Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2018-IC-002

2018-IC-002

Semantics-Based Code Search Using Input/Output Examples

Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei , Minxue Pan, Tian Zhang

International Working Conference on Source Code Analysis and Manipulation 2016

2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation

Semantics-Based Code Search Using Input/Output Examples

Renhe Jiang[†], Zhengzhao Chen[†], Zejun Zhang[†], Yu Pei[§], Minxue Pan[†*], Tian Zhang[†*]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[§]Department of Computing, The Hong Kong Polytechnic University
{131220105, 141220018, MG1733092}@smail.nju.edu.cn, csypei@comp.polyu.edu.hk, {mxp, ztluck}@nju.edu.cn

***Abstract*—As the quality and quantity of open source code increase, semantics-based code search has become an emerging need for software developers to retrieve and reuse existing source code. We present an approach of semantics-based code search using input/output examples for the Java language. Our approach encodes Java methods in code repositories into path constraints via symbolic analysis and leverages SMT solvers to find the methods whose path constraints can satisfy the given input/output examples. Our approach extends the applicability of the semantics-based search technology to more general Java code compared with existing methods. To evaluate our approach, we encoded 1228 methods from GitHub and applied semantics-based code search on 35 queries extracted from Stack Overflow. Correct method code for 29 queries was obtained during the search and the average search time was just about 48 seconds.**

***Index Terms*—code query, symbolic analysis, SMT solver**

I. Introduction

Code search has become an important assistance for software development. The case study [1] shows that developers search for source code very frequently, conducting an average of five search sessions with 12 total queries each workday. As the quantity and quality of open source code increase, code search also becomes a viable and competitive way to do business [2].

There are some widely used code search platforms, such as Google, SourceForge, and GitHub, which index projects and source code based on the textual information and the additional description given by developers. These code search platforms perform well on searching for open source projects using keywords, however, they usually find too many useless results since keywords cannot specify semantic specifications completely. There are other works trying to support semantics-based search for source code via software analysis and testing technologies [1], such as Exemplar [3], S6 [4], CodeGenie [5], and Sourcerer [6].

Our work is inspired by a novel semantic search technology, which uses input/output (I/O) examples as query specification and uses SMT solver to search for source code. The technology is proposed and implemented as a search engine called Satsy [7], [8]. Satsy encodes the source code into path constraints via offline symbolic analysis, as is done in symbolic execution [9] and program synthesis [10], [11]. A piece of code is considered consistent with an I/O example if it can transform the input to the output. In the online query phase, Satsy combines the I/O examples with path constraints and solves them via SMT solver to find source code consistent with the given I/O examples.

There are three advantages in this search technology. First, an I/O example is a lightweight specification that is easy to comprehend and write. Second, constraints can roughly reflect the program behavior ignoring some complex but irrelevant details. Third, even if we only provide concrete values to parts of variables in a program snippet, leaving the other values unknown, the constraint can still be solved.

Although Satsy has been applied to search for code in Java String library, Yahoo! Pipes mashup, and SQL select statement in previous studies [7], [8], it is not generalized to daily code search activities. Satsy can only process loop-free programs and handle String API invocations. Besides, it depends on Symbolic PathFinder (SPF) [9], a symbolic execution tool, to gather feasible paths and obtain constraints. In order to use SPF, one must ensure that the code is runnable and configure the entry point for symbolic execution, which is difficult to automate. Additionally, SPF may suffer from path explosion, path divergence, and complex constraint [12]. When dealing with API invocations, the developers of Satsy only prepared constraint templates for String class so that when a String API invocation occurs, instead of running the invoked method, Satsy renders its constraint template. However, most Java code contains loops, arithmetic, boolean expressions, and other method invocations besides String APIs, which also need to be supported.

Focusing on the limitations discussed above, we improve the existing technology in three aspects. First, we use a control flow graph (CFG) to extract paths from source code and filter out infeasible paths after encoding them into path constraints. Second, we support encoding methods with arithmetic, boolean expressions and user defined method invocations into path constraints. Third, we write constraint templates for container classes such as List, Set, and Map in JDK.

In our experiment, we collected 35 questions from Stack Overflow and their I/O queries, and built a local repository, which contains 1228 methods from 333 projects on GitHub and their path constraints. We evaluated our approach by applying the proposed semantics-based code search on our

This work is supported by National Natural Science Foundation (Grant Nos. 61690204, 61472180 and 61502228) of China, and partly supported by the Hong Kong RGC General Research Fund (GRF) PolyU 152703/16E and The Hong Kong Polytechnic University internal fund 1-ZVJ1 and G-YBXU.

*Corresponding author.

2470-6892/18/$31.00 ©2018 IEEE
DOI 10.1109/SCAM.2018.00018

local repository to answer the queries. The experimental result shows that correct solutions were found for 29 queries and the average search time was just about 48 seconds.

The primary contributions of our work are as follows:

- A formal description of the semantics-based code search using I/O examples.
- An approach that supports the semantics-based code search for general Java code.
- An implementation and evaluation of our approach.

The paper is organized as follows. Section II gives the overview and an illustrative example of our work. Section III describes our symbolic encoding approach. We formally define the code search problem and describe our query approach in Section IV. The implementation and evaluation are presented in Section V and Section VI respectively, followed by related work in Section VII and the conclusion in Section VIII.

II. Overview

We present the overview of our approach and an example for illustration in this section. Figure 1 shows the overview of our approach, which includes offline symbolic encoding and online query.

In the symbolic encoding phase, given a Java method, we first extract paths from the method body by traversals on the CFG. Then we reform the paths to Static Single Assignment (SSA) forms and encode them into symbolic constraints via syntax-directed translation [13]. We write constraint templates for String and container APIs according to their alternative specification (e.g. documentation), which will be used to handle method invocation statements. Finally, we solve the path constraints to find out feasible paths and store them into a constraint database.

In the query phase, given a set of I/O examples as the query specification, we first find the candidate methods that have I/O variables whose types are the same as the values in examples. Then we the bind I/O values in each example to every path constraint of the candidate methods for solving. Finally, we rank the candidate methods according to the solving results and return them to users.

We use an example from Stack Overflow to illustrate our approach.

Example 1 (Sort Array). The goal here is to sort an integer array in order from the lowest to highest value. The question can be described with the following I/O examples, which are used as queries.

