consideration. While correct implementations of the desirable behaviors are the most useful for a user, different people may have different opinions regarding which methods are indeed correct, which imposes threats to the construct validity of our findings. To mitigate the threat, two of our authors manually checked the returned methods independently and only marked the methods as correct when a consensus on their correctness can be reached.

Threats to **internal validity** mainly have to do with the uncontrolled factors that may have influenced the experimental results. In our experiments, we devised the keywords and IO examples to be used for finding matching methods in each query, while different choices of the keywords and examples may influence the methods returned by Quebio. In the future, we plan to alter the inputs and investigate to what extent those choices affect the quality of search results. Another threat to internal validity lies in the possible bugs in the implementation of Quebio and the scripts we prepare to run Quebio and to process the search results. To minimize the threat, we reviewed our implementation carefully and ran tests to make sure the tool and the scripts function as expected.

Threats to **external validity** concerns whether the experimental findings can generalize to other situations. One threat to external validity in this work is related to the representativeness of the subject queries examined in the experiments. All subject queries used in the experiments are gathered from StackOverflow—a popular Q&A platform for programmers—and queries derived from both popular questions and unpopular questions, i.e., questions with many or no answers, are used as the subjects, which helps to ensure that the queries reflect real questions programmers ask. Overall, Quebio proved to be effective in handling the queries. Another threat to external validity concerns the limited data types that Quebio supports in its current implementation. While Quebio has the ability to handle API calls in candidate methods, the availability and the complexity of logical formulas encoding those APIs' semantics can present challenges to Quebio in achieving similar levels of effectiveness and efficiency on other queries. In the future, we plan to extend Quebio to support more data types and conduct more extensive experiments to investigate how such extension affects the performance of Quebio.

5. Related Work

This work aims to help software developers search for and reuse code on the Internet. To support searching with functional semantics, we generate path constraints for a Java method via symbolic analysis. Our work is closely related to researches from three different areas: code search, symbolic execution, and specification generation.

5.1. Code Search

By treating source code just as texts, keyword-based code search can be easily conducted on systems like Google and Bing that implement general purpose information retrieval techniques and software project hosting websites like GitHub and SourceForge. Results from such searches, however, often have limited precision due to the gap between the concepts that the code manipulates and the terms used in natural language (NL) queries as keywords to refer to those concepts.

Various technique have been developed to fill this gap and to improve the search precision. Vinayakarao et al. [7] devise the ANNE approach to generate NL annotations for source code lines based on the syntactic elements contained in those lines, while Zhang et al. [8] propose an approach to expand NL queries with semantically related API class names; Some other techniques exploit semantic information of the code to improve the precision of search results. Stolee's paper [2] compares several semantics-based search engines for code, including Exemplar [3], S6 [4], CodeGenie [5], and Sourcerer [6]. These search engines use data-flow analysis, testing, and/or AST analysis to extract semantic information from source code and utilize it to guide the search. Li et al. [10] propose the RACS approach to find JavaScript code snippets that utilize existing frameworks to implement specific features. By mining API usage patterns in the form of method call relationship graphs and abstracting NL search queries to action relationship graphs, RACS reduces the code search problem to a graph search problem; Example code fragments, instead of keywords in NL, have also been used to construct queries that can better guide the search. FaCoY [12] leverages Q&A posts from StackOverflow to build mappings between code snippets and questions. Given a code fragment as the user input, FaCoY first searches for similar code snippets and obtains their related questions. Then, it searches for similar questions and uses all related code snippets to generate an expanded code query. Finally, it searches for similar code fragments from

GitHub using the expanded code queries. DyCLINK [13] records instruction-level traces from the execution of some sample code, organizes the traces into instruction-level dynamic dependency graphs, and employs subgraph matching algorithms to detect code with similar behaviors.

With the development of deep learning, there have been attempts to use deep neural network for code search. For example, Gu et al. [16] develop the CODEnn system that trains a deep neural network to embed a code snippet and its NL description into two similar vectors. The network is then used to retrieve code snippets related to an NL query according to their vectors. Zhao et al. [17] propose the DeepSim technique where the control flow and data flow of a piece of code is encoded into a semantic matrix and a deep learning model is developed to measure code functional similarity based on this representation. Nguyen et al. [18] develop the D2Vec neural network model to compute the vector representations for the APIs. The model maintains the global context of the current API topic under description and the local orders of words and APIs in the text phrases, which helps to capture the semantics of API documentation.

Based on the insight that code snippets returned for a query tend to serve as inspirations or triggers for further queries, Martie et al. [14] build CodeExchange (CE) and CodeLikeThis (CLT) to support incremental code search. CE allows programmers to use characteristics to filter search results, while CLT enables programmers to find code that is analogous to the previous search results. Sivaraman et al. [15] develop the ALICE interactive technique that searches for code with or without features specified in a query language. By adding or removing the specified features, users are able to refine a searche easily in iterations. Instead of always returning existing code as search results, the SWIM technique [11] developed by Raghothaman et al. combines code search and program synthesis to suggest code snippets for API-related NL queries. SWIM first searches for APIs relevant to the queries using a general-purpose search engine, and then finds code fragments corresponding to those APIs from GitHub. Next, it trains a probabilistic model to describe usage patterns from these APIs and code fragments, and uses the probabilistic model to synthesize an idiomatic piece of code to describe the usage of the APIs.

