Solving the Search for Source Code

KATHRYN T. STOLEE, Iowa State University
SEBASTIAN ELBAUM and DANIEL DOBOS, University of Nebraska-Lincoln

Programmers frequently search for source code to reuse using keyword searches. The search effectiveness in facilitating reuse, however, depends on the programmer's ability to specify a query that captures how the desired code may have been implemented. Further, the results often include many irrelevant matches that must be filtered manually. More semantic search approaches could address these limitations, yet existing approaches are either not flexible enough to find approximate matches or require the programmer to define complex specifications as queries.

We propose a novel approach to semantic code search that addresses several of these limitations and is designed for queries that can be described using a concrete input/output example. In this approach, programmers write lightweight specifications as inputs and expected output examples. Unlike existing approaches to semantic search, we use an SMT solver to identify programs or program fragments in a repository, which have been automatically transformed into constraints using symbolic analysis, that match the programmer-provided specification.

We instantiated and evaluated this approach in subsets of three languages, the Java String library, Yahoo! Pipes mashup language, and SQL select statements, exploring its generality, utility, and trade-offs. The results indicate that this approach is effective at finding relevant code, can be used on its own or to filter results from keyword searches to increase search precision, and is adaptable to find *approximate* matches and then guide modifications to match the user specifications when *exact* matches do not already exist. These gains in precision and flexibility come at the cost of performance, for which underlying factors and mitigation strategies are identified.

Categories and Subject Descriptors: D.2.4 [**Software**]: Software-Engineering—*Software/Program verification*

General Terms: Verification, Languages, Experimentation

Additional Key Words and Phrases: Semantic code search, symbolic analysis, SMT solvers, lightweight specification

ACM Reference Format:
Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. ACM Trans. Softw. Eng. Methodol. 23, 3, Article 26 (May 2014), 45 pages.
DOI: http://dx.doi.org/10.1145/2581377

1. INTRODUCTION

Today, searching for code is a regular activity for most programmers. Consider a novice Java programmer who is trying to find a snippet of code that extracts an alias (i.e., username) from an email address. The programmer turns to the online search engine Google, the most common approach in practice [Sim et al. 2011, survey in Section 2], using a search query with the following keywords: *extract alias from email address in Java*. As expected, the query returns millions of results. None of the top ten results

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 no. 9550-10-1-0406.
Authors' addresses: K. T. Stolee (corresponding author), Iowa State University, Ames, IA; email: kstolee@iastate.edu; S. Elbaum and D. Dobos, University of Nebraska-Lincoln, Lincoln, NE.

DOI: http://dx.doi.org/10.1145/2581377

(*P*@10, a typical IR measure to assess the precision of search engine results [Craswell and Hawkings 2004]) even provides a method for decomposing an email 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.

Despite the simplicity of the programming task, this illustrates a common situation for programmers. Following a search for code to reuse, programmers must sift through many irrelevant results, especially when the desired behavior cannot be tied to source-code syntax or documentation. As repositories of source code grow in size and diversity, and as programmers continue to turn to search during development [Sawadsky et al. 2013], finding relevant code becomes increasingly important.

We have designed an approach to code search that addresses many of the shortcomings of existing search techniques, most notably by allowing programmers to describe what they want their code to do rather than how it is implemented. When no exact solutions exist, *close-enough* solutions can be found, which informally means solutions may require minor modification to fit the target context. The general idea is that programmers provide examples of the behavior of their desired code as inputs and outputs and an SMT solver identifies available source code from a repository, which has been encoded as constraints, that matches the specifications and can be reused. For example, when searching for source code that extracts the alias from an email address, the input could be the string "susie@mail.com" and the output the string "susie". While this form of query changes the search model from the common keyword query, it lets the programmer specify the desired behavior without the need to know how to achieve a certain outcome, just what that outcome is. The results of the search are source-code snippets that behave as specified. The proposed change, then, is from a syntactic query to a semantic query.

