The Journal of Systems and Software 000 (2015) 1–14

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Code search with input/output queries: Generalizing, ranking, and assessment

Kathryn T. Stolee [a,*], Sebastian Elbaum [b], Matthew B. Dwyer [b]

[a] Iowa State University, Ames, IA 50011, United States
[b] University of Nebraska-Lincoln, Lincoln, NE, United States

ARTICLE INFO

Article history:
Received 29 August 2014
Revised 27 February 2015
Accepted 21 April 2015
Available online xxx

Keywords:
Semantic code search
Symbolic execution
SMT solvers

ABSTRACT

In this work we generalize, improve, and extensively assess our semantic source code search engine through which developers use an input/output query model to specify what behavior they want instead of how it may be implemented. Under this approach a code repository contains programs encoded as constraints and an SMT solver finds encoded programs that match an input/output query. The search engine returns a list of source code snippets that match the specification.

The initial instantiation of this approach showed potential but was limited. It only encoded single-path programs, reported just complete matches, did not rank the results, and was only partly assessed. In this work, we explore the use of symbolic execution to address some of these technical shortcomings. We implemented a tool, Satsy, that uses symbolic execution to encode multi-path programs as constraints and a novel ranking algorithm based on the strength of the match between an input/output query and the program paths traversed by symbolic execution. An assessment about the relevance of Satsy's results versus other search engines, Merobase and Google, on eight novice-level programming tasks gathered from StackOverflow, using the opinions of 30 study participants, reveals that Satsy often out-performs the competition in terms of precision, and that matches are found in seconds.

1. Introduction

Programmers frequently search for code when performing programming tasks (Sim et al., 2011; Stolee et al., 2014). The search approaches vary across several dimensions, including the scope of the search (from local to web-scale repositories), the purpose of the search (from learning to reuse), or the search tool (from general search engines to code specific search engine). Yet one aspect that most search approaches share is the use of keywords in the query to perform a search. To illustrate, a keyword query could be a description of a programming task, such as *remove the file extension from a file name in Java*. The search engine then traverses an indexed repository of programs in an attempt to find a syntactic match to the keywords.

In previous work, we defined a semantic approach to code search that takes an input/output query model and retrieves code from a repository that behaves as specified, using a constraint solver to identify search results (Stolee and Elbaum, 2012; Stolee et al., 2014) (illustrated as part of Fig. 7). The approach's value resides in enabling a different type of search where users search for *what* they want without knowing *how* it may have been implemented. Using the same programming task as an example, this query model would let the programmer specify an input, such as "file.txt", and an output, "file". A naïve semantic approach would attempt to execute programs with those inputs and outputs to find a match, but it would fail if the signatures differ or if there is only a partial match (a program that provides extra behavior or is missing some behavior). Instead, to enable a broader use of such queries, our approach performs a program indexing that considers the program's semantics. This indexing is done by transforming code snippets into constraints; at search time, to find a match an SMT solver is used to check for satisfiability between a program constraints and the input/output query. As previously proposed (Stolee et al., 2014), the approach was able to encode single-path programs with string, integer, character and boolean data types, return all code snippets that completely matched a specification, and its potential was illustrated through its application to three language subsets (SQL Select, Yahoo! Pipes, and Java Strings). The assessment showed that it was a feasible alternative to keyword based searches, especially when the precision of the results was a priority.

In this work, we use the program characterization produced by symbolic execution to significantly extend the previous search work in two dimensions. First, we generalize the encoding to include

* Corresponding author. Tel.: +1 515 294 0222.
E-mail addresses: kstolee@iastate.edu (K.T. Stolee), elbaum@cse.unl.edu (S. Elbaum), dwyer@cse.unl.edu (M.B. Dwyer).

http://dx.doi.org/10.1016/j.jss.2015.04.081
0164-1212/

multi-path, non-looping programs by integrating symbolic execution into the indexing phase of the code search approach. During indexing, each code snippet is executed symbolically and the path conditions at the leaves of the symbolic execution tree are collected; the set of path conditions represents the potential behaviors for the program. Then, during a search, programs are identified as matches when at least part of a specification matches at least one path in a program (leading to partial matches).

Second, we use the output from symbolic execution to rank the search results based on the strength of a match between an input/output specification and a source code snippet. To prioritize the matches, the ranking algorithm analyzes the level of correspondence between the input/output queries and the traversed program paths. We have defined various levels of matching between a specification and a program, including a *full match*, an *under match* in which the program is missing some behavior in the specification, and an *over match* in which the program contains extra behavior.

In addition to the technical extensions, in this paper we also perform an extensive evaluation of the approach implemented in a tool we call Satsy. In the context of Java programs, we compare Satsy with Google (which has been shown to be the state-of-the-practice code search tool (Sim et al., 2011; Stolee et al., 2014)) and a source code search engine, Merobase. The study includes eight novice-level programming problems posed on stackoverflow.com primarily related to string manipulation, which has been found to be medium difficulty for novice Java programmers (Milne and Rowe, 2002). For each programming task, queries were issued to each search approach. The study shows that, based on an evaluation by 30 programmers, Satsy is on average more effective than Google, and significantly more effective than Merobase. As part of the study we also evaluate Satsy's performance and began to investigate the impact of the specification size on the results. The results also indicate that the ranking algorithm has a profound impact on the relevance of Satsy's results, a larger repository could really improve the number of matches, and the approach does well when specifications contain 3–4 input/output examples.

Even considering the current limitations in scalability for symbolic execution, the code search context provides a compelling application for this powerful technique. First, performance is not a concern as indexing happens offline. Second, the target code fragments can be rather small, which is where symbolic execution excels. Last, just like when supporting test case generation, even if symbolic execution fails to complete it can still collect partial program encodings to enable search albeit with less precision.

The contributions of this work are:

- Generalization of our semantic search technique and matching criteria to consider multi-path programs through the use of symbolic execution and partial matches.
- The first semantic ranking algorithm for source code search results based on the strength of a match between a specification and the program paths.
- Evaluation with human participants showing that Satsy often outperforms Google, and that both are better than a code search specific search engine, Merobase, when searching for solutions to novice-level programming tasks.
- Evaluations measuring Satsy's performance and illustrating the impact of the number of input/output examples on result relevance.

2. Motivation and overview

In this section, we motivate the general search approach of using input/output examples and a constraint solver for search. Then, we present the need for the particular contributions of this work, specifically the need for multi-path program encoding, provide intuition for how to generate such encodings through symbolic execution, and illustrate how this creates an opportunity to use the strength of a match between specification and program in a ranking algorithm.

```
1 LS = { ({"alpha", "PHA"}, {true}),          \\ ls_a
2        ({"beta", "bet"}, {true})             \\ ls_b
```

Fig. 1. Specification for containment, case insensitive.

2.1. Limitations with keyword-based search and testing

The search approach we propose in this work is based on the use of input/output queries and a constraint solver to identify programs. This is specifically in contrast to previous work that has focused on keyword-based approaches (e.g., Bajracharya et al., 2006; Grechanik et al., 2010; Inoue et al., 2003; McMillan et al., 2011) or testing-based approaches (e.g., Lemos et al., 2007; Podgurski and Pierce, 1993; Reiss, 2009, details in Section 8).

Keyword-based approaches to code search generally require users to formulate text that describes their needs, and then match that textual query to the text contained in source code or related documentation. Thus, the success of the search is generally dependent on the users' abilities to select words that may have been used (or are similar to) words that exist in a program that performs the desired task. This process may require several reformations of the query (Fischer et al., 1991; Haiduc et al., 2013), and motivates the exploration of behavior-based approaches.

Testing-based approaches to code search find code based on behavior and work well when the signature of the specification matches the signature of a code snippet. However, when the specification contains too much information, or does not contain enough information to exactly fit a snippet's signature, such approaches fall short.

To illustrate, consider the problem of determining if one string contains another, case insensitive (this question is similar to one posed by a real developer on stackoverflow.com [1] and is used in our study – question 1 in Table 3). The keyword query to Google formed by one of the study participants, *"String contains String Java"*, returns over 1.5 million results. Four of the top five results provide a case-sensitive, rather than case-insensitive, solution. For this example, it may take clicks on several results and ultimately query refinement to learn that one needs to specify, "case insensitive" to get better results.

An enriched, hybrid search approach matches code based on keywords and type signatures. This is allowed by the search engine, Merobase. For instance, the query, `boolean substr(String, String)`, representing the type signature of the desired method, was generated by one of the study participants. It returns results like the following:

```
1 boolean validatePassword(String inputPassword,
      String password) {
2     if(inputPassword.equals(password)){
3         return true;
4     } else {
5         return false;
6     }
7 }
```

Clearly, this result does not capture the concept of searching for a substring, even if the type signature is the same as the desired method.

In our approach, the programmer would specify input/output pairs for the desired code, illustrating how the code should behave. For this programming task, an example lightweight specification, *LS*, is shown in Fig. 1. Two input/output pairs form the specification (inputs and outputs are quoted and separated by braces), ls_a and ls_b,

[1] http://stackoverflow.com/questions/86780/is-the-contains-method-in-java-lang-string-case-sensitive .

```
1 boolean isReserved(String str) {
2         return reserved.contains(str.toLowerCase());
3 }
```

Fig. 2. The `isReserved` method.

```
1 boolean lastIndexOf(String str, int fromIndex,
      String stringToSearch) {
2         if(stringToSearch.lastIndexOf(str, fromIndex)
              == -1) {
3             return false;
4         }
5         return true;
6     }
```

Fig. 3. The `lastIndexOf` method.

```
1 LS = { ({"log.txt"}, {"log"}),          \\ ls1
2        ({"log.txt.txt"}, {"log.txt"})   \\ ls2
3        ({"log"}, {"log"}) }             \\ ls3
```

Fig. 4. Specification for trimming file extension.

and each contains two input strings and a boolean output, specifying examples of the desired behavior. For example, ''`PHA`'' is a case-insensitive substring of ''`alpha`'', so a relevant function should return `true`.

Also consider the relevant[2] code snippet in Fig. 2. Both specifications, ls_a and ls_b, are matched by the code in Fig. 2 (i.e., where "PHA" $\mapsto$ `str` and "alpha" $\mapsto$ `reserved` for ls_a, and where "bet" $\mapsto$ `str` and "beta" $\mapsto$ `reserved` for ls_b). However, the specification provides two input values whereas there is only one argument to the method in Fig. 2. Thus, the method cannot be executed; in order to identify this code as a match, the field `reserved` must be set to the first input value, so a testing approach may not be appropriate with this method in isolation. For the purposes of a testing approach, a simple analysis may suffice to reveal that that is the case for this example, but more sophisticated and expensive analyses and mocking would be needed to generalize such an approach.