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]*

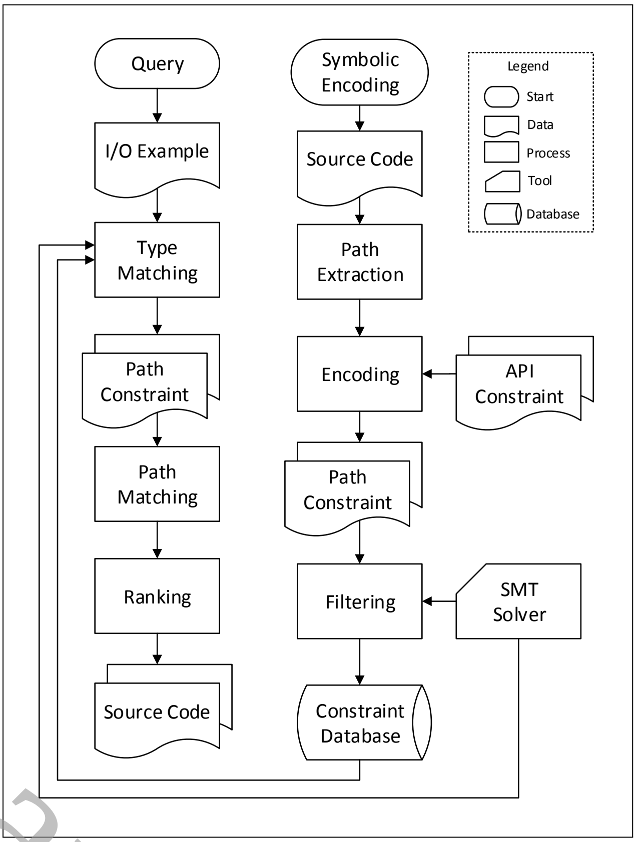


Fig. 1. Overview of the approach

```
1   int[] insertion(int[] array){
2       int n=array.length;
3       for (int j=1; j < n; j++) {
4           int key=array[j];
5           int i=j - 1;
6           while (i >= 0 && array[i] > key) {
7               array[i + 1]=array[i];
8               i--;
9           }
10          array[i + 1]=key;
11      }
12      return array;
13  }
```

Fig. 2. Insertion Sort

Figure 2 is an method named *insertion* that satisfies the queries. The input and output variables of the method *insertion* are both $array$. There are infinite paths of the method and each I/O example can be processed by one path. Since we have encoded a subset of these paths into path constraints, we bind the I/O example to each path constraint and check whether the path can process it. For example, we use a sequence of line numbers to represent a path, E1 can be processed by path $p_1 = (2, 3, 12)$, E2 can be processed by $p_2 = (2, 3, 4, 5, 6, 10, 12)$, and E3 can be processed by $p_3 = (2, 3, 4, 5, 6, 7, 8, 10, 12)$. Specifically, as we bind E3 to the path constraint of p_3 (shown in Figure 3), the SMT solver will return sat, which means p_3 can process E3. The details will be shown at the end of Section III and in Section IV.

III. Symbolic Encoding

In this section, we describe the approach that encodes a method in a Java project symbolically. The type of variables appearing in the method is restricted to primitive, array, String, and container classes. For brevity, we treat an array as a special container class List. The *[]=* (write a value by index), *=[]* (read a value by index), and *length* operators of an array are viewed as *set*, *get*, and *size* method invocations of a List, respectively. Table I shows the abstract syntax (that refers to paper [14]) of our supporting methods where *cons* represents the constant value, $aop(\boldsymbol{e})$ and $bop(\boldsymbol{e})$ are arithmetic and comparison operations on operands $\boldsymbol{e}$, such as $e_1 + e_2$ and $e_1 \geq e_2$. We simply refer $m(\boldsymbol{e})$ to the method invocation whose *this pointer* is hidden in $\boldsymbol{e}$. The inputs of a method are its parameters. The outputs of a method contain its return value and some inputs that may be modified because of the side-effect.

TABLE I
Abstract Syntax

(variable)	$a \in \widetilde{V} = \{a_1, a_2, \cdots\}$
(method)	$m \in \widetilde{M} = \{m_1, m_2, \cdots\}$
(expression)	$e \to cons \mid a \mid aop(\boldsymbol{e}) \mid m(\boldsymbol{e})$
(boolean expression)	$b \to$ **true** $\mid$ **false** $\mid e \mid bop(\boldsymbol{e}) \mid$
	$\neg b_1 \mid b_1 \wedge b_2 \mid b_1 \vee b_2$
(body)	$s \to a = e \mid s_1 s_2 \mid$ **if** $b\ s_1$**else** $s_2 \mid$
	while $b\ s_1 \mid$ **skip** $\mid$ **break** $\mid$ **continue**
(input)	$i \in \widetilde{I} \subseteq \widetilde{V}$
(output)	$o \in \widetilde{O} \subseteq \widetilde{V}$

Procedure 1 shows our encoding approach where line 2 extracts finite paths from the method body, line 4 encodes them into constraints in form of SSA, and line 5 filters out infeasible paths. The details will be discussed in the following sections.

Procedure 1 SymbolicEncoding

```
Input: Method body $S$, Maximum length $N$
Output: A set $\widetilde{P}_{sym}$ of path constraints
1: $\widetilde{P}_{sym} = \emptyset$
2: while there exists untreated path $p$ from entry to exit in
   CFG($S$) $\wedge$ $|p| \leq N$ do
3:     $P$ = PathConstruct($p$)
4:     $P_{sym}$ = SSAEncode($P$)
5:     if solve($P_{sym}$) = $sat$ then
6:         $\widetilde{P}_{sym} = \widetilde{P}_{sym} \cup \{P_{sym}\}$
7:     end if
8: end while
```

A. Path Extraction

Our path extraction approach is based on a CFG. In practice, we use Soot [15] to construct the CFG for a method that consists of basic blocks and directed edges. The basic block at each branch point will end with a boolean expression as the branch condition. From this basic block, there are two direct edges leading to the true and false branch. There always exists an entry block as the entry of the method and at least one exit block as the exit of the method. A path (which may be infeasible) of a method body S in the CFG starts from the entry and walks along the direct edges until reaching an exit. Although there may exist infinite paths in S because of the loops, in practice, we only extract a finite subset of paths by fixing the maximum number of basic blocks in each path explored in the CFG.

TABLE II
Semantic Rule

Production	Semantic Rule
a	$a.sym := a@a.version$
$e \to cons$	$e.sym := cons.literal$
$e \to a$	$e.sym := a.sym$
$e \to aop(\boldsymbol{e})$	$e.sym := (aop\ \boldsymbol{e}.sym)$
	e.g. $(e_1 + e_2).sym := (e_1.sym + e_2.sym)$
$e \to m(\boldsymbol{e})$	refer to III-C
$b \to$ **true**	$b.sym := True$
$b \to$ **false**	$b.sym := False$
$b \to e$	$b.sym := (e.sym = 1)$
$b \to bop(\boldsymbol{e})$	$b.sym := (bop\ \boldsymbol{e}.sym)$
	e.g. $(e_1 \leq e_2).sym := (e_1.sym \leq e_2.sym)$
$b \to \neg b_1$	$b.sym := (\neg\ b.sym)$
$b \to b_1 \wedge b_2$	$b.sym := (b_1.sym \wedge b_2.sym)$
$b \to b_1 \vee b_2$	$b.sym := (b_1.sym \vee b_2.sym)$
$s \to a = e$	$a.version$++
	$s.sym := (a.sym = e.sym)$
$s \to s_1 s_2$	$s.sym := (s_1.sym \wedge s_2.sym)$

After exploring a path from the CFG, we transform it by connecting its basic blocks together and judging whether a branch condition should be true or false according to the branch choice. This transformation is done in PathConstruct of Procedure 1 and the path is shown as a sequence of boolean expressions b and assignment statements $a = e$.