This example-based specification model is inspired by two lines of work, *programming by example* (or, *programming by demonstration*) and *program synthesis*. Some programming-by-example approaches aim to generate programs for tasks that are demonstrated by example [Cypher et al. 1993], such as providing a string before and after a transformation. Yet, the programs that can be generated through these approaches are limited and must follow well-defined templates or sequences. For example, in the TELS text-editing-by-example system [Witten and Mo 1993], programs are generated by recording and generalizing editing actions on strings in a text editor. For generalizations that involve string constants, they use a rigid hierarchy based on common subsequences. Other properties of strings, such as length- or case-insensitive equality, are not included in the hierarchy and thus would not be part of a generated program. In our work, we find *existing* code that performs a relevant transformation which promotes reuse. The intuition is that someone else has likely created a solution so there is no need for creating a new one from scratch. Our approach also does not depend on templates so it can return a larger variety of results, where the variety depends on the richness of the repository.

The other related line of work, program synthesis, aims to generate programs that match a provided input/output example, and program generation is guided by a constraint solver. Similar to programming by demonstration, this approach also relies on predefined functions and templates to guide the solver in finding a solution. The solver will try every possible combination of functions and templates to achieve the desired behavior, which can be time consuming even for small programs (e.g., in a toy problem, insertion and deletion on graphs can take several minutes to resolve [Singh and Solar-Lezama 2011]). Our approach is not restricted to predefined functions and templates, allowing us to return code that may be too complex for a code synthesizer to generate efficiently (we discuss more related work in Section 6).

Just like any other search engine, our approach indexes a repository of information offline, independently of the users' queries. Our indexing is unique in that it requires a transformer that uses symbolic analysis [Clarke 1976; Clarke and Richardson 1985; King 1976] to map a program's semantics onto constraints that summarize the program behavior. For example, the indexing process would map the two-line Java snippet

```
s1. int upper = input.indexOf('@');
s2. String output = input.substring(0, upper);
```

into the following constraints (roughly).

```
c1. (assert (input.charAt(upper) = '@') ∨ (upper = −1)))
c2. (assert (for (0 ≤ i < upper) input.charAt(i) ≠ '@'))
c3. (assert (for (0 ≤ i < upper) output.charAt(i) = input.charAt(i)))
```

Constraints `c1` and `c2` represent the first line of source code, `s1`. The first constraint, `c1`, defines `upper` as the location of "@" in `input` or -1, and `c2` asserts `upper` is the first index of "@" in `input`, per the semantics of the `indexOf` method in `java.lang.String`.[1] Constraint `c3` represents the second line of source code, `s2`. It asserts that the `output` matches `input` within bounds of `0` and `upper`, per the semantics of the `substring` method in `java.lang.String`. This is the basic process by which our approach indexes programs: mapping program semantics to constraints by evaluating each program statement. The constraints are generated automatically, a process we describe in Section 4.2. The constraints are never shown to the programmer, but rather are consumed by the solver during the search process to identify viable matches.