The other case in which a testing approach is not appropriate is when the specification does not provide enough information. Consider the same problem of matching strings case insensitive, the specification in Fig. 1, and the source code in Fig. 3. Here, the method matches the specification for ls_b when "beta" $\mapsto$ `stringToSearch` and "bet" $\mapsto$ `str`. The variable, `fromIndex`, is not mapped to the specification. Without a value for the argument, a testing approach would not be able to consider this somewhat relevant method as a viable candidate for results. However, when using the solver, the value for `fromIndex` is merely constrained to be greater than zero (a requirement of the `lastIndexOf` method), and is set to `0` in the satisfiable model identified by the solver for ls_b. Thus, even with too little information in the specification, this method is found relevant by our approach.

2.2. Beyond single source line matches

Imagine that a programmer wants to find code to remove the file extension from a file name (this question was posed by a real developer on stackoverflow.com [3] and is used in our study – question 4 in Table 3).

An example lightweight specification generated by one of our study participants is shown in Fig. 4. The three input/output pairs, ls_1, ls_2, and ls_3, define values for an input string and a trimmed output string.

```
1  String stripExtension(String scriptFile) {
2      int extension = scriptFile.lastIndexOf('.');
3      if( extension > 0 ) {
4          int start = scriptFile.indexOf("/") + 1;
5          if(start > 0){
6              return scriptFile.substring(start,
                   extension);
7          }
8          return scriptFile.substring(0, extension);
9      } else {
10         return scriptFile;
11     }
12 }
```

Fig. 5. The `stripExtension` method.

Now consider a source code repository that includes the `stripExtension` method shown in Fig. 5.

Our previous approach (Stolee et al., 2014) only supported the matching of single-line snippets of source code. Using that approach, the code in Fig. 5 would have been split into five independent code snippets for lines 2, 4, 6, 8, and 10, none of which provides an adequate solution to the query.

The previous approach would encode each single-line snippet as a constraint. For instance, line 8 would be encoded as $\mathcal{R} = \mathcal{S}.sub(0, \mathcal{E})$ where *sub* denotes the `substring` function. Here $\mathcal{R}$ stands for the computed result – the return – and $\mathcal{S}$ and $\mathcal{E}$ for input values `scriptFile` and `extension`. The latter variables are *free* in the formulae describing program behavior to reflect the fact that no assumption is made about the input values.

At search time, these five snippets would be candidates for matching against the specification *LS*. Finding a match consists of checking for satisfiability of each snippet constraint set conjoined with a constraint encoding each element of the specification. For line 8 and ls_1, the resulting formula, $\mathcal{R} = \mathcal{S}.sub(0, \mathcal{E}) \wedge \mathcal{R} =$ "`log`" $\wedge\ \mathcal{S} =$ "`log.txt`" is satisfiable, when $\mathcal{E} \mapsto 3$, and thus ls_1 would match line 8. Similarly ls_2 and ls_3 would match line 8, when $\mathcal{E} \mapsto 7$ and $\mathcal{E} \mapsto 3$, respectively. Consequently line 8 would match for any single specification ls_i, but it would not be a match for the whole *LS* since the matching criteria requires that the value for any free variables, such as $\mathcal{E}$, be the same among all input/output pairs in a specification. That is, the code cannot be tweaked to satisfy all the input/output pairs simultaneously. Line 6 would not be judged as a match for the same reasons, lines 2 and 4 would not match since they produce integer outcomes, and line 10 would fail to match ls_1 and ls_2.

This single-line encoding has the advantage of simplicity and it can still match non-trivial snippets of code (e.g., compositions of string function calls). However, its value is limited by the inability to account for intermediate computation and non-trivial control flow.

2.3. Generalizing program matches

In this paper, we generalize the process of encoding programs by using symbolic execution (Clarke, 1976; Clarke and Richardson, 1985; King, 1976) to capture the behavior of all paths in a method. Fig. 6 depicts the symbolic execution tree for the code in Fig. 5. Each node in the tree corresponds to a branch in the program's execution where the right child denotes the true outcome and the left the false outcome. The dereference of `scriptFile` is an implicit branch, since a null value leads to throwing a `NullReferenceException`; the remaining branches are all explicit in the source code.

In the symbolic execution of a method, the input values, (e.g., parameter `scriptFile`), are assigned free-variables, (e.g., $\mathcal{S}$ in

[2] Relevance was determined by study participants. See Section 5.
[3] http://stackoverflow.com/questions/941272/.

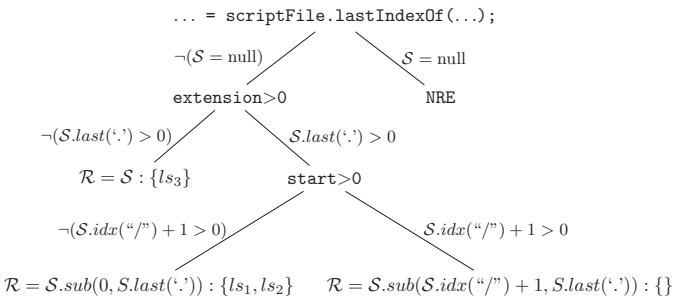

Fig. 6. Symbolic execution tree for Fig. 5.

Fig. 6); we again denote the return value with another variable, $\mathcal{R}$. For each branch outcome, a symbolic expression that encodes the constraints on the input values required to produce that outcome is recorded. For instance, the false branch of the test on line 5, `start >0`, has a constraint $\neg(S.idx(\text{“/”}) + 1 > 0)$ where *idx* is a short-hand for the string operation `indexOf`; similarly *sub* and *last* abbreviate `substring` and `lastIndexOf`, respectively. For a path in the tree to be executable the conjunction of its constraints – the *path condition* (PC) – must be satisfiable.

Along a path, assignment statements may set the value of variables that serve as outputs of a code fragment (e.g., return values, side-effected fields). We capture these *effect* constraints along with the PC. For instance on the leftmost path in Fig. 6 the `return` statement produces the effect constraint $\mathcal{R} = \mathcal{S}$.

To find a match with an input/output example, for each feasible path captured during symbolic execution, we conjoin the PC, the effect constraints, and a constraint encoding the binding of individual input/output pairs, and then check for satisfiability. Fig. 6 gives the set of satisfiable input/output pairs from *LS* for each path at every leaf node after the ":". For example, the leftmost path in the tree, corresponding to the return statement on line 10, has a PC of $\neg(\mathcal{S} = null) \wedge \neg(\mathcal{S}.last(\text{'.'}) > 0)$ and effects of $\mathcal{R} = \mathcal{S}$. For ls_3 we conjoin $\mathcal{S} = \text{“}\texttt{log}\text{”} \wedge \mathcal{R} = \text{“}\texttt{log}\text{”}$ and the result is satisfiable. Similarly we determine that ls_1 and ls_2 satisfy the constraints associated with the return at line 8.

Unlike the previous approach, a Java method here is represented as a set of paths, characterized as constraints, generated using symbolic execution. This allows the matching of larger, more complex snippets with multiple paths.

2.4. Matching levels and ranking

In the example described above, each $ls \in LS$ is matched to some path in `stripExtension`; thus, this method may be a relevant search result. The path leading to the return on line 6, the lowest and rightmost branch in the symbolic execution tree, is unsatisfiable when combined with any ls_i from the example. Thus, there exists a path in Fig. 5 that is not exercised by any of the input/output pairs in Fig. 4. While `stripExtension` is a match for *LS*, it *over matches* the specification by providing extra behavior that is unnecessary according to the specification.

If we were to strengthen *LS* by adding the input/output pair $ls_4 = (\{\text{“}\texttt{c : /abc.tex}\text{”}\}, \{\text{“}\texttt{abc}\text{”}\})$, then `stripExtension` would be judged a *full match* since ls_4 is satisfiable on the path leading to line 6, and then all paths and all input/output pairs would be matched.

Further strengthening *LS* by adding the input/output pair $ls_5 = (\{\text{“}\texttt{ftp : //path/abc.tex}\text{”}\}, \{\text{“}\texttt{abc}\text{”}\})$ would result in `stripExtension` being judged an *under match* since there is no path on which ls_5 is satisfiable, and thus the behavior of the method is less than what the specification states.

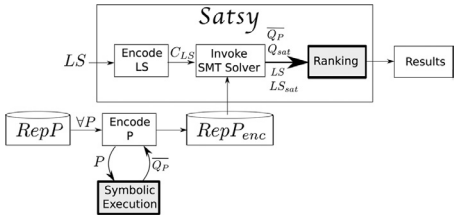

Fig. 7. Satsy and approach overview (new components are grayed, query matching is performed online, repository encoding offline).

In this paper we utilize the partiality of matching to rank potentially relevant source code results. With our more successful ranking algorithm (see Section 4.3), our basic intuition is to rank matches with extra behavior at the top (the specification model is weak, making it difficult to express all the desired behavior), followed by full matches. Matches with less behavior are not included as we expect that the user input/output pairs represent behavior that cannot be missed from the search code. It is also important to keep in mind that the degree of matching described here is from a program analysis perspective, and these notions may differ from a user's perspective on relevance. For this reason, the relevance of search results is evaluated by study participants.

We note that the example in this section, and the approach studied in detail in this paper, consider symbolic execution-driven semantic search in a restricted setting. While such methods can be handled by, for example, Symbolic PathFinder (SPF), the symbolic execution extension (JPF, 2012; Khurshid et al., 2003) to the Java PathFinder (JPF) model checker (Visser et al., 2003), symbolic execution tools come with limitations. For example, not all Java string API functions are fully supported in SPF yet. Generalizing the approach to index code that, for instance, manipulates data other than strings, invokes methods, and involves exceptional and iterative control flow, rests on the ability of symbolic execution engines to generate a sufficiently representative set of paths. Exploring those directions is in our future work (Section 9).

3. Approach

We present a brief definition of each piece of our original approach to semantic search via an SMT solver, depicted in Fig. 7. As this approach was first introduced in the context of programs with a single path (Stolee et al., 2014), and the Java specifications contained only single input values of type String, we now focus on generalizing the approach to richer specifications and how multi-path, non-looping programs are processed, highlighting the adaptations to the original definition.

3.1. Querying with input/output examples

This search approach takes lightweight specifications in the form of input/output examples that characterize the desired behavior of the code. These specifications, *LS*, are lightweight and incomplete in that they consist of concrete input/output pairs that exemplify part of the desired system behavior. As illustrated in Fig. 4, *LS* is defined as a set of input/output examples, $LS = \{ls_1, ls_2, \ldots, ls_k\}$, for k examples where $ls_j = (I_j, O_j)$. Each input, I_j, is a set of elements[4] and each output, O_j, is a set of elements related to I_j. Each element of the input and the output also has a defined type used to identify potentially matching programs.