B. Encoding Rules

In this section, we describe how to encode a path into constraints represented in SSA form. The reason that we use SSA to represent path constraints is to record the state transformation caused by each statement. Specifically, we rename each variable with $name@version$. The version of a variable will be updated when its value is modified, such as it appears in the left hand side of an assignment statement or it is the parameter transmitted to a method invocation with side-effect. Here is an example:

$$\cdots a \xrightarrow{a=e} a@1 \cdots a@1 \xrightarrow{m(a,\boldsymbol{e})} a@2 \cdots$$

where $a@1$ reads "a after the first modification". Note that $m(a, \boldsymbol{e})$ means a is modified after invoking m.

We use syntax-directed translation to translate a single path into SSA constraints. Table II gives the syntax-directed definitions associated to the grammar shown in Table I. Each production in Table I can be translated by a simple semantic rule except $e \to m(\boldsymbol{e})$, which will be discussed in III-C separately.

C. Encoding a Method Invocation

In this section, we describe the encoding rule of a method invocation. Given a method declaration $m(\boldsymbol{a}, \boldsymbol{b})$ with body S and return value r, we refer to $\boldsymbol{a}$ as the parameters on which m has a side-effect (e.g. *this pointer*) and $\boldsymbol{b}$ as those that would not be modified by m. Thus, the inputs of m are $\boldsymbol{a}_{in}, \boldsymbol{b}$ and the outputs are $r, \boldsymbol{a}_{out}$. We first encode S into path constraints $\widetilde{P}_{sym} = \textsc{SymbolicEncoding}(S)$, then we construct the constraint template as follows:

$$\bigvee_{P_{sym}\in\widetilde{P}_{sym}} ((?\boldsymbol{a}_{in} = \boldsymbol{a}@1) \wedge (?\boldsymbol{b} = \boldsymbol{b}@1) \wedge P_{sym} \wedge \\ (?r = P_{sym}[r]) \wedge (?\boldsymbol{a}_{out} = P_{sym}[\boldsymbol{a}]))$$

where $P_{sym}[r]$ and $P_{sym}[\boldsymbol{a}]$ are the symbols with the latest version of r and $\boldsymbol{a}$ in P_{sym}.

The constraint template leaves placeholders that start with “?” to be rendered by the invocation context. When we encounter a statement $r = m(\boldsymbol{a}, \boldsymbol{e})$ during the encoding process, we

1) bind $\boldsymbol{a}.sym$ to $?\boldsymbol{a}_{in}$ and $\boldsymbol{e}.sym$ to $?\boldsymbol{b}$ as the initial value
2) $\boldsymbol{a}.version$++, $r.version$++
3) bind the new $\boldsymbol{a}.sym$ to $?\boldsymbol{a}_{out}$ and $r.sym$ to $?r$ as the return value

Note that if $\boldsymbol{a}.sym$ and $\boldsymbol{b}.sym$ do not satisfy any path constraint in $\widetilde{P}_{sym}$, this method invocation is recognized as infeasible.

In practice, the encoding rules described above are not always applicable. Many methods are hard to be analyzed such as that from JDK or third party APIs in Java. First, their source code may be hard to acquire. Second, their invocation chains are too deep to extract a feasible path. Third, they may call native code that we cannot analyze. Fourth, constraints generated from source code are usually too complex to solve. These challenges are the same as the fundamental problems in symbolic execution work that have been only partially addressed [12].

Instead of analyzing source code, we can also infer constraint based on alternative specifications such as the documentation [16] of APIs. For example, we observe that String and container classes appear frequently in the Java code. The constraints for String class have been given in the previous work [8]. To increase the applicability of our approach, we write constraints for container classes such as List, Set, and Map based on their documentation. For example, Table III shows the Javadoc [17] of the method *add* declared in *java.util.ArrayList* and its constraint template we write according to the documentation.

We explain the detail of the constraint template of *add* shown in Table III, where $seq.*$ is the Z3 [18] build-in support for the sequence constraint. We refer to $this@1$ as the List before invoking the method *add* and $this@2$ as the List after the invocation. The constraint shows that the precondition is that $index$ must be greater than 0. The postcondition is that the subsequence from the start to the position $index$ of $this@1$ equals to that of $this@2$, the subsequence of $this@1$ from

TABLE III
DOCUMENTATION AND CONSTRAINT OF METHOD ADD

```
java.util.ArrayList
/**
* 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)
```

$$\begin{aligned}
&\ (?a_{in} = this@1) \\
&\wedge (?b_1 = index@1) \\
&\wedge (?b_2 = element@1) \\
&\wedge (t_1@1 = (seq.len\ this@1)) \\
&\wedge (index@1 \geq 0) \\
&\wedge (index@1 \leq t_1@1) \\
&\wedge (t_2@1 = (seq.extract\ this@1\ 0\ index@1)) \\
&\wedge (t_3@1 = (seq.unit\ element@1)) \\
&\wedge (t_4@1 = (seq.extract\ this@1\ index@1\ t1@1)) \\
&\wedge (this@2 = (seq.{+}{+}\ t_2@1\ t_3@1\ t_4@1)) \\
&\wedge (?a_{out} = this@2)
\end{aligned}$$

```
;Input#array_0, Output#array_2
(let ((a!1 (seq.++
  (seq.extract array_0 0 t8_0)
  (seq.++ (seq.unit t11_0)
    (seq.extract array_0
      (+ t8_0 1) (seq.len array_0)))))
(a!2 (seq.++
  (seq.extract array_1 0 t16_0)
  (seq.++ (seq.unit key_0)
    (seq.extract array_1
      (+ t16_0 1) (seq.len array_1))))))
(and (= n_0 (seq.len array_0))
(= j_0 1)
(< j_0 n_0)
(= t0_0 array_0)
(= t1_0 j_0)
(= (seq.unit key_0) (seq.at array_0 t1_0))
(= t2_0 (- j_0 1))
(= i_0 t2_0)
(>= i_0 0)
(= t3_0 array_0)
(= t4_0 i_0)
(= (seq.unit t5_0) (seq.at array_0 t4_0))
(> t5_0 key_0)
(= t6_0 array_0)
(= t7_0 i_0)
(= t8_0 (+ t7_0 1))
(= t9_0 array_0)
(= t10_0 i_0)
(= (seq.unit t11_0) (seq.at array_0 t10_0))
(= a!1 array_1)
(= t12_0 i_0)
(= t13_0 (+ t12_0 (- 1)))
(= i_1 t13_0)
(not (>= i_1 0))
(= t14_0 array_1)
(= t15_0 i_1)
(= t16_0 (+ t15_0 1))
(= a!2 array_2)
(= t17_0 j_0)
(= t18_0 (+ t17_0 1))
(= j_1 t18_0)
(not (< j_1 n_0))))
```

Fig. 3. Path Constraint of Insertion Sort

position $index$ to the end equals to that of $this$@2 from $index + 1$ to the end, and the element of $this$@2 at position $index$ equals to $element$.

