2012

# Solving Semantic Searches for Source Code

Kathryn T. Stolee
*University of Nebraska-Lincoln*, kstolee@cse.unl.edu

Sebastian Elbaum
*University of Nebraska-Lincoln*, selbaum@virginia.edu

Daniel Dobos
*University of Nebraska – Lincoln*, ddobos@cse.unl.edu

# Solving Semantic Searches for Source Code

Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos
Department of Computer Science and Engineering
University of Nebraska – Lincoln
{kstolee, elbaum, ddobos}@cse.unl.edu

***Abstract*—Programmers search for code frequently utilizing syntactic queries. The effectiveness of this type of search depends on the ability of a programmer to specify a query that captures how the desired code may have been implemented, and the results often include many irrelevant matches that must be filtered manually. More semantic search approaches could address these limitations, yet the existing approaches either do not scale or require for the programmer to define complex queries. Instead, our approach to semantic search requires for the programmer to write lightweight, incomplete specifications, such as an example input and expected output of a desired function. Unlike existing approaches to semantic search, we use an SMT solver to identify programs in a repository, encoded as constraints, that match the programmer-provided specification. We instantiate the approach on subsets of the Java string library, Yahoo! Pipes mashup language, and SQL select statements, and begin to assess its effectiveness and efficiency through evaluations in each domain.**

## I. Introduction

Today, searching for code is a regular activity for most programmers [21]. Yet, the mechanisms to support this activity have barely evolved in the last decade, and the limitations are becoming more apparent as code repositories get richer and programmers' expertise and needs more diverse.

Consider a novice Java programmer who is trying to find a snippet of code that extracts an alias from an e-mail address. The programmer turns to Google (like many others [21]) and issues a search query with the following keywords: *extract alias from e-mail address in Java*. As expected, the query returns millions of results. None of the top ten results (a typical IR measure to assess the precision of search engine results [5]), even provide a method for decomposing an e-mail address into parts, which is the first step towards extracting the alias. Now, if the programmer is knowledgable enough about the domain to refine the query with the term *substring*, then the top ten results include two relevant solutions. This illustrates what occurs in practice, where programmers must sift through many irrelevant results, especially when the desired behavior cannot be tied to source code syntax or documentation.

Our work targets this limitation. The general idea is that programmers provide concrete behavioral specifications as inputs and outputs and an SMT solver identifies available source code, encoded as constraints, that matches the specifications.

For example, when searching for a program that extracts the alias from an e-mail address, the input could be the string "susie@mail.com" and the output the string "susie". This form of query, while more costly than a keyword query, lets the programmer specify the desired behavior, without the need to know *how* to achieve a certain outcome, just *what* that outcome is.

Just like any search engine, our approach indexes a repository of information offline, independently of the user. Our indexing is unique in that it requires an engine that maps a program's semantics onto constraints that summarize the program behavior. For example, the indexing process would map the Java snippet:
`output=input.substring(0, input.indexOf('@'))`
into the following constraints (roughly):

```
c1. (assert (input.charAt(end) == '@'))
c2. (assert (for (0 <= i < end)
     output.charAt(i) == input.charAt(i))
c3. (assert (for (0 <= i < end)
     input.charAt(i) != '@')
```

Constraint `c1` defines `end` as the location of '@' in `input` and `c2` asserts that the `output` matches `input` within bounds. Constraint `c3` asserts `end` is the first index of '@' in `input`. The `substring` operation is achieved by the conjunction of `(c1 ∧ c2 ∧ c3)`. This is the basic process by which our approach indexes programs: mapping program semantics to constraints.

With an input/output query and an encoded repository of programs, the search can now find results. The first step in this phase consists of transforming the provided input/output into additional constraints. For the previous example, that would be:

```
c4. (assert (input == "susie@mail.com"))
c5. (assert (output == "susie"))
```

The second step consists of pairing the input/output constraints with each of the programs indexed in the repository, and using an SMT solver to identify which pairs are satisfiable and hence constitute a match. For our email alias example, an SMT solver would return *sat* for the snippet encoded through constraints `(c1,c2,c3)` and the input/output encoded through constraints `(c4,c5)`, indicating that the code indeed matches the specification. Contrastingly, if the specified output was instead "mail.com" (the programmer meant to identify the e-mail domain instead of the alias), the SMT solver would return *unsat* indicating that the code does not match the specification.

The previous example illustrates the essence and novelty of the approach, but it does not address some critical issues such as handling richer queries and repositories, refining the set of potential matches, and instantiating the approach in other domains. In this work we begin to explore those issues. More specifically, in terms of applicability to other domains, we instantiate and assess the approach in three domains: the Java string library, the Yahoo! Pipes mashups, and SQL select statements (Section V). These domains were selected in part to illustrate the generality of the approach by utilizing three diverse forms of input/output specification (Section II), and in part because of their relative simple and common underlying semantics and the availability

Department of Computer Science & Engineering, University of Nebraska-Lincoln, Technical Report, TR-UNL-CSE-2012-0012

of repositories that could be searched in the evaluation (Section VI). In terms of result refinement, we describe how the approach supports incremental weakening and strengthening of the specifications (queries), and weakening of the encodings to either enrich the results set with approximate matches or prune the result set of coincidental matches (Section IV). Last, we show how our approach performs when searching repositories, how it is impacted by queries of various sizes and complexities, and how it can be used to complement existing syntactic search engines (Section VI). The contributions of this work are:

- Definition of an approach that uses an SMT solver to identify matches given lightweight specifications and programs encoded as constraints
- Implementation of the approach in three domains: Java String library, Yahoo! Pipes, and SQL.
- Assessment of the generality, effectiveness, and efficiency of the approach in various settings.

Specifically, we explore 1) how our search results compare to syntactic search results and illustrate how our technique can improve the results from a syntactic search engine, 2) how changes in solving time and abstraction of the program encodings affect potential matches, and 3) how changes in the size and complexity of the search queries impacts the search performance. Our previous work in this area presented a brief and preliminary instantiation of our approach on the Yahoo! Pipes mashup language [22]. This work defines the approach and its implementation in more detail, provides a more comprehensive assessment by adding additional independent variables, and instantiates and assesses the approach on two additional domains (Java string library and SQL select statements).

## II. Approach Motivation and Illustration

In this section we briefly describe how search is performed in three domains, how our approach could be instantiated in those domains, and when it would be valuable to do so.

**String Manipulations in Java.** The alias extraction example in Section I illustrates when a syntactic search may return many irrelevant matches and how our approach would operate, and hints at scenarios when it could be beneficial. Depending on the setting, these benefits can be obtained by using our approach on its own or by complementing a syntactic search.

Our approach requires a query consisting not of keywords, but of input and expected output pairs. In this domain, those inputs take the form of strings and outputs could be one of several datatypes; integers, characters, strings, and booleans are supported by our implementation (Section V). This query format is not uncommon among programmers, especially when explaining a problem to a peer. Among the 67 string manipulation questions we evaluated from stackoverflow (Section VI), we observed that 40 (60%) already provide concrete input/output examples to describe their problem.