[4] This is in contrast to our previous work where the input was a single element (Stolee et al., 2014).

```
1 int start;
2 int extension;
3 String scriptFile;
4 extension = scriptFile.lastIndexOf('.');
5 pc: extension > 0
6 start = scriptFile.indexOf("/") + 1;
7 pc: start <= 0
8 return scriptFile.substring(0,extension);
```

Fig. 8. Path q_2 for Fig. 5.

3.2. Indexing: from code to constraints

Source code is indexed by symbolically executing code and collecting constraints that represent the semantics, as illustrated in Section 2. The encoding process takes a collected set of independent programs $RepP = \{P_1, P_2, \ldots P_n\}$ and transforms it into a collection of independently-encoded programs, $RepP_{enc} = \{P_{1_{enc}}, P_{2_{enc}}, \ldots, P_{n_{enc}}\}$. At a high level, each encoded program P_{enc} is represented as a disjunction of its paths, and each path is represented in conjunctive normal form.

Since the specification model requires concrete input and output values, each will exercise a single path, as illustrated with the examples in Section 2. For each program $P \in RepP$, a symbolic execution engine traverses the paths in P so each can be encoded separately. We define $Q_P = \{q_1, q_2, \ldots, q_m\}$ as the m paths in P, and $\overline{Q_P} \subseteq Q_P$ as the set of paths computed by symbolic execution and retained by our encoding. For those paths where symbolic execution fails, we can backtrack and continue gathering information for other paths, leading to a partial description of the program. We currently filter out paths that end in an exception state since we conjecture developers almost exclusively search for positive examples of behavior, but this conjecture requires more study.

To illustrate, consider again the method from Fig. 5 whose symbolic execution tree is shown in Fig. 6. The process of symbolic execution records symbolic expressions for intermediate program variables in terms of the free input variables, e.g., $\mathcal{S}$. For instance, the assignment on line 2 sets the value of `extension` to be $\mathcal{S}.last('.')$. This allows subsequent references to variables along a path to express branch constraints in terms of symbolic expressions in order to construct a PC. In this example the true outcome of the branch at line 3 generates the constraint $\mathcal{S}.last('.') > 0$. Continuing along this path to the return at line 8, results in setting the value of `start` to $\mathcal{S}.idx("/") + 1$, at line 4, and then taking the false branch outcome at line 5, which contributes $\mathcal{S}.idx("/") + 1 \leq 0$ to the PC.

At a leaf in the tree there are two distinct sources of information, the PC and the effects – expressed as equality constraints on output variables. The PC defines the constraints on input values that cause the program path to execute, and the effects define the computed results. Along this path ending at line 8, the PC is: $\neg(\mathcal{S} = null) \wedge (\mathcal{S}.last('.') > 0) \wedge (\mathcal{S}.idx("/") + 1 \leq 0)$ and the symbolic expression is $\mathcal{S}.sub(0, \mathcal{S}.last('.'))$ which is equated to the return value $\mathcal{R}$. While not present in this example, variables can, of course, be assigned multiple times along a path. To record such updates, we simply create a new *version* of the assigned variable and use it subsequently. This produces what amounts to a static single assignment encoding of the path.

The tree in Fig. 6 has four paths. The first three, from left to right q_1, q_2, and q_3, encode the non-exceptional behavior of the method. For each such path our approach produces a path encoding as described above. Fig. 8 shows the path encoding resulting from this process for the second path ending at line 8 in the source code; since `scriptFile` is declared but never defined it is interpreted as a free input variable. Note how the encoding includes two sources of information, the path condition (lines 5 and 7) and the equality constraints (lines 4, 6, and 8). The fourth path, q_4, ends with a throw of a `NullReferenceException` and is filtered out. Here, $\overline{Q_P} = \{q_1, q_2, q_3\}$ and $Q_P \setminus \overline{Q_P} = \{q_4\}$.

In the end, an encoded program P_{enc} is a set of its encoded paths, where each path is represented in conjunctive normal form, $C_q = \{c_1 \wedge c_2 \wedge \ldots\}$. Encoding is performed for every path to create an encoded representation of P as a disjunction of its paths. In the example, $P_{enc} = \{C_{q_1} \vee C_{q_2} \vee C_{q_3}\}$.

3.3. Matching code to specifications

To support the search for programs with multiple paths, the constraint solver must be invoked on each path in a program, rather than each program as a whole (Stolee et al., 2014), to determine if the program matches a specification. The SMT solver returns three possible results, *sat*, *unsat*, or *unknown*. We say a path q satisfies a specification ls if $Solve(C_q \wedge C_{ls}) = sat$.

To find matches in a repository, the approach first checks for type compatibility between the path q and the specification ls. We define a type signature for an input/output pair ls as TS_{ls}. This is derived by replacing the concrete values in ls by their types. For example, TS_{ls_1} for Fig. 4 is represented as ({*String*}, {*String*}). The type signature for a path q, TS_q, is based on the parameters and return values for the method. The type signature for path q_2 in Fig. 8 is the same as for ls_1. To match code to a specification, it is required that $TS_q \supseteq TS_{ls}$. When $TS_q \supset TS_{ls}$, there exist extra parameters in q that are not specified in ls, such as the type signature ({*String, int, String*}, {*boolean*}) from the code in Fig. 3 compared to $TS_{ls_a} = (\{String, String\}, \{boolean\})$ from Fig. 1. Any extra parameters are left symbolic and the solver attempts to find a value for each such that ls is satisfied. Allowing the superset relationship in type matching is a form of *relaxed search*. Such relaxation highlights the solver's advantage over concretely executing the code with the input/output, since matches can be found when the specification is incomplete in terms of the type signature for the code.

The approach invokes the solver with $Solve(C_q \wedge C_{ls})$ for every $C_q \in P_{enc}$ such that $TS_q \supseteq TS_{ls}$ and for every $P_{enc} \in RepP_{enc}$.[5] When a specification contains multiple input/output pairs (i.e., $k > 1$), each pair is checked separately. A match is identified if at least one path $q \in \overline{Q_P}$ satisfies at least one input/output pair $ls \in LS$. For the following definitions, we assume type matching between ls and q.

Definition 1. A program P is a **match** for LS if

$$\exists q \in \overline{Q_P},\ \exists ls \in LS : Solve(C_q \wedge C_{ls}) = sat$$

The strength of a match is defined by how well a program P satisfies a specification LS. We define two properties to help describe the relationship between P and LS. The first, Q_{sat}, identifies the set of paths q in a program P such that at least one $ls \in LS$ returns *sat*:

Definition 2. Q_{sat}

$$= \{q \in \overline{Q_P} \mid \exists ls \in LS : Solve(C_{ls} \wedge C_q) = sat\}$$

The second property, LS_{sat}, identifies the set of input/output pairs ls such that at least one path $q \in Q_P$ returns *sat*:

Definition 3. LS_{sat}

$$= \{ls \in LS \mid \exists q \in \overline{Q_P} : Solve(C_{ls} \wedge C_q) = sat\}$$

3.4. Ranking results

Rather than string matching, structural properties, popularity, or usage information (Bajracharya et al., 2006; Holmes et al., 2006; Inoue et al., 2003; McMillan et al., 2011), the ranking approach we

[5] In the future, the solver could be invoked as $Solve((\bigvee ls \in LS) \wedge (\bigvee q \in \overline{Q_P}))$ to reduce solver calls on fully mismatched methods and potentially optimize the runtime of the search.

Table 1
High level ranking.

	$\mid LS_{sat} \mid = \varnothing$	$LS_{sat} \subset LS$	$LS_{sat} = LS$
$\mid Q_{sat} \mid = \varnothing$	–	–	–
$Q_{sat} \subset \overline{Q_P}$	–	Splintered	Over match
$Q_{sat} = \overline{Q_P}$	–	Under match	Full match

propose considers the strength of a match between a specification and a program when returning search results. The strength of the match is calculated using $\overline{Q_P}$, as it represents the paths we have access to and can handle.

For a program P and a specification LS, when $LS = LS_{sat}$, all $ls \in LS$ have a match in P. When $\overline{Q_P} = Q_{sat}$, all paths captured in P are covered by LS. In Section 2 we illustrated two partial matching conditions, an *over match* and an *under match*, to describe when a program P contains more or less behavior compared to LS.

Definition 4. P **over matches** LS when:

$$\exists q \in \overline{Q_P},\ \forall ls \in LS : \neg(Solve(C_q \wedge C_{ls}) = sat)$$

The example in Fig. 5 has extra behavior compared to the specification in Fig. 4 since the third path q_3 matches none of the $ls \in LS$. In this way, P is an *over match* for LS. The extra behavior may be irrelevant, or it might be desirable if the developer missed part of the specification. This example also illustrates the fact that specifications can contain redundancy, since both ls_1 and ls_2 return *sat* for q_2.

The relationship between LS and LS_{sat} can identify an instance of a *under match*:

Definition 5. P **under matches** LS when:

$$\exists ls \in LS,\ \forall q \in \overline{Q_p} : \neg(Solve(C_q \wedge C_{ls}) = sat)$$

As discussed earlier, considering LS in Fig. 4 and ls_4 and ls_5 from Section 2.4, the method in Fig. 5 matches only part of the specification. Hypothetically, if $\overline{Q_P} \subset Q_P$, the unmatched $ls \in LS$ may be satisfied by a path $q \in Q_P \setminus \overline{Q_P}$. When possible, executing the program given the unmatched specification could provide more information on the completeness of P with respect to LS.

Table 1 presents a matrix with property relationships that can be used to rank programs at a high level. If either $\mid Q_{sat} \mid = \varnothing$ or $\mid LS_{sat} \mid = \varnothing$, P is not a result and thus is not part of the ranking. In the former case, this means none of the program was matched; in the latter, none of the specification was satisfied. If $LS_{sat} = LS$ and $Q_{sat} = \overline{Q_P}$, P is a *full match*. When $Q_{sat} = \overline{Q_P}$ and $LS_{sat} \subset LS$, P is an *under match* as all paths are matched but only part of LS contributed to the match. When $LS_{sat} = LS$ and $Q_{sat} \subset \overline{Q_P}$, P is an *over match* as it contains additional behavior not explored by LS, but all of LS was matched. The final condition is a *splintered match*, which happens when $Q_{sat} \subset \overline{Q_P} \wedge LS_{sat} \subset LS$, so only part of P is matched by only part of LS. As explained in Section 2.4, our best ranking algorithm considers *over matches* followed by *full matches* (see Section 4.3).