As for EXAMPLE 1 in Section II, Figure 3 shows the path constraint we generate for path p_3. From line 12 to 14, the constraint indicates that the value of n equals to the length of $array$ (line 12) and the value of j equals to 1 (line 13). Next comes the first path condition $j < n$ (line 14), which leads the path to the *for* statement. From line 15 to 24, the constraint indicates that the value of *key* equals to $array[j]$ (line 15∼17) and the value of i equals to $j-1$ (line 18∼19). Then the second path condition $i >= 0 \wedge array[i] > key$ is acknowledged by line 20 and line 24, which leads the path to the *while* statement. From line 25 to 35, the constraint updates $array$ by making $array[i+1]$ equal to $array[i]$ and decreases i by 1. Then line 35 satisfies the third path condition $i < 0$ making the path jump out of the *while* statement. After updating the values of $array[i+1]$ (line 36∼39) and j (line 40∼42), the constraint confirms the fourth path condition $j >= n$ to break the for statement and reaches the end of the path.

IV. I/O-Based Query

Before describing our approach to an I/O-based query, we give a formal definition of the query problem (that refers to paper [10], [19]). Given a method declaration with its inputs, outputs, and method body, we use a 3-tuple $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$ to model it where $\widetilde{I}$ is the set of its input variables, $\widetilde{O}$ is the set of its output variables, and $\widetilde{P}$ includes all its feasible paths. Each I/O variable has its type, which we use function τ to obtain.

A set of I/O examples is required as the query specification. Formally, we refer to $(\boldsymbol{\sigma}, \boldsymbol{\nu})$ as an I/O example, where $\boldsymbol{\sigma}$ and $\boldsymbol{\nu}$ are vectors of input and output values. Each I/O value also corresponds to a certain type, which can also be obtained by function τ. Note that τ can act on a vector of variables or values, which equals to applying τ to each of its element.

Given a method $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$ and an I/O example $(\boldsymbol{\sigma}, \boldsymbol{\nu})$, we use function α, β to bind $\boldsymbol{\sigma}, \boldsymbol{\nu}$ to a subsequence of variables in $\widetilde{I}, \widetilde{O}$, which satisfies $\tau(\boldsymbol{\sigma}) = \tau(\alpha(\boldsymbol{\sigma}))$ and $\tau(\boldsymbol{\nu}) = \tau(\beta(\boldsymbol{\nu}))$. Since we have encoded path $P \in \widetilde{P}$ into path constraint P_{sym}, the query problem turns into whether

$$\exists P \in \widetilde{P} : P_{sym}[\alpha(\boldsymbol{\sigma})] = \boldsymbol{\sigma} \wedge P_{sym} \wedge P_{sym}[\beta(\boldsymbol{\nu})] = \boldsymbol{\nu} \quad (1)$$

is satisfiable or not, where $P_{sym}[\alpha(\boldsymbol{\sigma})]$ and $P_{sym}[\beta(\boldsymbol{\nu})]$ are the symbols with the latest version of $\alpha(\boldsymbol{\sigma})$ and $\beta(\boldsymbol{\nu})$ in P_{sym}. We say a path accepts the example if the path satisfies the above formula.

Based on the formal definition, our approach to I/O-based queries can be divided into three phases: type matching, path matching, and result ranking, which are shown in Procedure 2 and Procedure 3.

The type matching phase refers to line 2 in Procedure 2. Given a method $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$ and an I/O example $(\boldsymbol{\sigma}, \boldsymbol{\nu})$, we check whether there exist functions α, β, which map each value in $(\boldsymbol{\sigma}, \boldsymbol{\nu})$ to variables in $\widetilde{I}, \widetilde{O}$ satisfying that the data type of $\alpha(\boldsymbol{\sigma}), \beta(\boldsymbol{\nu})$ are the same as that of values in $(\boldsymbol{\sigma}, \boldsymbol{\nu})$.

Procedure 2 QUERY

```
Input: Method $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, I/O Examples $\{(\boldsymbol{\sigma}_i, \boldsymbol{\nu}_i)\}_{i=1}^{m}$
Output: Priority $Pri$
 1: $Pri$ := 0
 2: if there exists function $\alpha, \beta$ that $\alpha(\boldsymbol{\sigma}) \sqsubseteq \widetilde{I}$, $\beta(\boldsymbol{\nu}) \sqsubseteq \widetilde{O}$,
    $\tau(\alpha(\boldsymbol{\sigma})) = \tau(\boldsymbol{\sigma})$, $\tau(\beta(\boldsymbol{\nu})) = \tau(\boldsymbol{\nu})$ then
 3:     for all such function $\alpha, \beta$ do
 4:         $N_{sat}$ := $N_{uns}$ := $N_{unk}$ := 0
 5:         for each $(\boldsymbol{\sigma}, \boldsymbol{\nu}) \in \{(\boldsymbol{\sigma}_i, \boldsymbol{\nu}_i)\}_{i=1}^{m}$ do
 6:             $state$ := PATHMATCHING($S$, $\alpha(\boldsymbol{\sigma})$, $\beta(\boldsymbol{\nu})$)
 7:             if $state$ = $sat$ then $N_{sat}$++
 8:             else if $state$ = $unsat$ then $N_{uns}$++
 9:             else $N_{unk}$++
10:             end if
11:         end for
12:         $Pri$ := max($Pri$, PRIORITY($N_{sat}$, $N_{uns}$, $N_{unk}$))
13:     end for
14: end if
15: function PRIORITY($N_{sat}$, $N_{uns}$, $N_{unk}$)
16:     if $N_{sat} = 0$ then return 0
17:     else
18:         $m$ := $N_{sat} + N_{uns} + N_{unk}$
19:         return $(N_{sat} + 0.5N_{unk})/m$
20:     end if
21: end function
```

Procedure 3 PATHMATCHING

```
Input: Method $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, Binding $\alpha(\boldsymbol{\sigma}), \beta(\boldsymbol{\nu})$
Output: Status $\{sat, unsat, unknown\}$
 1: $\widetilde{P}_{sym}$ := SYMBOLICENCODING($S$) //offline
 2: for each $P_{sym} \in \widetilde{P}_{sym}$ do
 3:     $iCons$ := $P_{sym}[\alpha(\boldsymbol{\sigma})] = \boldsymbol{\sigma} \wedge P_{sym}$
 4:     $ioCons$ := $iCons \wedge P_{sym}[\beta(\boldsymbol{\nu})] = \boldsymbol{\nu}$
 5:     if $|\widetilde{I}| = |\boldsymbol{\sigma}|$ then
 6:         if solve($iCons$) = $sat$ then
 7:             return solve($ioCons$)
 8:         end if
 9:     else if solve($ioCons$) = $sat$ then
10:         return $sat$
11:     end if
12: end for
13: return $unknown$
```

If the answer is “no”, we would not do path matching on this method. Otherwise, we enumerate all such $\alpha(\boldsymbol{\sigma}), \beta(\boldsymbol{\nu})$ and pass them together with the method to the path matching phase. We regard the method passed to the path matching phase as a candidate method.