Yet, finding relevant code depends on more than an input/output query. It also needs a large and diverse repository to capture a range of program behaviors and an efficient search engine to find code quickly. Our implementation and assessment of the approach for Java aims to meet these objectives (Section V). To illustrate, with the alias extraction example, our search returned 51 matches in less than one second from a repository with hundreds of encoded programs. Some of the results were coincidental matches. For example, `string scheme = uri.substring(0, 5);`, only matches because the alias “susie” has five characters, yet many others match the intention of the problem: `username = to.substring(0, to.indexOf('@'));`.

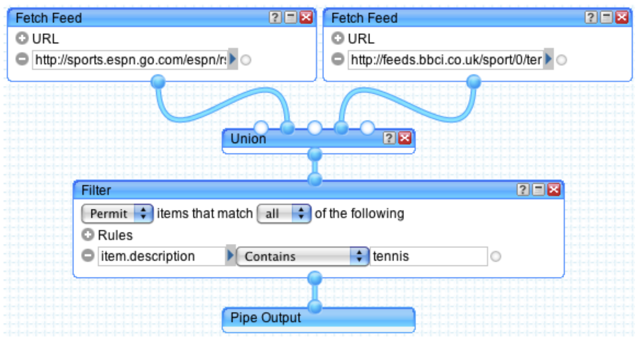
On their own, neither of these snippets forms a complete program. If we were to execute this code (as would be done in semantic search engines that utilize test cases [15], [19], [20]), each of the variables in the expression statement on the right-hand side would need to be instantiated. By encoding the behavior of the snippets as constraints, the uninitialized variables remain uninitialized, and we make no assumptions about the values they hold. These are called *symbolic* variables. In the above snippets, the variables `scheme`, `uri`, `username`, and `to` are *symbolic* as those do not have assigned values. The other variables have assigned values (i.e., `0`, `5`, and `'@'`), so those are *concrete* in the encoding. Part of the refinement process in our approach allows for concrete variables to be relaxed, which removes their concrete values making them symbolic, hence increasing the space of specifications that the snippet satisfies.

Nearly as important as finding relevant code is the ability to quickly discard irrelevant code. When applied on the results returned by a syntactic search, our approach was able to discard over a third of matches as they were irrelevant, effectively reducing the space that the programmer needs to sift through to find useful results (Section VI).

**Yahoo! Pipes Mashups.** Yahoo! Pipes is a mashup language with over 90,000 users [11], and a public repository of over 100,000 artifacts [18]. These programs combine, filter, sort, annotate, and manipulate RSS feeds. To write them, programmers use the Pipes Editor, dragging and dropping predefined modules and connecting them with wires to define the data and control flow. Figure 1 shows a sample pipe mashup to join and filter the content of two RSS feeds based on the word “tennis”.

Currently, programmers can syntactically search the repository by URLs accessed, tags, or keyword. To illustrate the challenges programmers face with these search mechanisms, we performed five searches (see Section VI) by URL. The number of matches can be in the thousands which is not surprising as many mashups include common URLs. The average number of relevant matches among the top ten results is 0.9. Using other built-in search capabilities does not fare much better. Searching by components retrieves even more results and would require for the programmer to know how the pipe would be built. The effectiveness of searching with tags was highly dependent on the community’s ability to systematically categorize their artifacts.

In our approach instantiation targeting pipes, the programmer provides the URLs for RSS feed(s) as input, and the list of expected records as output. The encoding process is briefly illustrated with the sample pipe and its approximate encoding in Figure 1. Constraints $c1$, $c2$, and $c11$ ensure that the right data is set as input and output. Constraints $c3$, $c4$, $c6$ and $c10$



http://sports.espn.go.com/espn/r

http://feeds.bbci.co.uk/sport/0/ter

Permit items that match all of the following

item.description Contains tennis

| Module | Encoding as Constraints |
|---|---|
| Fetch$_1$ | $c1 : Fetch1_o$ = URL$_1$ |
| Fetch$_2$ | $c2 : Fetch2_o$ = URL$_2$ |
| link$_1$ | $c3 : Fetch1_o = Union_{i1}$ |
| link$_2$ | $c4 : Fetch2_o = Union_{i2}$ |
| Union | $c5 : Union_o = union(Union_{i1}, Union_{i2})$ |
| link$_3$ | $c6 : Union_o = Filter_i$ |
| Filter | $c7 : hasRec(Filter_i, r) \wedge contains(field(r, \text{“descr”}), \text{“tennis”}) \rightarrow hasRec(Filter_o, r)$<br>$c8 : hasRec(Filter_o, r) \rightarrow hasRec(Filter_i, r)$<br>$c9 : \ldots$ |
| link$_4$ | $c10 : Filter_o = Output_i$ |
| Output | $c11 : Pipe_o = Output_o$ |

Fig. 1. Mapping Pipes into Constraints

represent links that ensure proper routing of the items from the source module to the destination module. Constraint $c5$ ensures that the output of the Union module is the concatenation of the two inputs. Constraint $c7$ ensures that if a record $r$ is in the input to the filter module and it contains the substring “tennis” in the description field ($field(r, \text{“descr”})$), then $r$ is in the output from the filter module. Constraint $c8$ ensures that if a record $r$ is included in the output of the $filter$ module, then $r$ in the input.

For some search queries, this level of encoding might be too strict. Say a programmer wants to filter based on “volleyball” rather than “tennis”. The filter module in Figure 1 encoded concretely would result in the constraint $contains(field(r, \text{“descr”}), \text{“tennis”}) = true$, which would not satisfy the constraints associated with “volleyball”. However, with a weaker encoding consisting of constraint $contains(field(r, \text{“descr”}), s) = true$ for some string $s$, an SMT solver could determine that for $s =$ “volleyball”, this program is a match. We implement and evaluate two abstraction levels within the pipes domain (Section VI).

**SQL Select Statements.** SQL select statements have been used for decades to support data retrieval, operating on their own or being embedded into other programming languages. Given the simplicity of the SQL syntax and its popularity, even well conceived syntactic queries will return many irrelevant results. Still, we observe that SQL programmers have many questions and they often turn to peer communities like stackoverflow, which has over 91,000 questions related to SQL, to find an answer. We aim to support those programmers in answering questions such as: “I have table with records ’user’ and ’balance’. How to show 10 usernames with highest balance? ... How to show but only when they have more than 1.000.000$?”[1] Here, the programmer knows the desired behavior and can describe it through a concrete example table:

```
 id  | user       |    balance
-----+------------+------------
 145 | rekin76    |   469370.44
 705 | shantee    |   149160.09
5725 | terro      |    93004.45
 ... | ...        |      ...
```

When instantiating our approach for SQL, the input and output take the form of database table(s). The indexed SQL select statements are encoded as constraints, similarly to what was done for strings and pipes (in fact the semantics for the pipe’s `filter` module, for example, are almost identical to that of the `where` clause in SQL). Given tables as input and output, the SMT solver determines, for each encoded SQL program, if it will achieve the output table from the input table(s). For the previous example, our approach would match the recommendation of three positively voted responses in stackoverflow:

```
SELECT user, balance FROM table WHERE balance
>= 1000000 ORDER BY balance DESC LIMIT 10
```

In SQL, the input/output tables can be large, since they may come from live databases, so it becomes important to understand the impact of large specifications on solver time. As we explore in Section VI, it is not just the size of tables that matters but also the complexity of behavior exhibited in the specification.