With a user-defined input/output query and an encoded repository of programs, the search can now find results. The first step in this phase is to transform the input/output into additional constraints. For the previous example, we get the following.

```
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 (described in Section 4.2), 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 conjunction of 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 email domain instead of the alias), the SMT solver would return *unsat* when paired with the previous code snippet, 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 the broader applicability of the approach to other domains, handling richer specification models required by diverse domains, and refining the set of potential matches. In this work, we begin to explore these issues.[2] We instantiate and assess various aspects of the approach in three domains: the Java String library, Yahoo! Pipes mashup programs, and SQL select statements (Section 4.2). These domains were selected in part to illustrate the generality of the approach by utilizing three diverse forms of input/output specification (Section 3) and in part because of their relatively simple and common underlying semantics and the

[1]According to the API, the value of `upper` must be -1 only in the event that '@' is not a character of `input`. Some additional constraints, not shown here for brevity, are required to prevent `upper` from defaulting to -1, which would make this system of constraints trivially satisfiable.

[2]Our previous work in this area presented a brief and preliminary instantiation of our approach on the Yahoo! Pipes mashup language [Stolee and Elbaum 2012b].

availability of repositories that could be searched for evaluation (Section 5). To refine the search results, we describe how the approach supports incremental strengthening of the specifications (queries) to prune the result set of coincidental matches and evaluate these concepts in the Java domain. Weakening the program encodings can enrich the results set with approximate matches that can be modified to match the specification, a process guided by the satisfiable model produced by the solver (Section 4), which we explore in the Yahoo! Pipes domain. We also explore how our Java search results compare to syntactic search results and illustrate how our technique can improve the results from a syntactic search engine, how changes in search parameters, specifically solving time and abstraction of the program encodings, affect the search results in Yahoo! Pipes, and how changes in the size and complexity of the search queries impact the search performance in SQL. The contributions of this work are:

(1) characterization of how developers use search to find code based on a survey of 99 participants;
(2) evidence of programmers using examples to explain their problem using an analysis of 300 questions posted to stackoverflow;
(3) definition of a novel approach to semantic code search that uses an SMT solver to identify matches given input/output examples and given programs encoded as constraints using symbolic analysis;
(4) instantiation of the approach in three domains: Java String library, Yahoo! Pipes, and SQL, illustrating the applicability of this approach;
(5) preliminary and broad evaluation of the approach:
    (a) comparison of search result relevance between a keyword-based approach and our approach in Java from the perspectives of 19 programmers;
    (b) proof-of-concept for combining syntactic and semantic search approaches to reduce the effort of evaluating search results;
    (c) exploration of the impact of solver time, specification size, and abstraction on precision and recall in Yahoo! Pipes; and
    (d) evaluation of the impact of specification size and complexity on solver time in SQL.

The rest of the article is organized as follows. Section 2 motivates this work using a survey of programmer search habits, and motivates the use of input/output queries by analyzing questions asked on an online help forum. Section 3 illustrates how we have instantiated the approach in each of the three targeted domains. Section 4 formalizes the approach definition and describes the domain-specific implementation details required for each instantiation of our approach. Our research questions, study, results, and threats to validity are presented in Section 5, followed by related work in Section 6 and the conclusion in Section 7.

2. MOTIVATION

Developers' contexts, workflows, tools, and languages vary widely. In this work we conjecture that code search is used across that variation and the input/output query model is a practical one. In this section, we motivate the need for further research in code search using a survey of 99 programmers about their code search habits. Next, we provide evidence for the utility of an input/output query model by exploring the frequency of input/output examples within questions asked in an online help forum.

2.1. Developer Survey on Code Search Habits

Previous work has studied the question of how and why programmers search for source code [Sim et al. 2011], with a survey that focused on graduate student behavior. Sim et al. showed that programmers most commonly search for code to reuse or to use as

Table I. Programming and Search Frequency

Activity	Daily	Weekly	Monthly
Programming	42	49	8
Code Search	25	52	22

a reference example and that Google is the most common, and often effective, tool for finding source code. To confirm the findings and understand more about how and why developers search for code and the tools used in code search, we conducted a survey with similar goals.

Our survey contained ten questions with the goal of addressing the following research questions:[3]

RQ1. How and why do programmers search for source code?
RQ1(a). How frequently do programmers search for code?
RQ1(b). Why do programmers search for source code?
RQ1(c). Which tools do programmers use to search for code?

2.1.1. Participants. We targeted two populations with this survey: students in two undergraduate classes at the University of Nebraska-Lincoln and workers on Mechanical Turk. Mechanical Turk [2010] is a service hosted by Amazon that allows people to reach and compensate others to complete tasks that require human input, such as tagging images or answering survey questions. It hosts the tasks, manages payment, and makes the tasks accessible to a large and existing workforce. With the Mechanical Turk population, we delivered the survey with four programming questions that required potential participants to analyze the behavior of simple Java methods. Correct responses were required for two or more questions in order for respondents to participate, as a means to control for quality.

In total, we received valid responses from 99 participants.[4] Of those, 42 came from junior/senior undergraduate classes at UNL while the remaining 57 came from Mechanical Turk. In a question about programming experience, 17 had less than two years of experience, 53 had between two and five years of experience, and 29 had more than five years of programming experience.