The path matching phase is shown in Procedure 3. Given a candidate method $S = \langle \widetilde{I}, \widetilde{O}, \widetilde{P} \rangle$, we use $\widetilde{P}_{sym}$ to denote the set of path constraints generated in the offline symbolic encoding procedure and each $P_{sym} \in \widetilde{P}_{sym}$ denotes the constraint of path $P \in \widetilde{P}$. In Procedure 3, we construct two constraints based on P_{sym}, $\alpha(\boldsymbol{\sigma})$, and $\beta(\boldsymbol{\nu})$, to check whether the path P accepts the I/O example $\boldsymbol{\sigma}, \boldsymbol{\nu}$. The first constraint, named $iCons$, is constructed by only binding each value of $\boldsymbol{\sigma}$ to its corresponding variable of $\alpha(\boldsymbol{\sigma})$ in P_{sym}, which requires $\boldsymbol{\sigma}$ to be a feasible input for P that satisfies all P's branch conditions. The second constraint, named $ioCons$, is constructed by binding each value of $\boldsymbol{\nu}$ to its corresponding variable of $\beta(\boldsymbol{\nu})$ further, which corresponds to Formula 1 and requires the path P should produce the expected output $\boldsymbol{\nu}$ given the feasible input $\boldsymbol{\sigma}$. Note that $iCons$ is used to check whether the result of path matching is $unsat$ (i.e. each path of the candidate method cannot accept the I/O example). There are three possible results of the path matching procedure:

sat means we find a path that accepts $\boldsymbol{\sigma}, \boldsymbol{\nu}$, which is returned only when the constraint $ioCons$ is satisfiable (line 7,10 of Procedure 3).

$unsat$ means in condition of $|\widetilde{I}| = |\boldsymbol{\sigma}|$, we find a path P that satisfies $iCons$ but not $ioCons$ (i.e. $P_{sym}[\alpha(\boldsymbol{\sigma})] = \boldsymbol{\sigma} \wedge P_{sym} \wedge \neg(P_{sym}[\beta(\boldsymbol{\nu})] = \boldsymbol{\nu})$). Note that if all inputs are assigned with concrete values (i.e. $|\widetilde{I}| = |\boldsymbol{\sigma}|$), the method will execute only one path. As we have found the path of which $\boldsymbol{\sigma}$ is a feasible input but it does not produce the expected output $\boldsymbol{\nu}$, there would not exist any other path that accepts $\boldsymbol{\sigma}, \boldsymbol{\nu}$ (line 10 of Procedure 3).

$unknown$ will appear in the following two situations:

- when $|\widetilde{I}| = |\boldsymbol{\sigma}|$, we find that each path P in our repository does not satisfy the constraint $iCons$. That is to say, although all input variables are assigned with concrete values, there is no path in our repository of which $\boldsymbol{\sigma}$ is a feasible input. However, there may exist another path with $\boldsymbol{\sigma}$ as its feasible input, which is not in our repository.
- when $|\widetilde{I}| \neq |\boldsymbol{\sigma}|$, we find no path in our repository that satisfies the constraint $ioCons$. But since we just assign parts of input variables with concrete values, there may exist other paths that accepts $\boldsymbol{\sigma}, \boldsymbol{\nu}$, which is not in our repository.

After performing the path matching for every I/O example on a candidate method S according to a binding function, we will update the priority of S, which is used to measure the probability of S being a correct solution for the query question. The calculation of priority is shown as a function of Procedure 2, where N_{sat} and N_{uns} refer to how many I/O examples are accepted and rejected, respectively, and N_{unk} refers to how many I/O examples are unknown to be accepted. Intuitively, a candidate method with higher N_{sat} and N_{unk} (thus, lower N_{uns}) is more likely to be a correct solution and N_{sat} plays a more important role. The function PRIORITY means if no examples are accepted by the candidate method then its priority is 0, otherwise the priority is the scale of sat (with weight 1) and $unknown$ (with weight 0.5) among all path matching results. In the result ranking phase, we will filter out the candidate methods whose priority is less than 0.5 and rank the remaining in a priority decreasing order.

As for E3 in EXAMPLE 1, $\sigma = [2, 1]$, $\nu = [1, 2]$. When we conduct path matching on p_3, there is only one binding way, $\alpha(\sigma) = array\_0$, $\beta(\nu) = array\_2$. In Procedure 3, we append (= array_0 (seq.++ (seq.unit 2) (seq.unit 1))) to the path constraint of p_3 to construct $iCons$ and append (= array_2 (seq.++ (seq.unit 1) (seq.unit 2))) to $iCons$ to construct $ioCons$. The result of solving $iCons$ and $ioCons$ are both sat, which means p_3 accepts E3.

V. IMPLEMENTATION

Our tool implementation of symbolic encoding uses Soot[1] to generate the CFG in a typed 3-address intermediate representation (IR) called Jimple. We choose Z3 [18] as the SMT solver to filter out infeasible paths and support the query service. The path constraint is written in Z3 input format [20], an extension of the one defined by the SMT-LIB 2.0 standard[2]. We design a simple I/O query language and use ANTLR[3] to parse it. Table IV shows its grammar, which support 7 kinds of data type: int, float, boolean, String, List, Set, and Map. The name of our search engine is *Quebio* (Query by input/output).

TABLE IV
QUERY LANGUAGE GRAMMAR

```
 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'
```

VI. EVALUATION

We propose three research questions to evaluate the feasibility and performance of our approach:

RQ1: How many realistic source code and questions can be handled by our approach?

RQ2: How effective is our approach when solving realistic questions?

RQ3: How efficient is our approach when solving realistic questions?

Among the research questions mentioned above, RQ1 is about the feasibility of our approach, which is the basis of our experiment. RQ2 and RQ3 are about the search performance. When it comes to evaluating the performance of a search engine, people are usually concerned with whether they can find the correct results and filter out the wrong results as much as possible, which reflects the effectiveness of a search engine. The other factor people are concerned with is the

[1] https://github.com/Sable/soot
[2] http://smtlib.cs.uiowa.edu
[3] https://github.com/antlr/antlr4