At this point, we have illustrated several interesting aspects of the approach in three domains, more specifically the forms of inputs and outputs, how the encoding is performed, and the potential to find relevant matches. In the next sections we will proceed to formalize those aspects, describe the implementation details for each domain, and assess their performance.

## III. Related Work

Our approach is related to recent work in code search, code reuse, automatic program generation and program synthesis, and verification that leverages SMT solvers.

**Code Search.** Recent studies have revealed that programmers typically use general search engines to find code for reuse [21]. More specialized code search engines (e.g., Koders, Krugle, ohloh) incorporate various filtering capabilities (e.g., language, domain, scores) and program syntax into the query to better guide the matching process [21]. Other approaches add natural language processing to increase the potential matches [8], [16]. Our work is different in that we perform a more semantic search, but as we show (Section VI), both approaches are complementary and can be easily combined.

Early work in semantic search required developers to write complex specifications of the desired behavior using first-order logic or specialized languages (e.g., [7], [17], [26]), which can be expensive to develop and error-prone. The cost of writing specifications can be reduced by using incomplete behavioral specifications, such as those provided by test cases (a form of input/output) [15], [19], [20], but these approaches require that the code be executed to find matches. Further, executing test

[1]http://stackoverflow.com/questions/11599636

cases against the code will only return exact matches, missing many relevant matches that may simply have a slightly different signature (e.g., extra parameter).

**Code Reuse.** In the code reuse process, there are two primary activities: finding and integrating. Our approach focuses on the first part, finding, but has potential to be useful with integration. Some recent work assists programmers with integrating new code by matching it with the their development context based on structural properties (e.g., method signature, return types) [4], [10]. These approaches guarantee structural matching, but the behavior of the integrated code may not be well understood, and is something our approach could guarantee given a specification.

**Automated Program Generation and Program Synthesis.** Previous work in the area of automated program generation [1] relates to our work in that the high level specifications are used as the basis to derive programs. Closer to our work is that in the area of program synthesis, more specifically, that which makes use of solvers to derive a function that maps an input to an output (e.g., [9]). The key difference is that our approach uses the solver to find a match against real programs that have been encoded, while these synthesis efforts have to define a domain specific grammar that can be traversed exhaustively to generate a program that matches the programmers' constraints.

**Verification.** Constraint solvers and SMT solvers have been used extensively for test case generation. Toward the goal of database generation for testing, reverse query processing takes a query and a result table as inputs and using a constraint solver, produces a database instance that could have produced the result [2]. Other work in test case generation for SQL queries has used SMT solvers to generate tables based on queries [24]. In our work, we do not generate database tables, but rather determine if a given query could have produced a specified result set (output) from specified input table(s).

## IV. Approach

Our general approach is illustrated in Figure 2. The gray boxes indicate the key components and technical challenges: defining *lightweight specifications* as input/output, *encoding* programs and specifications as constraints, and refining program encodings and specifications when *too few* or *too many* matches are found. The crawling and program encoding processes happen *offline*, whereas the search for relevant code happens *online*.

### A. Specifying Behavior

Instead of textual queries to find syntactic matches, our approach takes lightweight specifications that characterize the desired behavior of the code (*Lightweight Specifications* in Figure 2). These specifications, $LS$, consist of concrete input/output pairs that exemplify part of the desired system behavior, like "susie@mail.com" and "susie" from Section I. To more completely specify the desired behavior, multiple input/output pairs can be defined: $LS = \{(i_1, o_1), \ldots, (i_k, o_k)\}$, for $k$ pairs. The inputs and outputs take different forms depending on the domain. In the Java string library, the input is a string and the output could be a string, integer, boolean, or character. In Yahoo! Pipes, the inputs are URLs (from which the RSS feeds are retrieved) and the output is a list of items. For SQL select statements, the input is one or more tables and the output is a table.

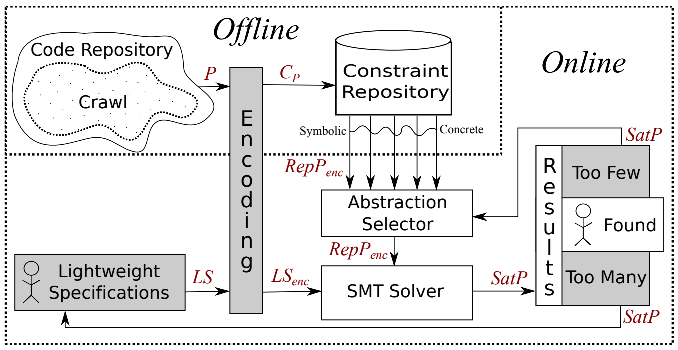


Fig. 2. General Approach

The size of $k$ defines, in part, the strength of the specifications and hence the number of potential matches. A programmer can provide specifications incrementally, starting with a small number of pairs and adding more to further constrain the search.

The last step of this process is the automated encoding of $LS$ into constraints, $LS_{enc}$, for the solver to consume when the search starts (recall $c4$ and $c5$ from Section I).

### B. Encoding

In our approach, encoding and solving are analogous to crawling and indexing performed by search engines [14]. Offline, a repository (*Code Repository* in Figure 2) is crawled to collect programs. These programs are encoded as constraints (*Encoding*, analogous to indexing), and stored in a *Constraint Repository*.

More formally, given a collected set of programs, the encoding process $Encodes : P_i \rightarrow C_{P_i}$, where $C_{P_i} = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ is a conjunction of $m$ constraints that describe a program $P_i$.[2] To illustrate using the alias extraction example in Section I, $P_i$ = `output=input.substring(0, input.indexOf('@'))`, and is encoded using $m = 3$ constraints, so $C_P = c_1 \wedge c_2 \wedge c_3$. In the end, the encoding process maps every program to a set of constraints such that $RepP_{enc} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$.

In the initial encoding, the variables are encoded as concretely as possible. If a variable is instantiated, the encoding assigns a concrete value. If it is not instantiated, then the encoding assigns a symbolic value. The one exception to this is the input and output; for a program to be solvable for any arbitrary $LS$, the input and output of the program are encoded symbolically.

Weaker encodings can approximate program behavior and be useful when searching over small repositories or in the presence of strong specifications. If the program snippet consists of `output=input.substring(0, varA)`, then `varA` is encoded as symbolic and `0` concretely. A weaker encoding simply defines some concrete variables to be symbolic. For the previous example, we could encode the first parameter as symbolic, `output=input.substring(varB, varA)`, to obtain a more general version of that program snippet.

[2]This constraint language is sufficient to encode the domains we consider. It is likely, however, that generalizing the approach further may require a more flexible constraint system involving disjunctions to accommodate, for example, branching. This is something that we leave as future work.