2.1.2. Search Frequency. To address *RQ1(a)*, we asked how frequently the participants write source code and how frequently they search for code. Table I summarizes our findings. Among all participants, 42 reported they program daily and 49 program on a weekly basis. Among the participants who program daily, over half (25 of 42) also search for code daily. As a summary, among those participants who program daily or weekly, 85% search for code at least weekly. This finding is consistent with a recent (and independent) survey that looked at the search habits of 36 graduate students [Sim et al. 2011]. It was reported that 50% of the participants search for code "frequently" while 39% do it "occasionally."

2.1.3. Why Programmers Search for Code. To address *RQ1(b)*, we asked participants what they did with the source code they were looking for. Once useful code is found, Table II summarizes what the participants did with it (using a multiselect question). Half of the participants reported that they would copy/paste and modify found code. Nearly three-fourths would use it to get ideas for implementation, and 11% would copy/paste

[3]Full survey details are available [Stolee 2013].

[4]Three participants were removed from the pool on account of inconsistent responses. These participants claimed to program weekly but search for source code daily, and this seemed suspicious. Ten participants from Mechanical Turk self-reported to have no programming experience, so those results were excluded as well.

Table II. Uses of Matched Code

Code Use	Count	Percent
Copy/paste as is	11	11%
Copy/paste and modify	52	52%
Get ideas for implementation	71	71%
Link to found code	9	9%
Other	4	4%

as-is. This is consistent with the previous findings that reuse and implementation examples are the most common purposes for a code search [Sim et al. 2011].

2.1.4. Tools Used in Code Search Activities. To address *RQ1(c)*, we asked participants about the tools they use for code search and the types of information they use for their search queries. In a free response answer about where the participants search for code, 69% mentioned using the Web, the Internet, or specifically Google. Nearly one-quarter (23%) mentioned searching stackoverflow specifically, a free question-and-answer site for programming- and technology-related questions. Only 17% mentioned using a code-specific search engine like Koders or Planet Source Code. Despite the availability of code-specific search engines, information search engines are the most common tools used for code search, echoing the findings in a previous survey [Sim et al. 2011].

Finding relevant source code, however, is not always easy with the current tools. The participants reported that they must explore an average of 3.5 snippets of code before something useful is found. A previous study found that approximately 3 out of the first 10 matches were useful, which aligns with our finding [Sim et al. 2011]. It is important to mention that neither survey accounted for the process of *query reformulation*, which involves restating a query after viewing irrelevant search results, is quite common in searching [Huang and Efthimiadis 2009], and adds to the overhead. Evaluating the cost of a syntactic search for finding source code to reuse is still an open question.

2.2. Input/Output Examples in the Wild

For the Yahoo! Pipes and SQL languages, our previous work has shown that programmers are able to compose input/output queries with 92% accuracy and in less than two minutes [Stolee and Elbaum 2013]. Yet how amenable programmers would be to this new query model has not yet been explored. In this section, we motivate the use of the input/output query model by investigating the extent to which programmers *already* compose examples in online help forums. We observe that Java, Yahoo! Pipes, and SQL programmers often turn to peer communities when they are looking for help. One popular community is stackoverflow, so this is the forum we use to identify input/output examples in the wild.

2.2.1. Sampling. We collected questions related to Yahoo! Pipes using the tag `[yahoo-pipes]`, questions related to SQL using the tags `[mysql]` and `[select]`, and questions related to Java using the `[java]` and `[string]` tags. The second tag in SQL was used to restrict the questions to those dealing with select statements, as that is the scope of our SQL implementation. The second tag in Java is meant to restrict the question to those dealing with strings, as this has been the primary focus of our implementation (Section 4.2.1). In Yahoo! Pipes, 248 questions were returned on March 27, 2013, in SQL, over 1,500 questions were returned on February 19, 2012, and in Java 7,420 questions were returned on June 17, 2013. In each domain, we sorted the results according to popularity (i.e., votes) and retained the top 100.

2.2.2. Analysis. An initial analysis was performed in the SQL domain to observe common question themes. This involved two passes over the questions. The first pass was

Table III. Question Type Categories in Stackoverflow

	SQL		YPipes		Java	
Question Type	#	Examples	#	Examples	#	Examples
How do I do *X* with {SQL, YP, Java}?	74	53 (72%)	58	40 (69%)	43	29 (67%)
Can I do *X* with/without *Y*?	8	4 (50%)	18	14 (78%)	9	4 (44%)
What's wrong with ..., or Why does ... work?	7	2 (29%)	9	8 (89%)	15	14 (93%)
How does *Y* work?	6	1 (17%)	4	1 (25%)	19	8 (42%)
What is an alternative to {SQL, YP, Java} for X?	0	0 (0%)	10	0 (0%)	0	0 (0%)
Best way to do *X*?	0	0 (0%)	0	0 (0%)	5	4 (80%)
Y versus *Z*?	4	1 (25%)	0	0 (0%)	9	4 (44%)
unrelated	1	0 (0%)	1	0 (0%)	0	0 (0%)

Reported are the number of questions for each category as well as the number and percentage of those questions that contain descriptive or input/output examples.

for content analysis to collect common question themes and the second pass categorized the questions. This same process was repeated for the Yahoo! Pipes domain, and then on Java. When a question fit two or more categories, we selected the category that most closely fit based on the accepted or highest-voted answer from the community. Eight categories emerged from this analysis. The next step was to check if the questions also had examples to illustrate their context, and if these examples were in the form of an input and output.