4. Implementation

Satsy is a source code search engine that has implemented semantic search using input/output queries on multi-path programs with ranking, as illustrated in Fig. 7. Satsy is a Java application that runs on a standard desktop and interacts with a MySQL database containing the repository data. Compared to the previous version (Stolee et al., 2014), the primary additions described in this work are in bold, gray boxes (i.e., *Symbolic Execution* and *Ranking* in Fig. 7). Here, we describe the implementation details for Satsy.

4.1. Transforming source code with SPF

The repository of programs *RepP* (Fig. 7) is encoded to form $RepP_{enc}$. This encoding process is akin to indexing and happens offline; it does not impact the run-time of the search, thus the performance of symbolic execution does not impact the search performance. Regardless, it is efficient. For our study (Section 5), encoding 1000 programs takes approximately 4 min.

All programs $P \in RepP$ contain only constructs supported by a subset of the Java language covering conditionals, assignments, return statements, and boolean, integer, character, and string expressions including most of the `java.lang.String` API. Compared to the previous implementation (Stolee et al., 2014), this Satsy-supported subset of the Java grammar has the addition of predicates, conjunction and disjunction, relational expressions, multiplicative, additive, and modulo expressions, and additional String API functions: *equalsIgnoreCase, toLowerCase, toUpperCase, replace*, and *trim*. A full grammar specification for the Satsy-supported Java grammar is available [(Stolee, 2013) Fig. 6.3]. Loops and non-library method calls can be encoded by symbolic execution, e.g., by unrolling loops and inlining methods. In this work, we explore the cost-effectiveness of Satsy in a setting where symbolic execution can produce nearly complete characterizations of code, i.e., where $Q_P \setminus \overline{Q_P}$ is small. This led us to consider code that is free of loops and method calls.

Code snippets that Satsy can encode are removed from their original implementation and considered independently. That is, if a class has only one method that conforms to the Satsy-supported grammar, we keep that method and throw away the rest. If a field value is accessed, we declare it within the method. Methods that contain a conditional, conjunction, or disjunction are processed as multi-path programs using symbolic execution, as described in Section 3.2. This is done using a listener attached to SPF (JPF, 2012; Khurshid et al., 2003). The basic process involves two steps. First, Satsy writes a SPF driver to invoke the method under evaluation, since SPF can only analyze executable code and the methods are removed from their original implementation. Second, the listener is attached and traverses the paths in the method, recording path conditions and the executed statements along those paths, as illustrated in Section 2.3. The output from the listener is a set of paths $\overline{Q_P}$ for each program P.

Next, each path is encoded using constraints. This is done using a set of transformation rules that map program constructs onto constraints in SMT-LIB2 (SMT-LIB, 2012) format assuming the availability of the UFNIA: *Non-linear integer arithmetic with uninterpreted sort and function symbols* theory in the SMT solver.[6] These constraints are stored with the original method in a constraint database, forming $RepP_{enc}$.

4.2. Matching with Z3

Given LS from a programmer, Satsy first encodes the specification, forming C_{LS}. For each $C_{ls} \in C_{LS}$, Satsy joins the encoded specification with each path with a type signature match compared to the specification (i.e., $TS_q \supseteq TS_{ls}$) and invokes the Z3 SMT solver (Z3, 2011). The joining process involves mapping each of the inputs and outputs in each ls to the symbolic variables in the path q. When there are multiple inputs with the same data type and multiple free variables, all possible permutations of the mapping are considered. For example, given the specification in Fig. 3, both inputs are of type string. For the code snippet in Fig. 2, there are two free string variables, `str` and `reserved`. Thus, the joining process will create two possible mappings, either $(i_0 \mapsto str \wedge i_1 \mapsto reserved) \vee (i_0 \mapsto reserved \wedge i_1 \mapsto str)$. It is the solver's decision which mapping will lead to a satisfiable result.

[6] Full transformation details are available [(Stolee, 2013) Appendix A].

Table 2
Implemented ranking scheme.

	$\|LS_{sat}\| = 1$	$1 < \|LS_{sat}\| < \|LS\|$	$LS_{sat} = LS$
$Q_{sat} \subset \overline{Q_P}$	(10)	(7) $\|LS_{sat}\| \leq \|Q_{sat}\|$ (8) $\|LS_{sat}\| > \|Q_{sat}\|$	(3) $\|LS\| \leq \|Q_{sat}\|$ (4) $\|LS\| > \|Q_{sat}\|$
$Q_{sat} = \overline{Q_P}$	(9)	(5) $\|LS_{sat}\| \leq \|\overline{Q_P}\|$ (6) $\|LS_{sat}\| > \|\overline{Q_P}\|$	(1) $\|LS\| \leq \|\overline{Q_P}\|$ (2) $\|LS\| > \|\overline{Q_P}\|$

Satsy Round Robin (RR): 2, 1, 4, 3, 6, 5, 8, 7, 9, 10.
Satsy Ranked: 4, 2, 1, 3.

Performance information about the matching process can be found in Section 6.2.

As a sanity check during the matching phase, the executable snippets could be run against the specifications, as discussed in Section 2.1. However, as with the example described in Section 2.1, not all snippets matched with Z3 would also be matched through execution; this information may prove useful in future refinements of the ranking algorithm as executing matches may be preferred.

4.3. Ranking results

Satsy provides ranking algorithms with access to the matching information in Table 2, a refinement of what was presented in the quadrants of Table 1. The first refinement considers how saturated a program is by *LS*, as a finer-grain ranking consideration. If $|LS_{sat}| > |Q_{sat}|$, at least one of the matched paths is covered by two or more $ls \in LS_{sat}$ and the program is saturated by the specification. On the other hand, if $|LS_{sat}| \leq |Q_{sat}|$, each path is traversed by a single *ls*. Consider, for example, when $Q_{sat} = \overline{Q_P}$ and $LS_{sat} = LS$ both hold (lower right cell of Table 2). In bucket (2) there exists a path in *P* that is covered by multiple $ls \in LS$, so a program in bucket (2) is more saturated by *LS* than a program in bucket (1). The second refinement provides more granularity on the relationship between *LS* and LS_{sat} in the columns. We point out that all single-path programs fall into the bottom row of Table 2 since $Q_{sat} = \overline{Q_P}$ when *P* is a match and has one path. If $|LS| = |LS_{sat}| = 1$, the right-most column was assumed since it conceptually maps to a full specification match.

Satsy Round Robin (RR): Our baseline algorithm uses a Round-Robin approach to select a snippet randomly from each bucket, and then repeat until all results are used. This is done first with that $TS_q = TS_{ls}$, and followed by a relaxed search where $TS_q \supset TS_{ls}$. The ordering of buckets is listed in Table 2. The idea is that a more saturated program better 'fits' a specification influenced the ordering, so for example, bucket (2) appears before bucket (1).

Satsy Ranked: With the intuition that a programmer is likely to want their entire specification matched ($LS_{sat} = LS$), our Satsy Ranked algorithm uses only buckets (1)–(4) and requires $TS_q = TS_{ls}$. This intuition was built based on an initial study with Satsy RR and observing the precision of results. Unlike Satsy RR, here we use a greedy approach, grabbing the results from bucket (4) first, followed by bucket (2), and so forth.

5. Study

To evaluate Satsy, we designed and implemented an experiment to measure the relevance of search results from Satsy RR, a keyword-based search engine, Google, and a code-specific search engine, Merobase. After its completion, we replicated the study with Satsy Ranked. Our goal was to evaluate the research questions:

RQ1: *How do search results from Satsy compare to those found using Google and using Merobase, in terms of relevance, from the perspective of a programmer?*

RQ2: *How quickly can Satsy find results when linearly searching the encoded repository?*

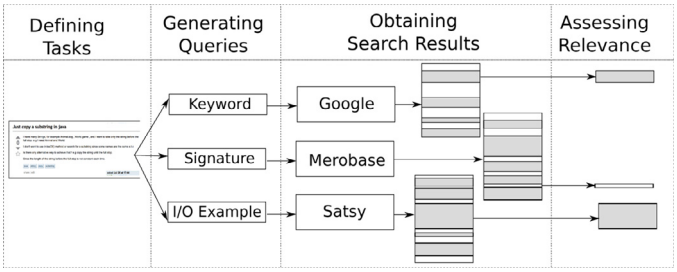

Fig. 9. Evaluation workflow.

Google was selected because it is the most common, and often effective, way in which programmers currently search for code (Sim et al., 2011; Stolee et al., 2014). Since Google was not specifically designed to search for source code, we also compare Satsy against Merobase, a code-specific search engine that has indexed over eight million Java components and allows programmers to search using the type signature of the desired code, which is the feature we explored in this evaluation.

5.1. Experimental design

An ideal evaluation of our search approach would involve issuing precisely the same queries to each search engine and for each search engine to retrieve results from the same repository. However, such a comparison is not possible as each search approach utilizes a different query format and we do not have access to their indexing and ranking algorithms. In our prior work (Stolee et al., 2014), we did a comparison of our search approach against Google using the same repository and found that our approach far outperformed Google in returning relevant results. Yet, we also recognize that Google was handicapped in that scenario since it did not have access to click data necessary for the PageRank algorithm (Page et al., 1999), thus treating Google as simply a textual search engine. To combat this bias, we have designed an experiment that uses queries in the formats required by each specific search engines and allows those search engines to return results from their own repositories using their own ranking algorithms. Fig. 9 depicts the workflow and the threats to validity are discussed in Section 7.3.

To obtain queries in each format while avoiding researcher bias, a group of human query generators evaluated each of several questions from stackoverflow.com and composed queries in each of three formats: keyword for Google, method signature for Merobase, and input/output for Satsy (*Defining Tasks* and *Generating Queries* in Fig. 9). *Obtaining Search Results* involved invoking each search approach with their respective queries and collecting the top 10 results. Then, each of the 10 results was evaluated by a human study participant for relevance (*Assessing Relevance*).

Treatment structure: Search queries were generated with respect to eight programming tasks or *questions* in a format dictated by the *search approach*. These form the two treatments that we manipulate in this experiment. The treatment structure is a 4×8 full factorial as every level of every factor is combined with every level of every other factor. The factors and their levels are:

A: Search approach (Google, Merobase, Satsy Round Robin, Satsy Ranked)
B: Question/programming task (8 questions)

Metrics: For *RQ1*, we use four precision metrics common in Information Retrieval: top-10 precision (P@10), normalized discounted cumulative gain (nDCG), mean average precision (MAP), the rank of the first false positive (f.f.p.), and the mean reciprocal rank (MRR). Descriptions and definitions of the metrics follow:

Table 3
Programming tasks for query generators.