TABLE I
BASIC OPERATIONS FOR CURRENT IMPLEMENTATION

| Term | Description | Java Strings | Yahoo! Pipes | SQL Select |
|---|---|---|---|---|
| Accessor | returns an object at a location, or identifies an object location. does not modify any object. | *charAt*: S x I $\mapsto$ C | | getCol: T x S $\mapsto$ Col |
| | | *indexOf*: S x S x I $\mapsto$ I<br>*lastindexOf*: S x S x I $\mapsto$ I | field: R x S $\mapsto$ I $\vert$ S<br>recordOf: L x I $\mapsto$ R | |
| Join | conjoins two objects | *concat*: S x S $\mapsto$ S | *union*: L x L $\mapsto$ L | join: T x T $\mapsto$ T |
| Filtering | modifies object based on criteria | *substring*: S x I x I $\mapsto$ S | *truncate*: L x I $\mapsto$ L<br>*tail*: L x I $\mapsto$ L<br>*filter*: L x S x Op $\mapsto$ L | *limit*: T x I $\mapsto$ T<br>*where*: T x Col x Op $\mapsto$ T<br>*distinct*: T x Col $\mapsto$ T |
| Copy | duplicates an object | | *split*: L $\mapsto$ L x L | |
| Permute | re-orders a list of objects | | *sort*: L $\mapsto$ L | *order by*: T $\mapsto$ T |
| Size | returns the size of an object | *length*: S $\mapsto$ I | | height: T $\mapsto$ I |
| | | | size: L $\mapsto$ I | |
| Operators (Op) | compare two objects | $<, \leq, >, \geq$: I x I $\mapsto$ B | equals: I x I $\mapsto$ B | equals: B x B $\mapsto$ B |
| | | contains: S x S $\mapsto$ B<br>*equals*: S x S $\mapsto$ B | equals: C x C $\mapsto$ B | equals: T x T $\mapsto$ B |
| | | *startsWith*: S x S $\mapsto$ B<br>*endsWith*: S x S $\mapsto$ B | equals: L x L $\mapsto$ B<br>hasRec: L x R $\mapsto$ B | |

C = Character, I = Integer, B = Boolean, S = String, R = Record (map with names as strings), L = List, T = Table, Col = Column (in Table)
Functions in *italics* indicate actual names of language constructs (and are located in the appropriate column).

Critical to the efficiency of the approach is the granularity of the encoding. The finest granularity corresponds to encoding the whole program behavior in $C_{P_i}$. At the coarsest granularity the encoding would capture none of the program behavior so $C_{P_i} = true$. These extremes correspond to the least and the greatest number of matches and the worst and the best search speeds respectively, but there is a spectrum of choices in between. We explore several in Section V, such as encoding at the component level (Yahoo! Pipes), query level (SQL), or library level (Java).

### *C. Solving*

The constraint repository, $RepP_{enc}$, is used by the solver, in conjunction with the encoded specifications $LS_{enc}$, to determine matches (*SMT Solver* in Figure 2). Given $LS_{enc}$, for each $C_P \in RepP_{enc}$, the approach invokes $Solve(C_P \wedge LS_{enc}) = (sat, unsat, unknown)$. $Solve$ returns *sat* when a satisfiable model is found or *unsat* when no model is possible. When the solver is stopped before it reaches a conclusion or it cannot handle a set of constraints, *unknown* might be returned. The set of matches, or *SatP*, consists of all programs that return *sat*.

### *D. Refinement*

If the specifications or the encoded program constraints are too weak, many matches may be returned (*too many* in Figure 2); if they are too strong, the solver may not yield any results (*too few*). Refinement is a process that helps to address these situations by tuning the lightweight specifications ($LS'$) and by using different program encodings (*Abstraction Selector*) to find solutions that are *close enough* when no exact solutions exist.

**Tuning Lightweight Specifications.** When $SatP$ has too many coincidental matches, a programmer may strengthen the specifications by providing additional $(i, o)$ that further demonstrate the desired behavior, similar to query reformulation [6]. The programmer can also replace an input/output pair with one that captures a more distinguishable aspect of the desired behavior.

Conversely, a programmer may weaken the specifications when a match is not found or when the search takes too long to provide a response. An example of this last case occurs when the tables provided as input for SQL have hundreds of rows causing the solving time to take minutes; in this case it may useful if possible to select the subset of the table that still captures the key desired behavior. We explore the impact of input size on search efficiency for SQL in Section VI.

**Changing Program Encodings.** When $SatP$ has too few results, refinement can direct the solver to use a more relaxed encoding through the *Abstraction Selector*. We exploit the fact that most languages contain constraints over multiple data types (e.g., strings, characters, integers, booleans) for which the variable values can be relaxed and encoded as symbolic. $Weakening : C_{P_i} \rightarrow {C_{P_i}}'$ means that $(Solve(C_{P_i} \wedge LS_{enc}) = unsat) \wedge (Solve({C_{P_i}}' \wedge LS_{enc}) = sat)$ for some relaxation of $C_{P_i}$ that yields $C'_{P_i}$. Encoding weakening is performed by systematically making the constraints on a particular datatype symbolic, similar to the pre/postcondition lattices in previous work on specification matching [17], [26].

## V. IMPLEMENTATION

We now highlight some aspects of instantiating the approach on three languages and variations across implementations.

To make the specifications usable by an SMT solver, our implementation takes the programmer's input/output in textual form and performs a transformation into constraints as illustrated in Section I. Since the specifications are given concretely, the encodings retain the information provided in the specifications (i.e., no variables are symbolic).

For each program in the repository, we first identify its input and output, which will be encoded symbolically. For Java assignment statements, the LHS constitutes the program output and the receiving object on the expression of the RHS is the input (assuming there is just one input). For pipes, specific modules are associated with inputs (e.g., Fetch module) and with producing the output (Output module). For SQL, the program inputs are the tables and fields that are referenced by the query, and the output is a table.

Our encoding implementations take as input a repository of programs, and can encode assignment or return statements of

Java that contain string manipulation functions (i.e., `charAt`, `concat`, `contains`, `endsWith`, `equals`, `indexOf`, `lastIndexOf`, `length`, `startsWith`, and `substring`), Yahoo! Pipes mashup programs containing the `fetch`, `filter`, `output`, `sort`, `split`, `tail`, `truncate`, and `union` modules, and SQL select statements with `count`, `distinct`, `limit`, `order by`, and `where` clauses.

Mapping from programs to constraints in the three target domains is performed similarly, so instead of describing each encoding implementation, we provide an abstracted version of the key data types and operations supported, and how the particular domain constructs map to each. This overview is presented in Table I. We support three primitive types (characters (C), integers (I), booleans (B)) and one composite type (list (L)). These basic types are sufficient to represent all the constructs we support across the three domains. For example, a string (S) is a shorthand given as a list of characters, a Yahoo! Pipes record (R) is a map of strings to objects with names modeled as strings, SQL tables (T) are lists of lists, and a column (Col) is a named list where the name is modeled as a string. Four of the datatypes that can hold concrete or symbolic values: integers, characters, booleans, and strings. The Java implementation uses all the data types. Yahoo! Pipes uses strings and integers, and our SQL implementation supports only integers.