Our approach Quebio is closely related to Satsy [19, 20, 21], which pioneered the idea of using IO examples and SMT solver to find source code snippets with specific IO behaviors. Quebio advances the state-of-the-art of example-based code search by supporting more language features and more data types, which significantly enhances the applicability of the techniques. Quebio is also the first technique to integrate a customized keyword-based search with the example-based code search, so that the overall approach strikes a good balance between effectiveness and efficiency.

5.2. Symbolic Execution

Symbolic execution [25] is a program analysis technique that generates test data for a program automatically by solving its path constraints. A recent survey [33] points out that all existing symbolic execution tools such as KLEE [34] and Symbolic JPF [23] suffer from three fundamental problems that limit their effectiveness on real-world software. The first problem is path explosion: A program may have such a large number of paths that it becomes impossible to enumerate those paths in a reasonable time. The second problem is path divergence: Some parts of a program may be in binary form or different programming languages, posing challenges to the symbolic executor in understanding the semantics of those parts and constructing the path conditions. The third problem is that many path constraints are too complex to solve. For example, it is difficult for existing constraint solvers like Z3 [26] to solve complex constraints involving nonlinear operations.

Since Quebio employs an symbolic executor to encode the semantics of methods, naturally it is influenced by the same problems from which symbolic execution tools suffer. Quebio, however, remains a rather promising approach to example-based code search for two reasons. First, Quebio seldom needs to enumerate a large number of execution paths of a target method. On the one hand, compared with whole programs, methods are typically less complex and have fewer different execution paths; On the other hand, in example-based code search, users often resort to small examples that are easy to write and comprehend when constructing queries, and those small examples are typically processed by methods along short execution paths. As a result, while Quebio only extracts relatively short execution paths of methods via symbolic execution during the offline encoding phase, the paths are often enough to decide whether the IO examples are supported or not. Second, the problems of path divergence and constraint complexity become significantly more severe when the code to be symbolically executed contains invocations to other methods, while Quebio alleviates the problems by treating library APIs as blackboxes and incorporating their semantics by instantiating their specifications at the call site. In this way, Quebio greatly reduces the number of cases where it needs to symbolically execute code in other languages or produces complex constraints because of method inlining.

5.3. Specification Generation

Various mechanisms have been developed in the past to specify the semantics of code. For example, the Java Modeling Language (JML) [35, 36] can be used to specify assertions, pre- and postconditions, and invariants of Java classes and interfaces. Manually constructing the specifications, however, can be very tedious and time-consuming, specification generation therefore has attracted much attention among researchers in the past few years. Daikon [37, 38] is a dynamic invariant inference technique, which generates pre- and postconditions of API methods by dynamically running them, collecting the execution traces, and using machine learning algorithms to analyze the traces. The inferred specifications, however, may overfit the executions analyzed during the inference and be incorrect. Zhai et al. [39] propose a technique that analyzes the documentation written in natural language for API methods and constructs equivalent, but simpler, implementations in Java as executable specifications for the methods. When used in symbolic execution, the executable specifications can significantly simplify the constraints produced.

Quebio relies on the availability of specifications for library APIs to encode the semantics of target methods efficiently, and high quality specifications automatically generated by tools will enable Quebio to be applied to support a wider range of searches.

6. Conclusions

Code search plays an important role in modern software engineering. With the amount of source code available on the Internet increasing, semantics-based code search can help software developers retrieve and reuse existing code more effectively. In this paper, we present an approach to example-based search for Java methods. We encode the semantics of Java methods along various execution paths into constraints and leverage an SMT solver to check whether the expected IO behaviors are supported by the methods. Our approach advances the state-of-the-art in example-based code search and is able to handle more language features like method invocation and loop and more data types like array/List, Set, and Map; Our approach also integrates a customized keyword-based search to quickly prune out methods that are less likely matches, so that the overall search efficiency is improved. We conducted experiments on 47 queries based on real-world questions from programmers to evaluate our approach. The approach was able to find correct methods from a pool of 14,792 candidates for 43 of the queries, spending on average around 213 seconds on each query. Such results suggest our approach is both effective and efficient.

[1] S. E. Sim, M. Umarji, S. Ratanotayanon, C. V. Lopes, How well do search engines support code retrieval on the web?, ACM Trans. Softw. Eng. Methodol. 21 (1) (2011) 4:1–4:25.

[2] C. Sadowski, K. T. Stolee, S. Elbaum, How developers search for code: A case study, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 191–201.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, Q. Xie, Exemplar: A source code search engine for finding highly relevant applications, IEEE Transactions on Software Engineering 38 (5) (2012) 1069–1087.