TABLE V
EXPERIMENTAL DATA

ID	Question	Keyword	#L	#V	RFC	#PATH	#SYM	TIME (s)
Q1	Get the separate digits of an int number	separate digits number	14	3	6	986	30,140	27.68
Q2	Reverse an int value without using array	reverse int value	221	93	1	9,168	187,591	188.06
Q3	Concatenate int values	concatenate int value	76	13	-	6,228	174,818	51.55
Q4	Sorting integers in order lowest to highest	int array sort	183	11	1	24,221	1,031,520	118.19
Q5	Convert letters in a string to a number	convert letter to int	13	4	1	1,349	24,976	16.05
Q6	Rounding a double to turn it into an int	round double int	9	3	1	63	1,144	1.02
Q7	Dividing two integers to a double	divide integer	5	4	1	70	530	1.08
Q8	Round a double to 2 decimal places	round double places	19	18	1	348	2,918	5.09
Q9	Checking if an int is prime more efficiently	check int prime	125	56	1	3,840	33,525	47.33
Q10	Pad an integers with zeros on the left	pad integer	0	0	-	932	27,655	140.72
Q11	Convert a String to an int	convert string to int	28	3	1	1,511	33,271	24.25
Q12	Check if a String is numeric	string numeric	34	12	1	1,791	26,065	16.69
Q13	Count the number of a char in a String	count char in string	3	2	1	996	31,835	8.31
Q14	Reverse a string	string reverse	43	0	1	17,418	608,336	33.5
Q15	Check whether a string is not null and not empty	string null empty	59	6	1	1,630	23,329	14.56
Q16	Check string for palindrome	string palindrome	38	12	1	1,664	24,251	17.82
Q17	Concatenate two strings	string concatenate	5	3	1	1,133	36,351	13.69
Q18	Evaluating a math expression given in string form	string evaluate	15	1	-	1,858	45,271	29.74
Q19	Convert an ArrayList to a string	convert list to string	2	1	2	382	23,740	1.26
Q20	Remove the last character from a string	string remove last	30	2	-	2,091	65,459	24.86
Q21	Test if an array contains a certain value	array contain	20	5	1	1,310	30,160	8.48
Q22	Concatenate two arrays	array concatenate	13	0	1	3,408	117,441	25.83
Q23	Reverse an int array	array reverse	217	9	1	11,667	346,873	94.39
Q24	Remove objects from an array	array remove	25	0	1	5,893	190,950	48.07
Q25	Add new elements to an array	array add	4	1	1	10,214	333,233	82.12
Q26	Finding the max value in an array of primitives	array max	63	18	1	6,854	196,849	56.23
Q27	Finding the min value in an array of primitives	array min	101	21	1	5,758	154,208	45.6
Q28	Finding index of maximum from slice of an array	array max index	101	32	1	5,440	143,427	42.88
Q29	Sort an array	array sort	201	5	7	21,746	924,149	180.3
Q30	Array Finding Duplicates	array find duplicate	45	29	1	2,187	44,929	15.61
Q31	Array Sort descending?	array sort descending	222	4	5	11,980	363,058	98.65
Q32	Find the index of an element in an array	array find index	42	13	1	5,419	175,594	41.17
Q33	Remove repeated elements from ArrayList	array remove repeated	181	3	-	12,455	373,654	99.75
Q34	Intersection of ArrayLists	arraylist intersection	22	0	-	3,130	106,394	23.94
Q35	Union of ArrayLists	arraylist union	23	0	1	3,019	101,743	22.77
		Total	2202	387	N/A	188,159	6,035,387	1,667.24
		Average	62.91	11.06	1.55	5,375.97	172,439.63	47.64

search time, which reflects the efficiency of a search engine. In our experiment, we design some metrics to measure the effectiveness and efficiency of our approach to answer RQ2 and RQ3.

A. Experimental Subjects

We collect questions and code from the programming Q&A community Stack Overflow, and the open source hosting website GitHub as our experimental subjects. After reviewing the most frequent questions about Java on Stack Overflow, which concern data types like *int*, *double*, *String*, *array*, and *List*, we collect 35 questions that can be described by I/O examples. These questions are shown in Table V, among which 10 (Q1∼Q10) are tagged with *int* or *double*, 10 (Q11∼Q20) are tagged with *String*, and 15 (Q21∼Q35) are tagged with *array* or *List*.

Our local repository is built on code gathered from GitHub, to avoid ambiguity, we use "project" to respresent the GitHub repository in this paper. Since there are billions of projects on GitHub, it is impossible for us to crawl them blindly. As a result, we build our local repository in a relatively practical way. It is necessary to note that GitHub supports search for projects and code snippets by keywords. Given the questions to answer, we first translate them to some keywords which are shown in Table V. Then we search for projects and code snippets containing these keywords from GitHub and record the top 5 projects and top 15 code snippets. For each code snippet, we further find the project it is located in. In this way, we collect 333 distinct projects from all 700 (20×35) projects, among which many projects are duplicate. We clone all these projects into our local machine and analyze the abstract syntax tree (AST) for each method (2,929,656 altogether). We select the method that we can handle using the following criteria: (1) it is not an abstract or native method; (2) its name does not start with "main", "test", "get" or "set"; (3) the data type of its variables is primitive, String or container. Finally, we select 32,261 methods in this way and successfully encode 1228 of them (29,099 lines of code) into 21,679 path constraints in our local repository.

The reasons why the 31,003 methods are not encoded into our local repository are grouped into 7 classes, which is shown in column **Reason** of Table VI. *Duplicate* means that we have encoded the method that has the same code as the current one into our local repository. *Soot Error* indicates that we meet a runtime exception when using Soot to transform the code into IR. *Invoke Error* means that the method invokes another method, of which we cannot get the source code or constraint template. *Unhandled Type* means that there are variables with data type that we cannot currently handle. *Bit Operation* indicates that the method contains bit operations that our approach cannot process. *Illegal SMT* means that the constraint has syntax error, e.g. there is a literal "null" in the constraint. *No Input* means the method has no variable as input. *No Output* means the method has no variable as output. *Others* represents the reasons that are not mentioned above. *Multiple Reasons* means that there are multiple reasons why the method cannot be encoded.

TABLE VI
REASONS FOR NOT ENCODING

Reason	#M	%M
Duplicate	184	0.6
Soot Error	25,777	83.1
Invoke Error	995	3.2
Unhandled Type	289	0.9
Bit Operation	232	0.8
Illegal SMT	438	1.4
No Input	1,703	5.5
No Output	625	2.0
Others	444	1.4
Multiple Reasons	346	1.1
Total	31,033	100

Column **#M**, **%M** of Table VI show the number and percentage of methods that are failed to be encoded for the reason in the **Reason** column, from which we can see that *Soot Error*, *Invoke Error*, *No Input*, and *No Output* are the main reasons, the percentage of which reaches 93.8% in total.

B. Experimental Protocol

Our experiment is designed to use semantics-based code search to answer programming questions. We first give 5 I/O examples for each question manually according to its description on Stack Overflow. The examples given here should be simple but representative. Then we use these examples as the query specification to conduct semantics-based code search for a method from the local repository. To answer RQ2 and RQ3, we evaluate the effectiveness and efficiency of our approach during the experiment.