Using these basic data types, there are seven basic operations to capture the core semantics of the programs we analyze. These operations are listed in the *Term* column of Table I, followed by a description and the operation mapping to the language subsets supported by our implementation. For example, filtering is supported in all three languages, by the `substring` function in Java (returning only a subset of a string), the `filter` module in Yahoo! Pipes, and the `where` clause in SQL select statements. The `charAt` accessor function is part of the Java language, but also used by Yahoo! Pipes. Each encoded program consists of a composition of the basic operations. For example, a SQL query `SELECT name, salary FROM employee, payroll WHERE employee.id = payroll.id ORDER BY salary` contains an implicit join of two tables, `employee` and `payroll`, achieved using: `output = permute(filtering(join(employee, payroll)))`.

To invoke the SMT solver for a given specification and encoded program, some additional information is needed, which we call *search parameters*. The first parameter is the abstraction level of the encoded programs. We begin with the strictest (most concrete) level, but it may be relaxed as the search process iterates in the presence of tight or complex constraints. The second parameter is the solver time, which defines how long the solver is allowed to run on a particular constraint system. In some cases, as shown in Section VI, it can take several minutes for the solver to return *sat* or *unsat*, so setting a maximum solver time can lead to an efficient search, though it can miss some matches.

As part of our implementation efforts, we have also prototyped several enhancements on individual domains that may be worth generalizing. In the context of encoding Yahoo! Pipes we observed that many had clones with slightly different syntax that were not worth encoding. So, to reduce the encoding effort (and consequently the search time), we refactor all pipes to obtain a more uniform representation, remove the duplicates, and then proceed with the encoding. Similarly, encoding the language fragments requires evaluating substring and equality relations over strings, and enumeration over all elements in a list. To efficiently support these operations, we consider bounded strings and list, where the bounds are configurable (in line with recent work on string constraints [3], [12]). In Java, the results are returned to the programmer ordered according to the density of concrete variables in the program, as these are more likely to fit the programmer's query *as is* and without modification (such as instantiating symbolic variables with values from the satisfiable model). In SQL, ANTLR is used to validate the SQL queries. Solving is performed by Z3 v.4.1 [25] for all languages.

## VI. Evaluation

The study is designed to provide a preliminary assessment of the approach effectiveness across multiple dimensions while highlighting some the key aspects through the three supported domains: Java, Yahoo! Pipes, and SQL.

To show how our search approach compares to traditional keyword searches, we focus on Java and compare the effectiveness of finding relevant code using our approach versus using Google. Our search approach is designed to be flexible and allow for *close* matches to be found when exact ones do no exist; on the Yahoo! Pipes domain, we show how tweaking the search parameters can affect the effectiveness of the search. To better understand the impact of specification complexity on search time, we manipulate the size and content of the queries in SQL, measuring the impact on search time.

Next, we state the specific research questions, describing for each how the repositories were built, the queries selected, the metrics used, and the results. All of the study artifacts are available online.[3] For all studies, our data were collected under Linux on 2.4GHz Opteron 250s with 16GB of RAM.

### *RQ1: How Does Our Search Compare to Syntactic Searches?*

RQ1 aims to compare an existing and popular search tool, Google, against our approach. First, we perform this comparison on a local repository that we created and control. Second, we perform a Google search on the web and explore how the results could be filtered by also using our search approach.

**Artifacts.** We built a local repository by issuing searches on Koders.com [13] for each of the Java string library functions supported by our encoding. We then scraped all lines of Java source code that contained a call to at least one of the supported functions, totaling 5192 lines. We pruned out duplicates, lines that contained functions we do not support, and those that are not assignment or return statements. This left 713 unique snippets of code that form the Java code repository used in this evaluation.

Comparing our search technique to Google requires two different query models, keyword for Google and input/output for our approach. So that the queries are representative of what programmers use, we derived them from questions asked on

[3]http://cse.unl.edu/~kstolee/semsearch/

TABLE II
JAVA ARTIFACTS

| Q | Title | Input | Output | Our Approach # | Our Approach P@10 | Google Local # | Google Local P@10 |
|---|---|---|---|---|---|---|---|
| 1 | Just copy a substring in java | "Animal.dog" "World.game" | "Animal" "World" | 4 | 4 | 99 | 1 |
| 2 | extract string including whitespaces within string (java) | "23 14 this is random" | "this is random" | 24 | 10 | 34 | 3 |
| 3 | How to get a 1.2 formatted string from String? | "1.500000154" | "1.5" | 48 | 10 | 37 | 2 |
| 4 | How to pull out sub-string from string (Android)? | "&lt;TD&gt;TextText&lt;/TD&gt;" | "TextText" | 13 | *10 | 100 | *2 |
| 5 | Trim last 4 characters of Object | "Breakfast($10)" | "Breakfast" | 48 | 10 | 5 | 0 |
| 6 | Removing a substring between two characters (java) | "I &lt;str&gt;really&lt;/str&gt; want ..." | "I really want ..." | 0 | 0 | 99 | 0 |
| 7 | Splitting up a string in Java | "i i i block_of_text" | "block_of_text" | 21 | 10 | 42 | 3 |
| 8 | How to find substring of a string with whitespaces in Java? | "c not in(5,6)" | true | 20 | 10 | 99 | 0 |
| 9 | Limiting the number of characters in a string, and chopping off the rest | "124891," "difference," "22.348," "montreal" | "1248" "diff" "22.3" "mont" | 49 | *10 | 41 | *3 |
| 10 | Trim String in Java while preserve full word | "The quick brown fox jumps" | "The quick brown..." | 0 | 0 | 40 | 0 |
| 11 | How to return everything after x characters in a string | "This is a looong string" | "is a looong string" | 23 | *10 | 70 | 3 |
| 12 | Slice a string in groovy | "nnYYYYYYnnnnnnn" | "YYYYYY" | 13 | *10 | 38 | 2 |
| 13 | How to replace case-insensitive literal substrings in Java | "FooBar" "fooBar" | "Bar" "Bar" | 24 | 10 | 2 | 0 |
| 14 | Removing first character of a string | "Jamaica" | "amaica" | 22 | *10 | 38 | 3 |
| 15 | How to find nth occurrence of character in a string? | "/folder1/folder2/folder3/" | "folder3" | 13 | 10 | 42 | 2 |
| 16 | Java finding substring | "**tok=zHVVMHy..." | "zHVVMHy" | 14 | 10 | 2 | 0 |
| 17 | Finding a string within a string | "...MN=5,DTM=DIS..." | "DTM=DISABLED" | 13 | 10 | 34 | 2 |
|  |  |  | **Average** | 20.5 | 8.5 | 48.4 | 1.5 |

**#**: The number of results from the search
**P@10**: The relevant results from the search (***** indicates some results match Stackoverflow responses)

stackoverflow.com, where the posting title was used as the keyword query and the input/output example was encoded for our semantic search. Of the 67 questions tagged with `java`, `string`, and `substring`, 40 (60%) contain some form of explicit example. For 17 of those cases, our current Java implementation supports encoding the example. The remaining 23 involve constructs not currently supported by our implementation, such as regular expressions or arrays. The titles and input/output for the 17 questions are shown in Table II.