Q	Description	Sample *LS*	Snippets
1	Check if one string contains another, case insensitive	({"*aBCd*", "*cd*"}, {*true*})	292
2	Capitalize the first letter of a string	({"*foo*"}, {"*Foo*"})	2575
3	Determine if a number is positive	({2}, {*true*}), ({−4}, {*false*})	820
4	Trim the file extension from a file name	({"*foo.txt*"}, {"*foo*"})	2575
5	Trim the last character from a string	({"*admirer*"}, {"*admire*"})	2575
6	Turn a string into a char	({"*c*"}, {'*c*'})	11
7	Determine if a character is numeric	({'5'}, {*true*}), ({'*F*'}, {*false*})	136
8	Check if one string is a rotation of another (a rotation is when the first part of a string is spliced off and tacked onto the end)	({"*stack*", "*cksta*"}, {*true*}), ({"*stack*", "*stakc*"}, {*false*})	292

P@10: This metric represents the number of relevant documents among the top 10 search results for each query (or in some cases, the top n results when fewer than 10 are returned); it is a typical IR measure to assess the precision of search engine results (Craswell and Hawkings, 2004). For each search query and each of the top 10 results, a study participant determined if it was relevant to the question, where relevance means *the source code can be easily adapted to solve the problem*. This is computed by:

$$P@10 = \frac{\sum_{k=1}^{n} rel_k}{n}$$

where n is the number of results (maximum 10) and rel_k is the relevance of the kth result. Relevance in our study is either 1 for relevant of 0 for irrelevant.

nDCG: This metric is sensitive to the ranking of the results and penalizes relevant documents that appear lower in the search list. We begin by computing the discounted cumulative gain (DCG) as follows:

$$DCG = rel_1 + \sum_{k=1}^{n} \frac{rel_k}{log_2(k)}$$

where k is the rank in the list and n is the number of results, maximum 10. Next, the results list is sorted so the relevant items appear first, forming the ideal ranking, and DCG is computed to create IDCG. The normalized metric is computed as follows:

$$nDCG = \frac{DCG}{IDCG}$$

MAP: Average precision is sensitive to the ranking and gives higher weight to relevant results appearing higher in the list. Average precision for a single query is computed as follows:

$$AveP = \frac{\sum_{k=1}^{n} (P(k) \times rel_k)}{number\ of\ relevant\ documents}$$

where $P(k)$ is the precision of the list at some rank k in the list. For instance, if a list has two relevant documents that appear at ranks 1 and 3, then $P(1) = 1.00$ and $P(3) = 0.67$. The mean average precision is for a set of queries and is computed as follows:

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$$

For our study, Q = 3 since we evaluated the results of three different queries in each search approach for each programming task.

f.f.p: The rank of the first false positive shows how far down the list a person must look to find a poor result. If all results are relevant, a value of n + 1 was assigned, since only the top n results were evaluated.

MRR: The mean reciprocal rank measures the inverse of the rank of the first positive result. This is calculated as follows:

$$MRR = \frac{\sum_{k=1}^{n} \frac{1}{rank_i}}{Q}$$

where $rank_i$ is the rank of the first relevant result. For example, if we have three queries where the first positive results appear at ranks 1, 2, and 4, $MRR = \frac{1+1/2+1/4}{3} = 0.58$.

With all metrics for *R1*, higher values are better.

For *RQ2*, we measure the time from search initiation with a given query until search completion (until the repository is exhaustively explored). When a result is found, we also compute the average time to find a result. For example, if it takes 100 s to find 15 results, the average time to find each result is 100/15 = 6.67s.

5.2. Defining programming tasks

Based on the hypothesis that certain search approaches are better suited for certain types of problems, we selected a variety of questions to explore the approaches' effectiveness under different contexts. To identify these questions, we performed a cursory manual analysis of 3500 stackoverflow.com questions tagged with **[java]**, ordered by popularity on April 16, 2013. Among the pool, we selected 13 questions that represent novice-level programming tasks and could be illustrated by input/output examples supported by our implementation (hence the narrow focus on string manipulation tasks), ignoring duplicates. In the end, eight questions were retained for the evaluation. These are the questions for which queries from human generators for Satsy Round Robin and Merobase returned at least 10 results; Google always returned 10 results, but since some results may not contain code, we ignore questions or queries that returned fewer than seven web pages with source code. The final set of questions, and sample *LS*, are shown in Table 3.

5.3. Generating queries

Twelve human query generators were used to generate queries that would be issued against each search engine in the context of the stackoverflow.com questions. These generators were graduate students and staff in Computer Science and Engineering at UNL. The generators were given paper packets with a page for each question, in random order. The question was stated at the top, followed by a box for a descriptive (keyword) query, a type signature for the desired code (for Merobase), and input/output pairs for Satsy. For example, given Question 4 in Table 3, a keyword query created by a human generator was, *"trim extension of a file name in java"*, a type signature query was, `removeExtension (String) : String;`, and an input/output query is shown in Fig. 4. Or, as illustrated in Section 2, given Question 1 in Table 3, a keyword query created by a human generator was, *"java string contain substring case insensitive"*, a type signature query was `boolean contains(String, String)`, and an input/output query included the examples, ({"*food*", "*foo*"}, {*true*}) and

({"*food*", "*f*99"}, {*false*}). In the end, we obtained 12 queries for each of the eight questions and each of the three search approaches.

Since each search result is evaluated by a study participant, to control the cost of the study, within each combination of search approach and question, we randomly sampled three queries from the 12 human generators. For internal consistency, Satsy RR and Satsy Ranked were treated the same.

5.4. Obtaining search results

With each query to each search engine, code snippets from the top 10 search results were obtained, as follows.

Google: Each of the queries to Google returned millions of results. For each of the top 10 results, we clicked through to the webpage and grabbed the first code snippet (i.e., block, line, method). Sometimes, results would not be provided in Java, but in another language like PHP or C#, and those snippets were still collected and provided as a potential result (frequently, study participants found those relevant). When a webpage had no code, we ignored it and returned fewer than 10 results. This happened for 9% of webpages explored from the Google search results (i.e., 22 of 240), so the average results per query was 9.1. In those cases, the metrics are calculated over the number of returned results. The alternative would have been to look beyond the top 10 results, which would violate the assumptions of our chosen metric.

Merobase: When searching the Merobase website, we restricted the results to Java using a flag in the search options. If a query included a method name, Merobase first returned methods with the same name and type signature, and then methods with just the same type signature. For each of the top 10 results, we clicked on the files and copied the method with the matched signature.

Satsy: For Google and Merobase, repositories already exist so obtaining results was a matter of issuing queries. For Satsy, we had to build the repository to search over. To do so, we collected an initial repository of programs by first scraping 2952 projects tagged as `[java]` from GitHub.com, a project hosting website. These projects were scraped on February 3, 2013, and represented about 10% of the total Java projects accessible through the website. We explored all the **.java* files, accounting for 197,473 files with over 700,000 methods. Of the files, 5506 contained at least one method Satsy could encode given its current supported Java semantics. We encoded 8430 methods in total with 9909 paths among the methods. Of the methods, 534 had multiple paths, with an average of 3.8 paths were captured per multi-path method. For example, with the method in Fig. 5, we captured three paths. Indexing this whole corpus took approximately 34 min. For each of the programming tasks in Table 3, we list the number of potentially matching snippets in the *Snippets* column when $TS_{ls} = TS_q$.

To gather search results, for each question and each input/output query, we searched Satsy using the repository. Then, using the ranking algorithms described in Section 4.3, we put together the top 10 results that would be evaluated. In Satsy Ranked, there were cases when 10 results were not returned; this happened for 19 of the 24 queries, and the total results evaluated were 130. As with Google, when fewer than 10 results were available, the metrics were calculated out of the number of returned results. For example, in Q1 with Satsy Ranked, two queries returned 10 results and one query returned two results. Thus, only 22 snippets were evaluated for this question, and P@10 is computed over the returned results. As an example, Fig. 2 shows a code snippet result from a query to Satsy related to Q1; it was ranked third.

As a point of comparison against the previous approach (Stolee et al., 2014), five of the eight sample *LS* would have been unsupported. That is, the prior work required a single input and single output, both of type string; the prior work would only be able to handle questions 2, 4, and 5. The extensions to the specification model used in this work allow for the rest of the questions to be evaluated.

5.5. Assessing relevance of results

After collecting the code snippets for all the queries, calculating rel_k, which is required for all the precision metrics, involves human participants evaluating each search result for relevance. We performed two iterations of this study. In the first, results from Google, Merobase, and Satsy RR were used. Then, a replication was performed using just Satsy Ranked.

A *basic task* presents a participant with a programming task description and three code snippets. An example snippet is shown in Fig. 5, which was found by a query to Satsy related to Q4 in Table 3. Each code snippet is accompanied by two questions. The first asks if the code is relevant to the task (yes/no response), and the second asks for a justification (free response). In the first iteration, each basic task contained one code snippet from each search approach. The ordering of search approaches was randomized within each basic task. The query from which the search result came was randomized, as was the rank of the snippet within the search results. The participant was not made aware of the search engine used to obtain the code, the rank of the result, or the query used. In the second iteration, all three results came from Satsy Ranked.

To recruit participants we deployed the assessment of relevance on Amazon's Mechanical Turk (Amazon Mechanical Turk, 2010). An *experimental task* is composed of eight *basic task*s, with one from each question in Table 3. The ordering of basic tasks within an experimental task was randomized. Each experimental task is implemented as a *human intelligence task*, or HIT. For the first iteration, we created 30 HITs, so each search result from each query appeared exactly once.[7] If fewer than 10 results were returned for a search approach, we inserted a dummy snippet that was not included in the results. For the second iteration, we created 10 HITs. When fewer than 10 results were returned for a query, we replicated the results randomly from within all queries for that question. Each HIT paid $3.25 and each participant could complete one HIT.

A prerequisite for participation was to pass a qualification test. This included four Java questions to ensure participants are reasonably competent with Java before participating.]Participants spent an average of 55 min to complete a HIT, which includes any pauses and distractions.

6. Results

Here, we present the results for our research questions. All snippets and results from the Mechanical Turk study are available.[8]

6.1. RQ1: relevance assessment

We computed all four metrics for each of the three queries from every combination of search approach and question, using both rankings for Satsy. Averages for P@10 are shown in Table 4, for nDCG in Table 5, for MAP in Table 6, for f.f.p in Table 7, and for MRR in Table 8. The search approaches are in the columns (*Google, Merobase, Satsy RR* and *Satsy Ranked*), and questions in the rows. For a given question, the highest value is bolded. The final row reports the column averages.