1) Effectiveness: In order to evaluate the effectiveness, we label the returned methods with three labels: *likely*, *valid*, and *correct*. All methods with priority higher than 0.5 will be labeled with *likely*. If a method can accept all I/O examples (with priority equals to 1), we label it with *valid*, which means it satisfies the query specification. We label the method that actually solves the question with *correct*. It is necessary to note that a valid method may not solve the question, whether a method is correct should be checked manually. The method that is valid but not correct will affect the experimental result directly. In our experiment, labeling the correct methods is finished by two authors. One author labels correct methods according to the question description on Stack Overflow and the other author reviews these labels. Even this process cannot guarantee the objectivity, and a method will only be marked as *correct* when there is consensus between the authors.

Based on the three labels, we use three metrics to evaluate the effectiveness. **#L** refers to the number of likely methods returned. **#V** refers to the number of valid methods. **RFC** refers to the rank of the first correct method among all likely methods. Note that the rank of a method mentioned here is the number of methods with higher priority than it. For example, suppose that we in total have 5 likely methods, 3 with priority 0.9 and 2 with priority 0.8, the rank of methods with priority 0.9 is 1st and the rank of methods with priority 0.8 is 6th.

2) Efficiency: The efficiency of our approach is reflected by the search time, which we think is relevant to the number and scale of path constraints solved in each search session. We record **#PATH**, **#SYM**, and **TIME** during the experiment, where **#PATH** refers to the number of path constraints solved in each search session, **#SYM** refers to how many symbols exist in these path constraints, and **TIME** refers to the total search time. In practice, we set the timeout of Z3 to 2 seconds such that it returns "unknown" if it cannot solve the constraint within the time limitation. The reason why we use 2 seconds as the timeout is based on our empirical observation, which shows that most path constraints in our local repository can be solved within 2 seconds by Z3. If Z3 spends more time on solving a path constraint, most likely it will return "unknown" or crash because of running out of memory. That is to say, solving time longer than 2 seconds is not helpful to improve the search performance but leads to unnecessary overhead.

The search time can be improved by parallelization, because the path matching procedure can be processed for different candidate methods at the same time during a search session. In our experiment, we use 4 threads to process the path matching procedure on the computer with 8 Intel Core i7-6700 CPUs and 16G RAM, the operating system of which is Ubuntu 16.04.

C. Experimental Results

1) RQ1: We use the experimental subjects to answer RQ1. There are 29 questions that can be answered by our approach, and 1228 methods that have been encoded into our local repository. Since these questions and code are realistic on the Internet, they are convincing to show that our approach is feasible on general Java code search situations.

2) RQ2: Columns **#L**, **#V**, **RFC** of Table V give the experimental data to answer RQ2. We succeed in finding the correct methods for 29 queries and fail to find them for Q3, Q10, Q18, Q20, Q33, and Q34. The reason why we fail to find correct methods for these queries is unknown up to now. We guess there is no correct method for these queries in our local repository, but it is hard for us to check all of them. Further more, we observe that the rank of the first correct method is almost 1st among all likely methods and the worst case (RFC of Q29) is 7th, which means it requires users to review at most 7 methods from top to down to find the correct method.

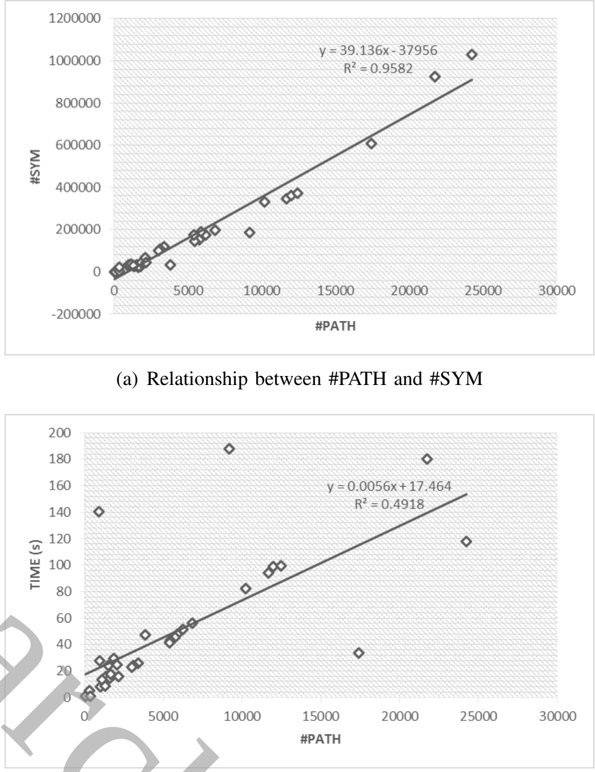
There are some queries with more than 100 likely methods such as Q2 and Q23, and some with less than 5 such as Q13 and Q19. The experimental result depends on what examples are used as the query specification. For example, the I/O examples we use to answer Q23 are `([1]->[1])`, `([1,2]->[2,1])`, `([1,1]->[1,1])`, `([1,3,2]->[2,3,1])`, `([1,3,2,4]->[4,2,3,1])`. Many methods can accept parts of them. For example, an array copy method can accept the first two examples and a sort method can accept the first three. But a valid method should accept all examples. That is to say, although there are 217 likely methods for Q23, only 9 are valid, we can select correct methods from the valid methods in this case. As for Q13, its I/O examples are `("a","a"->1)`, `("ab","c"->0)`, `("aab","a"->2)`, `("aab","b"->1)`, `("acab","a"->2)`. Because these examples are fully representative, we can imagine that if a method cannot solve Q13, most likely it would not accept any of them, that is why we only find 3 likely methods for Q13.

There are also some queries with many valid methods such as Q2 and Q9. For example, we use `(123->321)` as an I/O example for Q2. Suppose there is a method "int add(int a, int b){return a+b;}" in our local repository, if we assign 123 to a, the SMT solver can find a model where $b = 198$ to satisfy $a + b = 321$, which means this method is always valid. This phenomenon is caused by our query strategy which cannot be avoided. A ranking process can be installed to order the likely methods in a particular way. In our experiment, the first correct method for Q2 names *reverse* and that for Q9 names *isPrime*, which are similar to the question keywords. These methods will gain a high rank if the ranking process focuses on the similarity between question keywords and method names.

In brief, we successfully find correct methods for 29 queries and the first correct method ranks on top 10 among all likely methods. These experimental results are convincing to show that our approach is effective on solving realistic questions.

3) RQ3: Columns **#PATH**, **#SYM**, **TIME** of Table V show the experimental data to answer RQ3. It is necessary to emphasize that we give 5 I/O examples for each query, Z3 timeout is 2 seconds, and we use 4 threads to solve the constraints of candidate methods concurrently in our experiment. Table V shows that the average search time is around 47.64 seconds, and the average number of paths and symbols are 5,376 and 172,440, respectively. We use two scatter diagrams in Figure 4 to reflect the relationship between TIME, #PATH, and #SYM. Figure 4(a) shows the distribution of (#PATH, #SYM) in all search sessions and Figure 4(b) shows that of (#PATH, TIME), from which we can observe that there exist linear relationships between TIME, #PATH, and #SYM. The outlier in Figure 4(b) appears when we solve Q2, Q10, and Q14. The reason why these outliers exist depends on the complexity of specific path constraints solved in these search sessions, which is currently hard to measure.