**Metrics.** To compare the results across the search techniques on the local repository, we use the number of results and *P@10*. Since the search results on the web may return pages with multiple snippets of code, we define a new metric, *S@10*, which represents the number of code snippets returned in the top ten results, and its complement *S'@10*. To capture *S@10*, we issue Google queries, then scrape and count the one-line Java code snippets from the top 10 page results. Next, we run our search technique using the scraped snippets as a repository to discard snippets that are irrelevant.

**Results: Local Repository.** For each of the 17 stackoverflow questions, we encoded the input/output as $LS_{enc}$ and searched our local repository for matches. On average, 20.5 matches were found for each query, ranging from zero (in two cases, questions 6 and 10) to 49 results.[4] As an example, for the second question in Table II, given the input "23 14 this is random" and output "this is random", our search approach finds 24 matches including: `string message = name.substring(6);`. The results from the search were ordered by the number of symbolic variables in the snippet, in decreasing order (with the idea that the concrete results are more directly transferrable to the programmer's context). Regardless of the ordering, by the design of our semantics search, all the results are relevant as they match the input/output specifications, and so the *P@10* metric is ten or the number of returned results, whichever is smaller (for the first question, there are only four matches, so *P@10* = 4).

Using the Google search engine, on average, 48.5 matches were found for each query, ranging from two to 100. For the second query, for example, we see that Google returns 34 results. Checking the top ten results against the input/output specifications reveals that only three of the top ten results were relevant. For example, `string = string.substring(0, end);` is irrelevant because it grabs the first part of a string, rather than the last part as illustrated in the example. On average, 1.5 of the results are relevant, with a range from zero to three.

Overall, Google returns over twice as many results as our approach, but among the top ten, our approach is over five times more effective. For four of the 17 queries (5, 8, 13, 16), our approach provides matches when Google does not find any as the syntactic query was not rich enough to identify relevant results.

If we compare the results to the solutions proposed and positively voted by the stackoverflow community, five of the queries using our local search also match the proposed solutions from the community (marked with the *). A match was determined if all API calls were the same between two snippets.[5] When using

[4]For those questions that have multiple input/output, we ensure that the models returned can satisfy all input/output pairs.

[5]The stackoverflow community often proposed solutions that used regular expressions, string tokenizers, and arrays, which are not currently supported by our encoding and thus do not appear in any of the result sets.

TABLE III
GOOGLE + OUR APPROACH (RESULTS REFLECT THOSE RETURNED IN ENGLISH DURING THE WEEK OF AUGUST 6, 2012. )

| Question | S@10 | Discarded | S'@10 | % Reduction |
|---|---|---|---|---|
| 1 | 25 | 18 | 7 | 72% |
| 2 | 17 | 0 | 17 | 0% |
| 3 | 0 | 0 | 0 | – |
| 4 | 36 | 12 | *24 | 33% |
| 5 | 3 | 0 | 3 | 0% |
| 6 | 37 | 6 | 31 | 16% |
| 7 | 16 | 4 | 12 | 25% |
| 8 | 38 | 7 | 31 | 18% |
| 9 | 0 | 0 | 0 | – |
| 10 | 9 | 2 | 7 | 22% |
| 11 | 6 | 3 | 3 | 50% |
| 12 | 7 | 2 | *5 | 29% |
| 13 | 29 | 11 | 17 | 38% |
| 14 | 0 | 0 | 0 | – |
| 15 | 0 | 0 | 0 | – |
| 16 | 26 | 14 | 12 | 54% |
| 17 | 8 | 7 | 1 | 88% |
| **Avg** | **15** | **5** | **10** | **34%** |

| | |
|---|---|
| **S@10** | One-line Java snippets from top 10 Google pages |
| **Discarded** | Snippets from S@10 that we support and are *unsat* |
| **S'@10** | The reduced pool of snippets to evaluate (* indicates that a *sat* snippet was found) |
| **Reduction** | The reduction in snippets that need to be evaluated |

the same evaluation criteria for Google we find only two queries that match the stackoverflow solutions.

In terms of performance, encoding all 713 snippets takes 2.991 seconds (averaged over ten runs), which is approximately 4ms per snippet. Among all the input/output examples in Table II and all searches, the average solver time to determine *sat* is 0.0483 seconds and to determine *unsat* is 0.0051 seconds.

**Results: On the Web.** Using the titles from the stackoverflow questions shown in Table II, we invoke Google to search for relevant code; we report the *S@10* metric in Table III. On average, 15 snippets were gathered per question, ranging from zero to 38 (zero occurred when none of the retrieved pages were in the Java language). By encoding each of these snippets, we are able to check the input/output pairs from Table II against the retrieved snippets. The number of snippets for which the SMT solver returns *unsat* is shown in the *Discarded* column. This number captures the space of results that the programmer would not have to examine if using our approach on the Google results. The programmer must then only look at *S'@10* snippets. Overall, the number of snippets returned could be reduced by 34% just by using our semantic search on top of the Google results. In two cases, for questions 4 and 12 (marked with *), at least one snippet returned *sat*, indicating that it is indeed a match. Since we do not support the entire Java language, matches were not as common; for those snippets that we do support, most could be quickly discarded. Clearly this motivates the need for more complete coverage of the Java language in our implementation, which we leave for future work.

### *RQ2: What is the Impact of Tweaking Search Parameters*

RQ2 aims to explore the impact of solver time and the abstraction level of program encodings on the precision and recall of the search. In Yahoo! Pipes, the specifications can be quite large and complex. Each input and output is a lists of records, and each record is a map with several long strings. As a result, returning *sat* can take up to several minutes. This allows us to explore how changes in solver time and abstraction impact the effectiveness of the search.

**Artifacts.** In a previous study with Yahoo! Pipes [23] we scraped 32,887 pipes programs from the public repository by issuing 50 queries against the repository and removing all duplicates. To perform the searches for the study, we gathered specifications from five pipes in the repository. The pipes were clustered based on their structural similarity, and one pipe was selected from each of the median five clusters. *Pipe-1* has six modules and two URLs. It filters the records by "10-Day" or "Current" in the title field, for the purpose of returning weather information. *Pipe-2* has five modules and two URLs. It retains records that contain "hotel" in the description field, then sorts the records according to publication date and retains the first three records. *Pipe-3* has nine modules and three URLs. It grabs the first three records from each URL and sorts them according to publication date. *Pipe-4* has four modules and one URL. It performs head and tail operations to return the third record in the input list. *Pipe-5* contains six modules and has two URLs. It aggregates and sorts the items from the input lists, where one list is filtered by the presence of "au" in the description field. The specifications for these pipes were generated by executing each and capturing their inputs and outputs, then transforming their input/output lists into constraints.