2.2.3. Results. In each domain, there were a handful of dominant question types, shown in Table III. In a question containing *X*, it represents a specific task, such as *remove a field from an RSS item* in Yahoo! Pipes, *combine two tables* in SQL, or *capitalize the first letter in a string* in Java. *Y* refers to a language construct, such as the `Regex` module in Yahoo! Pipes, the `Inner Join` construct in SQL, or the `hashCode()` function in Java. For each question, we provide the frequency of occurrence in each language (column #), as well as the number of those questions with which a descriptive or input/output example was provided (column *Examples* shows a count and the percentage). As an example, *How does Y work?* questions describe six questions from SQL, four questions from Yahoo! Pipes, and 19 questions from Java; one question in each of Yahoo! Pipes and SQL provided an example to more clearly illustrate the question being asked (representing 17% and 25% of the questions, respectively). In Java, examples were more common with eight of those questions providing examples, representing 42%.

The dominant type of question for all languages is "How do I do *X*?" This represents 74 questions in SQL, 58 questions in Yahoo! Pipes, and 43 questions in Java. A sample of this type in Java is as follows.

> "How does one convert a String to an int in Java? I have a string which contains only numbers (1–4 numbers to be specific), and I want to return the number which it represents.
> For example, given the string "1234" the result should be the number 1234."[5]

This question is asking for how to do a type conversion in Java. Generally, syntactic search mechanisms are not well equipped to answer this type of question as the developer does not know what query or query components may be used to solve the problem; the developer asking this type of question knows only the behavior that is desired.

We also observed that these questions usually come with examples that help developers better specify the required behavior. Of the 74 "How do I do *X*?" questions in SQL,

[5]http://stackoverflow.com/questions/5585779/how-to-convert-string-to-int-in-java.

53 contained examples. Further, 36 examples were actual snippets of tables that serve as inputs and records that they expect as outputs. Of the 58 questions in Yahoo! Pipes of this type, 40 contained examples and 8 of those had inputs and outputs. In Java, 29 of the 43 questions contained examples, and 13 of those were inputs and outputs. In some questions, the outputs were difficult to specify. One Java question asks how to convert a string to a Date() object.[6] While the input is easy to specify, the output is not, and thus this was not counted as an input/output example. To work around that difficulty, some questions used test cases to illustrate input/output examples in a more standard format.[7]

From this analysis, we see evidence that programmers already think in terms of examples when trying to accomplish tasks using the SQL and Java, and to a lesser extent, the Yahoo! Pipes language. Another interesting observation is that "How does *Y* work?" questions are not asked often by programmers to their community, particularly for SQL and Yahoo! Pipes. This may indicate that this type of question is well handled by existing resources like existing documentation, tutorials, or other syntactic search engine findings.

2.3. Summary

We have presented the results of a survey that asked 99 participants about their programming experience and search habits. We observed that code search is common, Google is the most frequently used tool, and the overhead, which comes from examining and determining whether or not a match is useful, is nontrivial. However, these results are based on self-reported answers and may not be representative of actual behavior. Observing search habits in practice would allow us to validate these results.

Next, since state-of-the-practice code search requires the use of a keyword query and our approach uses an input/output query model, we also explored the frequency of input/output specifications in the wild, observing that examples are commonly used in questions asked on stackoverflow.

The next section provides some illustrative examples of our input/output search approach in the three targeted languages, Java, Yahoo! Pipes, and SQL.

3. ILLUSTRATIVE EXAMPLES

We started exploring this approach to semantic search in the context of end-user programming languages, specifically a Web mashup language called Yahoo! Pipes which performs operations on lists of RSS items. To generalize the approach to a more common language with similar semantics, we targeted SQL select statements that perform similar filtering operations (in order to reuse the transformation infrastructure we had developed), but over tables of data, which added a dimension of complexity to the implementation. Supporting the Yahoo! Pipes language fragment also requires operations on strings, specifically identifying equality and substring relationships. To build on that support and explore our search approach in a broader context, we have targeted Java program snippets that contain calls to the `java.lang.String` library.

In this section, we illustrate the benefits and unique features of our approach to search through a series of examples. These examples are intended to represent situations in which the input/output example-based search may be useful. Additionally, for each domain, we briefly describe how search is currently performed and how our approach is instantiated.

[6]http://stackoverflow.com/questions/4216745/java-string-to-date-conversion.

[7]http://stackoverflow.com/questions/2559759/how-do-i-convert-camelcase-into-human-readable-names-in-java.

3.1. String Manipulations in Java

The alias extraction example in Section 1 illustrates a state-of-the-practice search for code to reuse which returns many irrelevant results that must be evaluated manually. Similar situations are likely quite common, given that most programmers we surveyed frequently utilize syntactic search to find code to reuse or to obtain examples (Section 2.1).

Our search approach requires a query consisting of example input and expected output pairs, such as "susie@mail.com" as input and "susie" as output. In the context of the Java String library, those inputs and outputs could be one of several datatypes; integers, characters, strings, and booleans are supported by our current implementation. Here, we provide four examples in the Java domain to illustrate key aspects of our search. First, we show how to bind input/output specifications to snippets of code; second, we show how refinement on the specification can impact search results; third, we show how to handle more complex code examples; fourth, we illustrate how ambiguity in code snippets is handled.

Example 1. Consider a programmer who wants to find the length of a file extension (including the dot). For the desired code snippet, the query's input is the file name, let's assume as a string, and the query's output is the length of the extension as an integer. A concrete query to illustrate this behavior could be the input "foo.txt" with an expected output of 4. To provide a data point on performance (more extensively assessed in Section 5), our search with that concrete query identifies 83 potential matches from a repository with hundreds of encoded programs. The following snippet is a match that involves four API calls.