Now we can answer RQ3. In the experiment, our approach takes around 47.64 seconds on average to search for source code. There exist a linear relationship between the search time and the number of path constraints to be solved. If we use a single thread in the search experiment, the average search time is around 114 seconds, which is 2.4 times of the current search time.



(a) Relationship between #PATH and #SYM

(b) Relationship between #PATH and TIME

Fig. 4. TIME, #PATH, and #SYM of 35 search sessions

D. Threats to Validity

There are several threats to the validity of our studies. The main threat to internal validity arises as we do not evaluate the correctness of path constraints that we generate from source code. We mitigate this threat by reviewing some path constraints in our local repository. After filtering out infeasible paths of a method, the SMT solver will also return a model for each feasible path that satisfies its constraint. We check whether the I/O instance included in the model is a valid I/O example of the path.

Threats to external validity arise when our experimental results cannot be generalized. The first threat is that our code is obtained by keywords of the questions to be solved. We mitigate this threat by encoding 1228 methods into the local repository, the quantity of which are far greater than that of the questions. Both correct solutions and other irrelevant code are in our local repository, which can be regarded as a simulation of realistic code search environment. The second threat is that our experimental subjects only include methods with primitive, String, or container variables, which is limited by the processing ability of our approach. The third threat is that the 5 I/O examples for each question are written by authors. The fourth threat is that we label the correct method manually and the result depends on personal judgement. We mitigate it by finishing this task by two authors.

VII. Related Work

Our work is an assistance to help software developers search and reuse code on the Internet. To support semantics-based code search, we generate path constraints, a kind of specification, for a Java method via symbolic analysis. There are three kinds of work related to the present paper: code search, symbolic execution, and specification generation.

A. Code Search

There is a large body of work aims to help developers search for source code efficiently. Current code search engines can be roughly divided into two classes. The first class is the keyword-based search engine, such as Google, Koders, Krugle[4], Google Code Search, and SourceForge[5], which have been compared in [2].

Nowadays, GitHub becomes more and more popular for developers as an online code hosting website. In our experiment, we use keywords to search for projects and code as our experimental subjects on GitHub. There are 66 projects and 3 million code returned for each query on average, which is a fairly large quantity. We find correct projects for 20 queries, and correct code for 34 queries, the first rank of which are about 3rd and 8th on average. Compared with the keyword-based search, our approach is more effective on refining the search results, but sacrifices the efficiency. The keyword-based search can be finished within one second, which is quite faster than our 48 seconds search time.

The second class is semantics-based code search. Stolee's paper [1] compares several semantics-based search engines such as Exemplar [3], S6 [4], CodeGenie [5], and Sourcerer [6]. These search engines use data-flow analysis, testing, and AST analysis to get the semantic information behind the source code and utilize it to guide the search for code. The work of Raghothaman et al. [21] combines code search and program synthesis together to suggest code snippets given API-related natural language queries. It first searches for APIs relevant to queries in the Bing search engine and code fragments relevant to these APIs from GitHub. Then it learns a probabilistic model to describe usage patterns from these APIs and code fragments. Finally, they use the probabilistic model to synthesize a idiomatic code describing the usage of these APIs.

Our work is mainly related with Satsy [7], [8]. It is the first that proposes the basic idea that uses I/O examples and SMT solver to find source code behaving as specified by these examples. Our work can be regarded as an extension and improvement of it, which applies the technology to realistic Java code search situations.

B. Symbolic Execution

Symbolic execution [22] is a program analysis technique that generates test data for a program automatically by solving its path constraints. The survey [12] points out that all existing symbolic execution tools such as KLEE [23] and SPF [9] suffer from three fundamental problems that limit their effectiveness on real world software. The first problem is path explosion because one program may have extremely large number of paths, which is impossible to enumerate a large subset of all feasible paths in a reasonable time. The second problem is path divergence, which means there is some code of a program that is shown in binary form, e.g. native code in Java program, which is hard to construct the path constraint. The third problem is that many path constraints are too complex to solve, e.g. it is difficult for existing constraint solvers like Z3 [18] to solve complex constraints involving nonlinear operations.

Our work is also faced with the three problems. As for path explosion, because we use path constraint to search for code instead of testing, a small subset of all feasible paths is enough to accept the examples given by users. In practice, when developers search for code with I/O examples, they usually give some simple but representative examples which can be accepted by some short paths appearing in the small subset. As for path divergence and complex constraints, they often occur when dealing with method invocation. Our approach uses constraint templates to deal with method invocations instead of running the invoked method, which can avoid meeting binary code and generating quite complex constraints.

C. Specification Generation

Essentially, path constraint is a kind of specification of a program. There is a lot of work that generates specification from source code automatically, such as invariant inference [24], [25], and documentation analysis [16]. Invariant inference can infer the precondition and postcondition of a method by analyzing the runtime data gathered from several times of dynamic executions, but it cannot guarantee the correctness of these invariants. Zhai et al.'s paper [16] constructs models for Java APIs by analyzing the documentation. These models are simpler implementations in Java compared to the original ones and hence easier to analyze. Based on existing work, we can improve our approach by generating constraint templates from alternative specifications automatically, which we leave as future work.

VIII. Conclusions

Code search plays an important role in modern software engineering. With the number of source code available on the Internet increasing, semantics-based code search can help software developers more effectively retrieve and reuse existing code. In this paper, we present an approach to semantics-based code search using I/O examples for Java language. We encode Java methods into path constraints and leverage SMT solvers to check whether a method satisfies the query specification. Our approach extends the applicability of this

[4] http://www.krugle.com
[5] https://sourceforge.net

search technology to more general Java code compared with existing methods. The experiments conducted on realistic code search queries suggest our approach's efficacy. We collected 35 questions from Stack Overflow, and encoded 1228 methods from GitHub into our local repository. Based on these experimental subjects, we successfully find correct methods for 29 questions and the first correct method ranks on top 10 among all likely methods. The average search time is just about 48 seconds.

References

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

[2] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 4:1–4:25, Dec. 2011.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

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

[5] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and 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*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 525–526.

[6] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and 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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 681–682.

[7] K. T. Stolee, "Solving the search for source code," Ph.D. dissertation, University of Nebraska - Lincoln, 2013.

[8] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 26:1–26:45, Jun. 2014.

[9] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and 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*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 15–26.

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

[11] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 441–452.

[12] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey on automated software test case generation," *Journal of Systems & Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

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

[14] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11.

[15] P. Lam, E. Bodden, L. Hendren, and T. U. Darmstadt, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oct. 2011.

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

[17] "Java se 8 api specification." [Online]. Available: https://docs.oracle.com/javase/8/docs/api/index.html

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

[19] O. Polozov and 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*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 107–126.

[20] "Z3-guide," Microsoft Research. [Online]. Available: https://rise4fun.com/Z3/tutorial/guide

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

[22] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, May 1976.

[23] C. Cadar, D. Dunbar, and 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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

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

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