**Metrics.** We manipulate two search parameters, the abstraction of the program encodings and the maximum solver time. We report the number of pipes in the repository that return *sat* (S), *unsat* (U), and *unknown* (?) at each of four solver times, 1, 10, 100, and 1000 seconds, considering two levels of abstraction on the program encoding, all concrete and all symbolic, on the string and integer fields. We also calculate precision and recall, where $precision = \frac{relevant \cap sat}{sat}$ and $recall = \frac{relevant \cap sat}{relevant}$. The relevant results are defined as those that will eventually (given infinite time) return *sat* with a symbolic encoding, which represents the pipes for which some instantiation of the module field values can achieve the desired behavior.

**Results.** We search the repository using the five pipe specifications, given the solver times and abstraction levels described. The precision of the search will always be 1.00, as we protect against spurious results by design. The results of our experiments are shown in Table IV.

We observe that using symbolic constraints usually yields more results than concrete. For instance, with *Pipe-specification- 4* in Table IV(d) at 1000 seconds, all the programs have been determined to be *sat* or *unsat* for the concrete and symbolic encodings (*?* = 0 for both). Yet, the symbolic encoding yields 89 possible matches while the concrete encoding only finds one. Here, there is only one pipe in the repository that will return *sat* with the concrete encoding, *Pipe-4*. With the symbolic encoding, the title string in the *filter* modules are symbolic so pipes that return all records with a particular title would be returned as *sat* by the solver.

TABLE IV
PRECISION AND RECALL FOR YAHOO! PIPES STUDY

(a) Pipe-specification-1

| | Concrete | | | | Symbolic | | | |
|---|---|---|---|---|---|---|---|---|
| **sec.** | **S** | **U** | **?** | **Recall** | **S** | **U** | **?** | **Recall** |
| 1000 | 17 | 2842 | 0 | 0.1650 | 100 | 2756 | 3 | 0.9709 |
| 100 | 16 | 2842 | 1 | 0.1553 | 24 | 2756 | 79 | 0.2330 |
| 10 | 0 | 2836 | 23 | 0.0000 | 0 | 2723 | 136 | 0.0000 |
| 1 | 0 | 2794 | 65 | 0.0000 | 0 | 2572 | 287 | 0.0000 |

(b) Pipe-specification-2

| | Concrete | | | | Symbolic | | | |
|---|---|---|---|---|---|---|---|---|
| **sec.** | **S** | **U** | **?** | **Recall** | **S** | **U** | **?** | **Recall** |
| 1000 | 1 | 2858 | 0 | 0.3333 | 2 | 2856 | 1 | 0.6667 |
| 100 | 0 | 2858 | 1 | 0.0000 | 0 | 2853 | 6 | 0.0000 |
| 10 | 0 | 2836 | 23 | 0.0000 | 0 | 2785 | 74 | 0.0000 |
| 1 | 0 | 2783 | 76 | 0.0000 | 0 | 2567 | 292 | 0.0000 |

(c) Pipe-specification-3

| | Concrete | | | | Symbolic | | | |
|---|---|---|---|---|---|---|---|---|
| **sec.** | **S** | **U** | **?** | **Recall** | **S** | **U** | **?** | **Recall** |
| 1000 | 3 | 2856 | 0 | 0.1429 | 18 | 2838 | 3 | 0.8571 |
| 100 | 0 | 2856 | 3 | 0.0000 | 0 | 2833 | 26 | 0.0000 |
| 10 | 0 | 2835 | 24 | 0.0000 | 0 | 2651 | 208 | 0.0000 |
| 1 | 0 | 2798 | 61 | 0.0000 | 0 | 2554 | 305 | 0.0000 |

(d) Pipe-specification-4

| | Concrete | | | | Symbolic | | | |
|---|---|---|---|---|---|---|---|---|
| **sec.** | **S** | **U** | **?** | **Recall** | **S** | **U** | **?** | **Recall** |
| 1000 | 1 | 2858 | 0 | 0.0112 | 89 | 2770 | 0 | 1.0000 |
| 100 | 1 | 2858 | 0 | 0.0112 | 86 | 2770 | 3 | 0.9663 |
| 10 | 1 | 2858 | 0 | 0.0112 | 3 | 2770 | 86 | 0.0337 |
| 1 | 0 | 2795 | 64 | 0.0000 | 0 | 2758 | 101 | 0.0000 |

(e) Pipe-specification-5

| | Concrete | | | | Symbolic | | | |
|---|---|---|---|---|---|---|---|---|
| **sec.** | **S** | **U** | **?** | **Recall** | **S** | **U** | **?** | **Recall** |
| 1000 | 1 | 2858 | 0 | 1.0000 | 1 | 2858 | 0 | 1.0000 |
| 100 | 1 | 2858 | 0 | 1.0000 | 0 | 2857 | 2 | 0.0000 |
| 10 | 0 | 2851 | 8 | 0.0000 | 0 | 2773 | 86 | 0.0000 |
| 1 | 0 | 2799 | 60 | 0.0000 | 0 | 2607 | 252 | 0.0000 |

For all examples and abstraction levels, at least one match is found with the maximum solver time of 1000 seconds, which is fitting as each specification was derived from a pipe in the community. Despite the many results that can be found with the symbolic encoding, the concrete encoding is generally better at discarding programs that don't match. For all examples, and all ranges of solver times, the number of *unsat* programs for the concrete encoding is always greater than or equal to the number of *unsat* for its symbolic counterpart. This certainly makes sense for the longer solver times since the symbolic encoding has the potential to recognize more matches, yet, the trend is also true for the smaller solver times.

Solver time has a clear impact on the recall. Since cutting the solver time before it has reached a conclusion returns *unknown*, the recall is reduced as only the *sat* pipes are returned to the programmer. Treating the *unknown* pipes as results will increase recall to 1.00, but at the cost of precision.

### *RQ3: What is the Impact of Query Complexity on Search Time*

RQ3 explores the impact of size and complexity of specifications on the time for the solver to return *sat*.

**Artifacts.** To answer this question we required a careful manipulation of the specification. In order to do that, we selected a program (SQL select query) from stackoverflow and systematically decomposed it to generate input/output of different sizes and complexity. To identify that program, we searched stackoverflow postings for select statements containing the clauses we support and input/output, and selected the first one when ranked by number of votes. The selected program is: `SELECT balance FROM table WHERE balance >= 1000000 ORDER BY balance DESC;` . The decomposition consisted of combinations of its component clauses, as indicated in the key of Figure 3 (e.g., *select order* maps to `SELECT balance FROM table ORDER BY balance DESC;`).