[4] S. P. Reiss, Semantics-based code search, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 243–253.

[5] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, C. V. Lopes, Codegenie: Using test-cases to search and reuse source code, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 525–526.

[6] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, C. Lopes, Sourcerer: A search engine for open source code supporting structure-based search, in: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 681–682.

[7] V. Vinayakarao, A. Sarma, R. Purandare, S. Jain, S. Jain, Anne: Improving source code search using entity retrieval approach, in: Proceedings of the Tenth ACM International Conference on Web Search and Data Mining - WSDM '17, ACM Press, New York, New York, USA, 2017, pp. 211–220.

[8] F. Zhang, H. Niu, I. Keivanloo, Y. Zou, Expanding queries for code search using semantically related api class-names, IEEE Transactions on Software Engineering 44 (11) (2018) 1070–1082.

[9] Z. Lin, Y. Zou, J. Zhao, B. Xie, Improving software text retrieval using conceptual knowledge in source code, ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (2017) 123–134.

[10] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, H. Mei, Relationship-aware code search for javascript frameworks, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, ACM Press, New York, New York, USA, 2016, pp. 690–701.

[11] M. Raghothaman, Y. Wei, Y. Hamadi, Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 357–367.

[12] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, Y. L. Traon, FaCoY - a code-to-code search engine, in: Proceedings of the 40th International Conference on Software Engineering - ICSE '18, ACM Press, New York, New York, USA, 2018, pp. 946–957.

[13] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, T. Jebara, Code relatives: detecting similarly behaving software, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, ACM Press, New York, New York, USA, 2016, pp. 702–714.

[14] L. Martie, A. van der Hoek, T. Kwak, Understanding the impact of support for iteration on code search, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017, ACM Press, New York, New York, USA, 2017, pp. 774–785.

[15] A. Sivaraman, T. Zhang, G. Van den Broeck, M. Kim, Active inductive logic programming for code search, in: 2019 IEEE/ACM 41st International

Conference on Software Engineering (ICSE), Vol. 2019-May, IEEE, 2019, pp. 292–303.

[16] X. Gu, H. Zhang, S. Kim, Deep code search, in: Proceedings of the 40th International Conference on Software Engineering - ICSE '18, ACM Press, New York, New York, USA, 2018, pp. 933–944.

[17] G. Zhao, J. Huang, Deepsim: deep learning code functional similarity, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, New York, New York, USA, 2018, pp. 141–151.

[18] T. Nguyen, N. Tran, H. Phan, T. Nguyen, L. Truong, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, Complementing global and local contexts in representing api descriptions to improve api retrieval tasks, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, New York, New York, USA, 2018, pp. 551–562.

[19] K. T. Stolee, Solving the search for source code, Ph.D. thesis, University of Nebraska - Lincoln (2013).

[20] K. T. Stolee, S. Elbaum, D. Dobos, Solving the search for source code, ACM Trans. Softw. Eng. Methodol. 23 (3) (2014) 26:1–26:45.

[21] K. T. Stolee, S. Elbaum, M. B. Dwyer, Code search with input/output queries, Journal of Systems and Software 116 (C) (2016) 35–48.

[22] R. Jiang, Z. Chen, Z. Zhang, Y. Pei, M. Pan, T. Zhang, Semantics-based code search using input/output examples, in: Proceedings of the IEEE 18th International Working Conference on Source Code Analysis and Manipulation, SCAM '18, 2018, pp. 92–102.

[23] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing nasa software, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 15–26.

[24] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley, Boston, MA, USA, 2006.

[25] L. A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. Softw. Eng. 2 (3) (1976) 215–222.

[26] L. de Moura, N. Bjørner, Z3: An efficient smt solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.

[27] K. S. Jones, A statistical interpretation of term specificity and its application in retrieval, Journal of Documentation 28 (1972) 11–21.

[28] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, 3rd Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[29] S. Jha, S. Gulwani, S. A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, New York, NY, USA, 2010, pp. 215–224.

[30] O. Polozov, S. Gulwani, Flashmeta: A framework for inductive program synthesis, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, ACM, New York, NY, USA, 2015, pp. 107–126.

[31] Z3-guide.
URL `https://rise4fun.com/Z3/tutorial/guide`

[32] C. Barrett, P. Fontaine, C. Tinelli, The smt-lib standard: Version 2.6, Tech. rep., Department of Computer Science, The University of Iowa, available at `www.SMT-LIB.org` (2017).

[33] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, An orchestrated survey on automated software test case generation, Journal of Systems & Software 86 (8) (2013) 1978–2001.

[34] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 209–224.

[35] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of jml tools and applications, International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232.

[36] P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond assertions: Advanced specification and verification with jml and esc/java2, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 342–363.

[37] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, in: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, ACM, New York, NY, USA, 1999, pp. 213–224.

[38] C. Csallner, N. Tillmann, Y. Smaragdakis, Dysy: Dynamic symbolic execution for invariant inference, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 281–290.

[39] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, F. Qin, Automatic model generation from documentation for java api functions, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 380–391.