```
int begin = s.lastIndexOf(".");
int end = s.length();
String ext = s.substring(begin, end);
int len = ext.length();
```

Here, the input is bound to the only undefined variable in the code snippet, `s` (inferred to be of type string). The output is bound to the LHS of the final assignment statement, `len`, which is the only unused variable. These bindings are calculated by the approach by computing and exploring the def-use pairs [Nielson et al. 2004].

There may be many potential bindings of an input/output specification to an arbitrary code snippet, with some bindings being better than others. For instance, if there are multiple undefined variables in a snippet and multiple elements in the input, some bindings may lead to a satisfiable results whereas others may not; we discuss this later in Section 4.2.

Example 2. For the previous example, the input/output specification yielded 83 potential matches. In Section 1, we presented a specification that could be used as a query to find code that extracts an alias from an email address. The input, "susie@mail.com", and the output, "susie", form the specification. With this input/output pair encoded as constraints, our search returned 51 matches. In these searches, the specifications are relatively weak so many results may be irrelevant. For the alias extraction example, consider the following independent results, `r1` and `r2`.

```
r1. String scheme = uri.substring(0, 5);
r2. username = to.substring(0, to.indexOf('@'));
```

The first result, `r1`, is found by binding the output to `scheme` and the input to `uri`. The second result is found by binding the output to `username` and the input to `to`. Deciding which results are actually relevant, rather than coincidental, may not be straightforward. To help with this process, the developer can provide additional input/output pairs to prune coincidental matches. For example, adding the input/output pair,

$input_2$ = "alex@univ.edu" and $output_2$ = "alex", will remove `r1` from the result set (i.e., it only matches the first input/output because the string "susie" has five characters), leaving `r2` as a result.

Example 3. This approach is also effective at retrieving larger snippets of code. Consider a programmer who wants to obtain the subdomain from a domain name, for example, by providing an input of "http://subdomain.example.com" and output "subdomain"[8]. The following code snippet will match the specification.