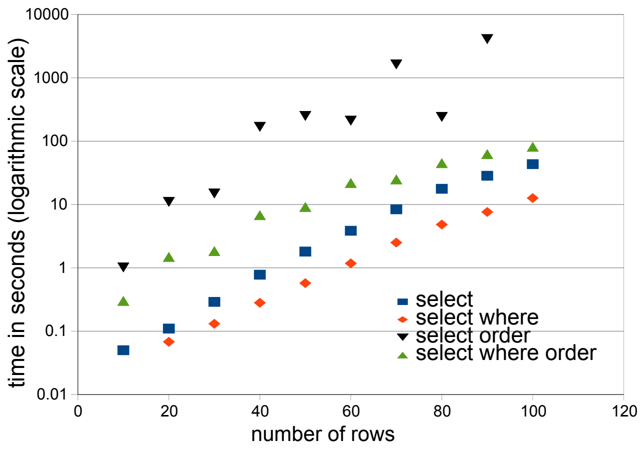


Fig. 3. Row Count Versus Solver Time

For each decomposed program, we generated input tables with 10 to 100 rows in increments of 10. Each input was generated to satisfy each SQL query. Since the `where` clause operates over the `balance` column, we generated the input tables using a normal distribution with $\mu = 1000000$. In this way, the output size was a function of the input size; when the `where` clause is present, the number of rows in the output is approximately half of the rows in the input. When the `where` clause is omitted, the sizes of the input and output tables are equal.

**Metrics.** For each program, and each input size, we report the time to return *sat*, averaged over ten runs (where each of the runs pulls a new input table from the normal distribution).

**Results.** Figure 3 shows the results of the experiment, with solver time on the y-axis in seconds (on a logarithmic scale) and the input size, in number of rows, on the x-axis. Each of the four decomposed programs maps to a line on the graph. As shown, the solving time increases exponentially with the number of rows for all the specifications. It is more subtle, however, how the complexity of the specification may impact the solving time. Specifications that require more clauses to be matched do not necessarily require more time. For example, the specification that requires `ORDER BY` as part of the solution takes more time than the one requiring `WHERE` and `ORDER BY`, as the expensive sorting constraints from `ORDER BY` need to operate on the smaller filtered dataset generated by `WHERE`. Further study is needed to tease apart these nuances, but it is clear that the application of multiple clauses makes the results harder to predict and that there is a trend of exponentially increasing solver time as the input size increases.

## VII. Discussion and Threats to Validity

Our evaluation explores different aspects of the approach in each of the three languages, and each comes with their own limitations and threats to validity. In the Java string study, we show that our search approach finds more relevant results than Google when using our local repository. In practice, however, syntactic searches thrive in large repositories. By applying our technique on top of snippets gathered from general Google searches, we are able to quickly discard an average of 34% of the code snippets that are irrelevant, and also identify some matches. We recognize two primary threats to validity. First, the syntactic queries were taken from the titles of the stackoverflow questions, and may differ from queries issued by the programmers. Second, some queries may require more than a one line solution, and by our current methodology, those potential solutions are ignored.

In the Yahoo! Pipes study, we found that symbolic encodings find many more relevant examples, but the concrete encodings are better at discarding irrelevant results. The relevant results were identified as those that would return *sat* eventually for some instantiation of the pipe. With this domain, the input is generated from a URL, which is stateful. Gathering the RSS feeds on a different day or at a different time can yield a different input/output, which can lead to a different set of relevant results.

With the SQL study, we show that solving time increases with input size (exponentially in all clause combinations). Our instantiation of SQL only works on integers, and it is likely that the time would be much longer in the presence of strings or other complex datatypes, so further study is needed.

Selection bias and potential implementation errors are two threats that may have affected the results on the three domains. We made our selection process explicit and develop extensive test suites to mitigate these threats.

Although we instantiate our approach on three different languages, the subsets covered perform relatively similar operations. Extending this work toward languages and constructs that require looping or branching has yet to be explored. An additional threat to validity comes from the fact that we have developed an approach to code search that is designed to help programmers, but we do not evaluate it in the hands of users. To show the benefits in practice requires an empirical study with actual programmers, but illustrating the generality, effectiveness, and efficiency of the approach are the first steps toward the ultimate goal of building an efficient code search engine for programmers.

## VIII. Conclusion

We have presented an approach to semantic search that uses an SMT solver to match lightweight specifications in the form of input/output pairs against programs whose behavior has been encoded as constraints. We describe how to encode search queries and programs in three languages, the Java string library, Yahoo! Pipes, and SQL select statements, and explore the effectiveness of our approach in each of these domains. While the approach seems promising in these domains, generality remains a goal and needs to be addressed in the context of richer programs, such as those contains loops and other complex constructs, and in the context of programmers using the approach.

### *Acknowledgments*

This work is supported in part by NSF SHF-1218265, NSF GRFP under CFDA-47.076, UNL-UCARE Program, a Google Faculty Research Award, and AFOSR #9550-10-1-0406.

## References

[1] R. Balzer. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng.*, 11(11):1257–1268, Nov. 1985.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 506–515. Ieee, 2007.

[3] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Int'l Conf. on Tools and Algos for the Construction and Anal. of Systems*, pages 307–321, 2009.

[4] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *International Symposium on Foundations of software engineering*, pages 214–225, 2008.

[5] N. Craswell and D. Hawkings. Overview of the rrec 2004 webl track. In *Proceedings of the 13th Text Retrieval Conference*, NIST, pages 1–9, 2004.

[6] G. Fischer, S. Henninger, and D. Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th international conference on Software engineering*, pages 318–328, 1991.

[7] C. Ghezzi and A. Mocci. Behavior model based component search: an initial assessment. In *ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, 2010.

[8] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. Exemplar: Executable examples archive. In *International Conference on Software Engineering*, pages 259–262, 2010.

[9] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Conf. on Prog. lang. design and implementation*, 2011.

[10] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, Dec. 2006.

[11] M. C. Jones and E. F. Churchill. Conversations in Developer Communities: A Preliminary Analysis of the Yahoo! Pipes Community. In *International Conference on Communities and Technologies*, 2009.

[12] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *International symposium on Software testing and analysis*, ISSTA '09, pages 105–116, 2009.

[13] Koders.com. http://koders.com/, August 2012.

[14] A. Langville and C. Meyer. *Google page rank and beyond*. Princeton Univ Pr, 2006.

[15] O. A. Lazzarini Lemos, S. K. Bajracharya, and J. Ossher. Codegenie:: a tool for test-driven source code search. In *Conf. on Object-oriented programming systems and applications companion*, 2007.

[16] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *International Conf. on Software Engineering*, 2011.

[17] J. Penix and P. Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6, April 1999.

[18] Yahoo! Pipes. http://pipes.yahoo.com/, June 2012.

[19] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2, July 1993.

[20] S. P. Reiss. Semantics-based code search. In *Proceedings of the International Conference on Software Engineering*, pages 243–253, 2009.

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

[22] K. T. Stolee and S. Elbaum. Toward semantic search via smt solver. In *Foundations of Software Engineering NIER*, 2012. to appear.

[23] K. T. Stolee, S. Elbaum, and A. Sarma. End-user programmers and their communities: An artifact-based analysis. In *Int'l Symp. on Empirical Soft. Eng. and Measurement*, pages 147–156, 2011.

[24] M. Veanes, N. Tillmann, and J. De Halleux. Qex: Symbolic sql query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 425–446. Springer, 2010.

[25] Z3: Theorem Prover. http://research.microsoft.com/projects/z3/, November 2011.

[26] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6, October 1997.