For three metrics, P@10, nDCG, and f.f.p, Satsy Ranked outperforms the other search approaches overall. For the other metrics, MAP and MRR, Google outperforms the other search approaches overall.

[7] 30 HITs * 8 questions * 3 snippets = 720 snippets.

[8] https://sites.google.com/site/semanticcodesearch/publications/generalizing-ranking/empirical-evaluation .

Table 4
P@10 responses from evaluation.

Q	Google		Merobase	Satsy RR	Satsy Ranked	
1	**0.63**		0.50	0.53	0.57	(22)
2	0.73	(28)	0.17	0.63	**1.00**[1]	(16)
3	**0.57**	(26)	0.37	0.40	0.40	(25)
4	0.67	(25)	0.33	0.67	**1.00**[1]	(19)
5	0.73		0.63	0.47	**1.00**[1]	(10)
6	0.60	(27)	0.60	**0.83**	0.67	
7	0.67	(26)	0.27	0.47	**1.00**[1]	(2)
8	**0.80**[a]	(26)	0.13	0.27	0.67[1]	(9)
Avg.	0.675		0.375	0.533	**0.709**	

Notes: The bold values denote the highest value per row.
[a] The top result for each query came from the original Stackoverflow question.
[1] The query from n user(s) returned 0 results.
(x) For some queries, fewer than 10 results are returned. The total snippets, summed over the three queries, is represented by x. By default, $x = 30$.

Table 5
nDCG from evaluation.

Q	Google	Merobase	Satsy RR	Satsy Ranked
1	0.86	0.81	0.82	**0.87**
2	0.90	0.49	0.81	**1.00**
3	**0.77**	0.58	0.70	0.61
4	0.99	0.64	0.87	**1.00**
5	0.91	0.91	0.69	**1.00**
6	0.75	0.80	**0.96**	0.74
7	0.91	0.51	0.67	**1.00**
8	0.91	0.45	0.53	**1.00**
Avg.	0.876	0.649	0.756	**0.902**

Note: The bold values denote the highest value per row.

Table 6
MAP from evaluation.

Q	Google	Merobase	Satsy RR	Satsy Ranked
1	**0.76**	0.66	0.65	0.49
2	0.82	0.27	0.74	**1.00**
3	**0.62**	0.48	0.60	0.38
4	0.97	0.52	0.80	**1.00**
5	0.87	0.82	0.56	**1.00**
6	0.65	0.65	**0.89**	0.63
7	0.84	0.35	0.53	**1.00**
8	**0.87**	0.24	0.38	0.67
Avg.	**0.800**	0.499	0.644	0.771

Note: The bold values denote the highest value per row.

Table 7
f.f.p (+ indicates a default value of 11 was used when no false positive was found in the top 10 results).

Q	Google	Merobase	Satsy RR	Satsy Ranked
1	**3.00**	2.33	2.67	1.00
2	+6.00	2.00	+5.00	**+11.00**
3	2.00	**3.33**	2.00	1.00
4	5.67	1.33	4.67	**+6.50**
5	6.00	5.00	2.00	**+8.50**
6	3.33	2.33	**4.67**	1.33
7	**2.67**	1.33	1.33	2.00
8	5.00	1.00	1.33	**7.00**
Avg.	4.208	2.333	2.958	**4.792**

Note: The bold values denote the highest value per row.

This is interesting as two of the metrics are sensitive to the ranking, MAP and nDCG, yet one favors Satsy Ranked and the other favors Google. Similarly f.f.p and MRR are dependent on the ranks of the first false positive and the first true positive, respectively, yet one favors Satsy Ranked and the other favors Google.

Table 8
MRR from Evaluation.

Q	Google	Merobase	Satsy RR	Satsy Ranked
1	**1.00**	0.83	0.83	0.42
2	0.83	0.31	**1.00**	**1.00**
3	0.83	0.45	**1.00**	0.40
4	**1.00**	0.61	**1.00**	**1.00**
5	**1.00**	0.83	0.53	**1.00**
6	0.58	0.67	**1.00**	0.67
7	0.83	0.50	0.58	**1.00**
8	**1.00**	0.22	0.48	0.67
Avg.	**0.885**	0.553	0.803	0.769

Note: The bold values denote the highest value per row.

For P@10 (Table 4), the highest is Satsy Ranked with $P@10 = 0.709$, followed by Google at 0.675[9], Satsy RR at 0.533, and Merobase at 0.375. Further, Satsy Ranked outperforms Google for 5/8 questions, and than Merobase for 8/8 questions. The story is similar for MAP (Table 6), except in that Google is the winner overall with $MAP = 0.800$ versus $MAP = 0.771$ for Satsy Ranked.

For nDCG, Google outperforms Satsy Ranked on only 2/8 questions. Overall the average across all questions for Google is 0.876 versus 0.902 for Satsy Ranked. Satsy Ranked also outperforms Google with f.f.p.

For MRR, there were many ties, notably for Q4 on which three of the four search approaches returned relevant results at rank 1 for all queries. While Google has the highest MRR value overall, Satsy RR has a higher overall value than Satsy Ranked. As shown in Table 2, Satsy Ranked puts results from bucket 4 ahead of results from buckets 1 and 2, which may have led Satsy Ranked to underperform on measures related to ranking.

In summary, it would seem that Satsy is more likely to return relevant results overall (P@10), but Google is more likely to rank a relevant result first (MRR). When Satsy Ranked finds results, it either has a very high MAP (Table 6, Q2, Q4, Q5, Q7), or it does quite poorly. We can see this in the range of MAP values, where Satsy Ranked has a large range, (0.38, 1.00), and Google has a much smaller range (0.65, 0.97).

Looking qualitatively at the questions, Satsy experienced the most success when the task required the output to be a modification of the input, rather than computing something new. Four of the top questions for Satsy, 2, 4, 5, and 6 (Satsy RR), involve an output that is a variant on the input (e.g., capitalize a letter, remove a suffix, trim the input, type conversion). Conversely, the lowest questions, 1, 3, and 8 involve computing something new based on the input (e.g., a boolean value). The exception is task 7, which computes something new, but Satsy Ranked performs very well.

Statistical analysis: As the experiment has a full factorial treatment structure, to understand why differences in the means were observed, we use a 2-factor ANOVA. Considering each of the metrics, the F-ratios are significant for the search factor, the question factor, and the interaction at $\alpha = 0.001$, indicating that the observed differences in means are not likely due to chance and depend on the search approach, the question selected, and the combination thereof. When Satsy does better for certain tasks and with the ranking algorithm we developed, this may not always be the case.

We continue the statistical analysis on the P@10 metric since it has one of the larger overall difference in values between the top two search approaches, with Satsy Ranked at 0.709 and Google at 0.693, providing a greater chance of finding a statistical difference.

Let μ_g be the average P@10 for Google, μ_{rr} for Satsy RR, μ_{sr} for Satsy Ranked, and μ_m for Merobase. Assuming non-normality of the data, we perform a test of means between Google and Satsy RR

[9] For Q8 and Google, all three queries found first the question on stackoverflow.com from which the task was generated, biasing toward Google.

Table 9
P@10 for Satsy Ranked with I/O breakdown; *with zeros* assigns a precision of 0 when no results are returned; *no zeros* ignores those queries.

	$\|LS\| = 1$	$\|LS\| = 2$	$\|LS\| = 3$	$\|LS\| = 4$
With zeros	0.344	0.498	0.536	0.636
Count	2	8	8	6
No zeros	0.344	0.796	0.857	0.636
Count	2	5	5	6

Table 10
Total search time per query, in seconds.

	Min	Mean	Median	Max	σ
Satsy RR	1.3	64.4	26.5	180.1	66.1
Satsy Ranked	0.4	42.8	29.8	113.8	43.0

using the Mann–Whitney Wilcox test and the null hypothesis, H_0: $\mu_g \geq \mu_{rr}$, is not rejected with $p = 0.9831$. Substituting μ_{sr} for μ_{rr}, the null hypothesis is also not rejected with $p = 0.3273$. Testing for equality of means, H_0: $\mu_g = \mu_{rr}$ is rejected at $\alpha = 0.05$ with $p = 0.0355$. However, we see that Google and Satsy Ranked are statistically equal, with H_0: $\mu_g = \mu_{sr}$ not being rejected as $p = 0.6545$. So, the ranking algorithm for Satsy provides an advantage, since Google is significantly different than Satsy RR, but not statistically different from Satsy Ranked. No difference is detected between Google and Satsy Ranked for the other metrics, either.

With no difference between Google and Satsy Ranked, we turn to Merobase and compare it to Satsy RR (its closest competitor), with the null hypothesis, H_0: $\mu_m \geq \mu_{rr}$. The null hypothesis is rejected at $\alpha = 0.05$ with $p = 0.0163$. Substituting μ_{sr} for μ_{rr}, the null hypothesis is rejected at $\alpha = 0.001$ with $p = 0.0002$. The conclusion here is that Satsy is significantly more effective at retuning relevant results than Merobase. *In conclusion, Satsy Ranked returns significantly more relevant results than Merobase and results at least as relevant as Google (but complementary in many cases, depending on the task).*

Sensitivity to I/O sizes: Using input/output queries provides the opportunity to manipulate the strength of a query by adding, removing, or modifying input/output pairs. In the evaluation, the human query generators were able to specify as many input/output pairs as they wished and each used between one and four pairs. Table 9 shows the average precision based on the specification size for Satsy Ranked.

The table is separated based on whether or not a precision of zero was used in the averages for queries that returned zero results. Overall, six queries had four input/output pairs. For those queries, the average precision is 0.636, which is the highest average relevance for all the search approaches when zeros are considered. The highest precision among the queries that returned results comes from those queries with three input/output pairs where precision is 0.857. From this, precision seems quite sensitive to the number of input/output pairs. *With larger, but still modestly sized specifications, Satsy does better.*

6.2. RQ2: performance assessment

We measured the performance of Satsy as is, without any special effort allocated to performance enhancements, tweaking the solver parameters, or modifying SPF, and running the queries in Eclipse serially on a standard laptop. Total search times for each of the 24 queries are shown in Table 10, averaged over three runs. The performance of Satsy Ranked is on average better than Satsy RR since Satsy RR considers subset relationships on the type signatures (Section 4.3) and thus checks on average 16% more snippets per query.

Since Satsy Ranked had the highest precision, the rest of this analysis focuses on it. If a result exists using Satsy Ranked, it will be found

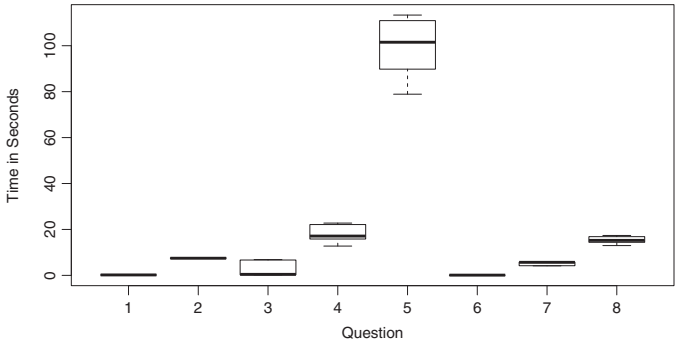

Fig. 10. Time to find a result in Satsy Ranked .

in an average of 16.2 s with a median of 5.7 s. The average is skewed due to poor performance on Q5; Fig. 10 shows the average time to find a result for each of the questions. Several factors influence the search time, including the type and size of the specification and the size and content of the repository.

There is a weak but positive correlation between the specification size and the total search time, with Spearman's $r = 0.21$ for Satsy Ranked. Interestingly, the data type for the query impacts the search time. Half of the queries were specified with a string output (Q2, 3, 5, 6) and half with a boolean output (Q1, 4, 7, 8). In Satsy Ranked, finding a result takes 2.5 times longer with a string output than boolean, averaging 23.8 and 9.7 s, respectively. This is likely due to a combination of factors, including how strings are represented in our encoding, and the fact that the boolean data type is native to the UFNIA theory used by the solver, whereas the String data type was implemented using uninterpreted functions.

The time to find a result also depends on the size and content of the repository. For example, the larger the repository, the more candidate snippets for a given query. Similarly, the content of the repository makes a big difference; if there is only one result, in the worst case, every candidate snippet must be checked before the winner is encountered. In the average case, half of the snippets must be checked. This is likely why the performance on Q5 was so poor; for each of the two queries that returned results with Satsy Ranked, only one result out of 2,575 snippets (Table 3) was found. Q4 and Q2 have the same number of potential snippets, but there were between 5 and 13 results for each of those queries in Satsy Ranked. While Q7 and Q8 have similar solution densities in the repository, their output data types are booleans, leading to faster solver times. Thus, the search time is dependent on when that snippet is checked. On average, it takes 21.8 ms for the SMT solver to make a decision about whether or not a snippet is a match for an input/output pair.

7. Discussion

Code search does not have a one-size-fits-all solution as programmers have various goals and purposes for searching for source code. For code searches with the goal of reuse and where the desired behavior can be expressed in terms of inputs and outputs, Satsy provides a compelling alternative or complement to the state-of-the-practice.

The judges of the relevance of search results are users, which is why we turned to humans to evaluate relevance. For Satsy Ranked, we observe that the average relevance is higher than that for Google, Merobase, and Satsy RR. Of course, we cannot ignore the fact that the query model necessary for Satsy is drastically different from that for Google. The user cost of changing the query model has yet to be evaluated and is left for future work. That said, the costs for evaluating Google results include clicking through to the result page and locating the code; these costs are not captured in the evaluation, either.

Please cite this article as: K.T. Stolee et al., Code search with input/output queries: Generalizing, ranking, and assessment, The Journal of Systems and Software (2015), http://dx.doi.org/10.1016/j.jss.2015.04.081

7.1. Opportunities

There are opportunities with this work to impact and benefit from other applications of symbolic execution. Satsy could be adapted to take advantage of recent techniques that store path conditions from symbolic execution in a repository (Visser et al., 2012; Yang et al., 2012), though additional meta-data might be needed to support Satsy. Other symbolic execution application, such as test case generation, (e.g., Cadar et al., 2008; Tillmann and De Halleux, 2008; Visser et al., 2004), could also leverage such repositories thereby amortizing the cost of symbolic execution.

Programs that under-match a specification provide an opportunity for program synthesis approaches (e.g., Gulwani et al., 2011; Harris and Gulwani, 2011; Solar-Lezama et al., 2006) to bring programs closer to the specification. Additionally, test generation techniques that use the control flow graph could be used to create test case that cover parts of the graph, or cover the graph in particular ways, such as using edge-pair or prime-path coverage (Ammann and Offutt, 2008). Providing these additional test cases to the user would allow them to better evaluate if a particular snippet is appropriate for their application. Alternatively, pruning the control flow graph, and thus the code, could create a minimal solution to a query.

Another opportunity comes from automated reuse, where the input/output query could be extracted from a test case and the search automatically plugs in the top-ranked code snippet. This form of automated test-driven development could be promising as the language coverage of our approach grows.

Exploring these opportunities is part of our future work.

7.2. Limitations

Our work is subject to several limitations, both in the implementation presented here and also in a more general, practical sense.

A major limitation in operating Satsy for our study was the repository size. We collected fewer than 10,000 methods (Section 5.4), and considering the sizes of the projects we scraped, these numbers seem low. We identify two reasons for this. The first is that our encoding is missing some commonly-used constructs, specifically loops, objects, and the value, *null*. The second reason follows from some limitations of SPF. String processing is incomplete, meaning errors are thrown on some multi-path methods that match the Satsy-supported Java grammar, and thus those methods are not encoded and stored. A second limitation comes from the use of Z3 as a prover, since we are limited by its capabilities. For instance, integer overflow is handled by returning *unsat* when an integer exceeds 32-bits.

A limitation of the approach in general relates to scaling to larger, more complex pieces of code. As the number of paths in a program increases (e.g., due to loops or nested if-statements), the number of paths that need to be checked with the solver increases, impacting the search performance. This may necessitate heuristics to limit the number of paths, which would lead to an incomplete approach that could miss matches. Careful study is needed to weigh the tradeoff between the search time and the completeness of the search algorithm.

7.3. Threats to validity

The evaluation required many steps and processes and each introduces threats to validity.

Internal: When building the repository of encoded programs, we remove each code snippet from its context so the encoded programs may not be entirely representative of the actual behavior of the original full code. For example, if a method accesses a field and it has a static value defined within a class, we ignore that value and represent the field value symbolically. Addressing this threat and considering a broader scope of code snippets is part of future work.

Google, Merobase, and Satsy returned matches from separate repositories of varying sizes and content, yet we compared the relevance of the results directly. Google searches over a web-scale repository and Merobase over a very large source code repository. Allowing those search engines to return code that Satsy couldn't support was a design choice to preserve external validity.

Questions used in the evaluation were selected if they could be represented with input/output examples, which may have favored Satsy. Yet, the questions came from StackOverflow, so Google has access to community-accepted answers. Questions were also selected if searches returned enough results (Section 5.2). This decision may have favored Satsy or Google as no questions were rejected on account of the Merobase results.

The performance of Satsy was measured on a standard laptop rather than using a large distributed computing infrastructure, as is done with the compared search approaches. Thus these measures are likely not representative of performance if adapted and deployed for public use.

The query generators were graduate students in computer science and were accustomed to writing Google queries. This may have led to higher quality queries for Google compared to Merobase and Satsy.

External: We chose to compare Satsy to Google and Merobase, under the assumption that tools such as these are used for code search in practice. Google is reported to be the most common tool used for code search (Sim et al., 2011; Stolee et al., 2014), but the actual usage of Merobase is unknown.

The selection of programming tasks may not be representative of tasks for which programmers would search for answers. To increase the potential for using representative questions, we selected questions from stackoverflow.com.

In the evaluation, we separated the query generator from the person who is evaluating the relevance of the results. In practice, these two roles are held by the same person with the same context. Here, the query generators may have interpreted the question differently than the study participants who judged the relevance. To combat this, three different query generators were assigned to each question/approach combination.

Construct: We use *P*@10, *nDCG*, *f.f.p*, *MRR*, and *MAP* as metrics to indicate relevance of search results. Yet, these might not capture other aspects of search, such as the cost of looking at irrelevant results or the search latency. Further study is needed to understand the cost tradeoffs between using the three search approaches.

Conclusion: The relevance of each source code snippet was judged by a Mechanical Turk participant based on a static view of the source code. Their judgment of relevance may not be reliable which is why we compute and report average relevance among the top 10 results per query.

8. Related work

To our knowledge, this is the first work to apply symbolic execution to the code search problem and to use semantic levels of matching in a ranking algorithm for search results. Our previous work introduced this approach to code search but targeted single-path programs (Stolee and Elbaum, 2012; Stolee et al., 2014). The differences between the previous work and this work were made explicit earlier in the paper.

Many code search tools have been proposed and evaluated. Several search approaches use keyword queries and rank results based on component usage or structural information (Bajracharya et al., 2006; Grechanik et al., 2010; Inoue et al., 2003; McMillan et al., 2011). Sourcerer searches open source code and exploits structural and usage information to rank results (Bajracharya et al., 2006). Portfolio uses visualization in the results to illustrate how the code is used and ranks code based on the call graphs (McMillan et al., 2011). Exemplar takes a natural-language query and uses information retrieval

Please cite this article as: K.T. Stolee et al., Code search with input/output queries: Generalizing, ranking, and assessment, The Journal of Systems and Software (2015), http://dx.doi.org/10.1016/j.jss.2015.04.081

and program analysis techniques to retrieve relevant applications (Grechanik et al., 2010).

More relevant to our approach are the code searches that target a specific code search sub-problem and include non-standard query models (Milne and Rowe, 2012; Sahavechaphan and Claypool, 2006). XSnippet allows programmers to search over a repository for sample code related to object instantiation (Sahavechaphan and Claypool, 2006). The query model exploits the type and hierarchy of the provided object and matches are based on mining code for instantiations of that object. Another approach focuses on searching for source code with specific API usage (Milne and Rowe, 2012). Queries use a partial program and an automata-based approach finds source code based on mined temporal specifications.

Other research has sought to recommend code-related web pages (Sawadsky et al., 2013) or to augment web searches with context from a programming environment (Brandt et al., 2010). Our search approach operates under a similar assumption, that programmers frequently search the web for source code, except we return source code relevant to a behavioral example.

In semantic code search, our work is most related to approaches that use theorem provers to identify relevant components (Penix and Alexander, 1999; Zaremski and Wing, 1997) and those that use test cases to execute against source code (Lemos et al., 2007; Podgurski and Pierce, 1993; Reiss, 2009). Compared to the former, our work differs in the use of a lightweight, input/output specification model rather than formal logic. Compared to the latter, our work differs in that we do not execute the code, allowing us to identify *close* matches by applying abstractions to the constraints (Stolee et al., 2014) or by considering code that has extra parameters (i.e., when $TS_q \supset TS_{ls}$, Section 3.3).

Also close to our work is program synthesis that uses solvers to derive a function that maps an input to an output (e.g., Gulwani, 2011; Gulwani et al., 2011; Harris and Gulwani, 2011; Jha et al., 2010; Singh and Solar-Lezama, 2011). The domains of applicability include string processing (Gulwani, 2011), spreadsheet table transformations (Harris and Gulwani, 2011), a domain-specific language for geometric constructions (Gulwani et al., 2011), data structure manipulations (Singh and Solar-Lezama, 2011), and loop-free bit manipulation programs (Jha et al., 2010). Some of these approaches also utilize supplementary information like the code structure to help guide the solver (Singh and Solar-Lezama, 2011). Another means for program synthesis utilizes partial programs called sketches (e.g., Raabe and Bodik, 2009; Solar-Lezama et al., 2007, 2008, 2006). These approaches typically operate over small, low-level and finite programs like bit vector manipulation (Solar-Lezama et al., 2006) and hardware circuits (Raabe and Bodik, 2009), but more recently have targeted higher-level applications such as concurrent data structures (Solar-Lezama et al., 2008) and optimizing code (Solar-Lezama et al., 2007). Recent efforts add context from a program execution using breakpoints and suggests snippets of code to integrate into a code base (Galenson et al., 2014). The key difference between program synthesis and our approach is that we use the solver to find a match against real programs that have been encoded, while these synthesis efforts must define a domain specific grammar that can be traversed exhaustively to generate a program that matches the programmers' constraints.

9. Conclusion

We have implemented and evaluated a significant extension to a code search approach that uses input/output queries and identifies results using an SMT solver. This extension, implemented in our tool Satsy, takes advantage of symbolic execution to traverse and identify feasible paths in a multi-path method and uses that information to identify partial matches and rank search results. We have shown that Satsy returns more relevant results than Merobase and results at least as good as Google, based on the responses of 30 study participants. We have also started to explore the execution cost of Satsy, which is clearly more expensive than existing approaches but still viable given the precision of the results.

There are multiple avenues for improving Satsy: extending the encoding to consider loops, method calls, nulls, exceptions, objects and arrays, strengthening the specification model by allowing wildcards (e.g., `(E.txt, E)` where `E` is any string) or regular expressions, allowing for cross-snippet matches where multiple programs could be returned if they matched different parts of the same specification, extending the size and richness of the repository, increasing the efficiency of the implementation to put it in the hands of developers, and refining the ranking algorithm based on larger data sets. We speculate that addressing these issues in future work will lead to versions of Satsy that will outperform keyword-based queries for certain kind of code search problems that can be described succinctly with an example.

Acknowledgements

This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, NSF GRFP under CFDA-47.076, the Harpole-Pentair endowment at Iowa State University, a Google Faculty Research Award, and AFOSR #9550-10-1-0406.

References

Amazon Mechanical Turk Command Line Tool Reference. http://docs.amazonwebservices.com/AWSMturkCLT/2008-08-02/, January 2010.

Ammann, P., Offutt, J., 2008. Introduction to Software Testing. Cambridge University Press, Cambridge, UK.

Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C., 2006. 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, pp. 681–682.

Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R., 2010. Example-centric programming: integrating web search into the development environment. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10. ACM, New York, NY, USA, pp. 513–522.

Cadar, C., Dunbar, D., Engler, D., 2008. 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, pp. 209–224.

Clarke, L.A., 1976. A system to generate test data and symbolically execute programs. IEEE Trans. Soft. Eng. SE-2 (3), 215–222.

Clarke, L.A., Richardson, D.J., 1985. Applications of symbolic evaluation. J. Syst. Softw. 5 (1), 15–35.

Craswell, N., Hawkings, D., 2004. Overview of the rrec 2004 webl track. In: Proceedings of the 13th Text Retrieval Conference. NIST, pp. 1–9.

Fischer, G., Henninger, S., Redmiles, D., 1991. Cognitive tools for locating and comprehending software objects for reuse. In: Proceedings of the 13th International Conference on Software Engineering, pp. 318–328.

Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K., 2014. Codehint: dynamic and interactive synthesis of code snippets. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York, NY, USA, pp. 653–663.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., Cumby, C., 2010. A search engine for finding highly relevant applications. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 1, ICSE '10. ACM, New York, NY, USA, pp. 475–484.

Gulwani, S., 2011. Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11. ACM, New York, NY, USA, pp. 317–330.

Gulwani, S., Korthikanti, V.A., Tiwari, A., 2011. Synthesizing geometry constructions. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11. ACM, New York, NY, USA, pp. 50–61.

Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., Menzies, T., 2013. Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 842–851.

Harris, W.R., Gulwani, S., 2011. Spreadsheet table transformations from examples. SIGPLAN Not. 46 (6), 317–328.

Holmes, R., Walker, R.J., Murphy, G.C., 2006. Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans. Soft. Eng. 32 (12), 952–970.

Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S., 2003. Component rank: relative significance rank for software component search. In: Proceedings of the 25th International Conference on Software Engineering, ICSE '03. IEEE Computer Society, Washington, DC, USA, pp. 14–24.

Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A., 2010. 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, pp. 215–224.
JPF Symbolic PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc, December 2012.
Khurshid, S., Păsăreanu, C.S., Visser, W., 2003. Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'03. Springer-Verlag, Berlin, Heidelberg, pp. 553–568.
King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.
Lemos, O.A.L., Bajracharya, S.K., Ossher, J., 2007. Codegenie: a tool for test-driven source code search. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA '07. ACM, New York, NY, USA, pp. 917–918.
McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C., 2011. Portfolio: finding relevant functions and their usage. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11. ACM, New York, NY, USA, pp. 111–120.
Milne, I., Rowe, G., 2002. Difficulties in learning and teaching programming - views of students and tutors. Educ. Inf. Technol. 7 (1), 55–66.
Mishne, A., Shoham, S., Yahav, E., 2012. Typestate-based semantic code search over partial programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12. ACM, New York, NY, USA, pp. 997–1016.
Page, L., Brin, S., Motwani, R., Winograd, T., 1999. The pagerank citation ranking: bringing order to the web.
Penix, J., Alexander, P., 1999. Efficient specification-based component retrieval. Autom. Softw. Eng. 6, 139–170.
Podgurski, A., Pierce, L., 1993. Retrieving reusable software by sampling behavior. ACM Trans. Softw. Eng. Methodol. 2, 286–303.
Raabe, A., Bodik, R., 2009. Synthesizing hardware from sketches. In: Proceedings of the 46th Annual Design Automation Conference, DAC '09. ACM, New York, NY, USA, pp. 623–624.
Reiss, S.P., 2009. Semantics-based code search. In: Proceedings of the International Conference on Software Engineering, pp. 243–253.
Sahavechaphan, N., Claypool, K., 2006. Xsnippet: mining for sample code. SIGPLAN Not. 41 (10), 413–430.
Sawadsky, N., Murphy, G.C., Jiresal, R., 2013. Reverb: recommending code-related web pages. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, NJ, USA, pp. 812–821.
Sim, S.E., Umarji, M., Ratanotayanon, S., Lopes, C.V., 2011. How well do search engines support code retrieval on the web? ACM Trans. Softw. Eng. Methodol. 21 (1), 4:1–4:25.
Singh, R., Solar-Lezama, A., 2011. Synthesizing data structure manipulations from storyboards. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11. ACM, New York, NY, USA, pp. 289–299.
Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S., 2007. Sketching stencils. SIGPLAN Not. 42 (6), 167–178.
SMT-LIB, December 2012.
Solar-Lezama, A., Jones, C.G., Bodik, R., 2008. Sketching concurrent data structures. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08. ACM, New York, NY, USA, pp. 136–148.
Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V., 2006. Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII. ACM, New York, NY, USA, pp. 404–415.
Stolee, K.T., 2013. Solving the Search for Source Code (Phd thesis). University of Nebraska–Lincoln.
Stolee, K.T., Elbaum, S., 2012. Toward semantic search via SMT solver. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12. ACM, New York, NY, USA, pp. 25:1–25:4.
Stolee, K.T., Elbaum, S., Dobos, D., 2014. Solving the search for source code. ACM Trans. Softw. Eng. Methodol. 23, 26:1–26:45.
Tillmann, N., De Halleux, J., 2008. Pex: White box test generation for .net. In: Proceedings of the 2nd International Conference on Tests and Proofs, TAP'08. Springer-Verlag, Berlin, Heidelberg, pp. 134–153.
Visser, W., Geldenhuys, J., Dwyer, M.B., 2012. Green: reducing, reusing and recycling constraints in program analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12. ACM, New York, NY, USA, pp. 58:1–58:11.
Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F., 2003. Model checking programs. Autom. Softw. Eng. 10 (2), 203–232.
Visser, W., Păsăreanu, C.S., Khurshid, S., 2004. Test input generation with java pathfinder. SIGSOFT Softw. Eng. Notes 29 (4), 97–107.
Yang, G., Păsăreanu, C.S., Khurshid, S., 2012. Memoized symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012. ACM, New York, NY, USA, pp. 144–154.
Z3: Theorem Prover. http://research.microsoft.com/projects/z3/, November 2011.
Zaremski, A.M., Wing, J.M., 1997. Specification matching of software components. ACM Trans. Softw. Eng. Methodol. 6, 333–369.

Kathryn T. Stolee is the Harpole-Pentair Assistant Professor in the Department of Computer Science and the Department of Electrical and Computer Engineering at Iowa State University. She received her PhD degree in computer science in 2013 from the University of Nebraska-Lincoln, where she also received her MS and BS degrees in computer science. Her research uses program analysis to develop tools and techniques with the goal of making software easier to build, maintain, and understand.

Sebastian Elbaum is a Professor at the University of Nebraska-Lincoln. He received the systems engineering degree from the Universidad Catolica de Cordoba, Argentina, and the PhD degree in computer science from the University of Idaho. His research aims to improve software dependability through testing, monitoring, and analysis. He is the recipient of the US National Science Foundation (NSF) Career award, an IBM Innovation award, and two ACM SigSoft Distinguished Paper awards.

Matthew B. Dwyer is the Lovell Professor and Chair of the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. He earned a Doctorate in computer science from the University of Massachusetts at Amherst in 1995 for work on data flow analysis of correctness properties of concurrent software – a topic that still interests him to this day. Dr. Dwyer is an ACM Distinguished Scientist (2007), a Fulbright Research Scholar (2011), and an IEEE Fellow (2013). He has chaired the PC of FSE, ICSE, and OOPSLA and serves as Editor-in-Chief of *IEEE Transactions on Software Engineering*.

Please cite this article as: K.T. Stolee et al., Code search with input/output queries: Generalizing, ranking, and assessment, The Journal of Systems and Software (2015), http://dx.doi.org/10.1016/j.jss.2015.04.081