```
int lastIndex = domain.lastIndexOf(''.'');
String noext = domain.substring(0, lastIndex);

lastIndex = noext.lastIndexOf(''.'');
String subdomain = noext.substring(0, lastIndex);

int firstIndex = subdomain.lastIndexOf(''/'', lastIndex);
firstIndex = firstIndex + 1;

return subdomain.substring(firstIndex, lastIndex);
```

For this code, the variable `domain` is used but never defined, making it the input. The return statement forms the output values, as would be the case if this snippet was encapsulated in a method. Encoding will follow the same process as with the other examples, mapping the code to constraints. Unlike the previous examples, this code involves more complexity from the definition and use of several String and integer variables, and illustrates how the approach can work with larger, more complex code. Constructs that modify the control flow, such as loops and predicates, are not part of the current implementation (Section 4.2.1), but are increasingly being supported by symbolic analysis engines. We discuss this related work in Section 6.

Example 4. In some snippets, there may be additional variables that are used, but are not defined and also not bound to the input. Consider the following snippet that matches for the input/output used in Example 1.

```
int index = names.length() - names.indexOf( flag );
```

After binding the input to `names` and the output to `index`, this code is not executable because we know nothing about the value of `flag`, so state-of-the-art semantic search engines that utilize test cases to identify matching code (e.g., Lazzarini Lemos et al. [2007], Podgurski and Pierce [1993], and Reiss [2009]) would fail. In our approach, the uninitialized variables in the snippet remain uninitialized in the encoding, and we make no assumptions about the values they hold (though we must use type inference to reveal that `flag` is either a character or a string, and we assume the more expressive case of string). This snippet is identified as a match because the satisfiable model produced by the solver reveals that the specification matches this snippet when `flag` is set to ".txt" (the solver could have identified ".", ".t", or".tx" as possible values, but it only needs to find one to complete the model). By encoding the behavior of the snippets as constraints, we can identify incomplete code as a match and leverage the solver to guide its instantiation. Applying that guidance yields the following modified and complete code.

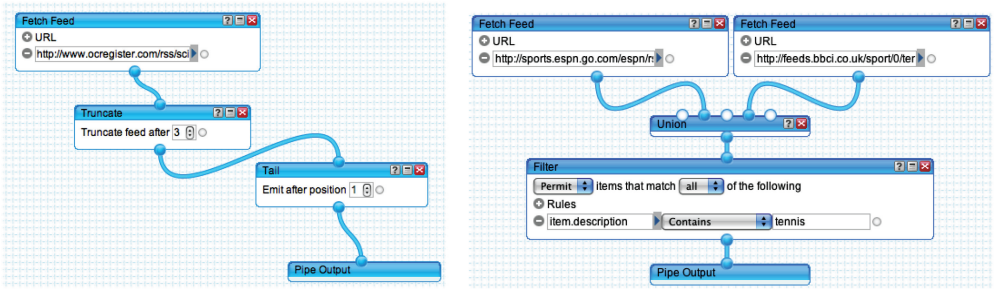
```
int index = names.length() - names.indexOf( ".txt" );
```

Clearly, this code would not be considered a match for other input/output examples in which the file extension is not ".txt". A working solution could be found by adding additional input/output examples and forcing `flag` to equal ".".

[8]http://stackoverflow.com/questions/1189128.

Field	Value
`Title`	Your Local Doppler Radar
`Description`	This map shows the location and intensity of precipitation in your area. The color of the precipitation corresponds to the rate at which it is falling. This map is updated every 15 minutes.
`Link`	http://www.weather.com/weather/map/93012
`Date`	Fri Jan 13 11:15:22 CST 2012

Fig. 1. Example record/item from an RSS feed.



(a) pipe 1 (b) pipe 2

Fig. 2. Two example pipes, possible solutions to input/output example in Figure 4.

We refer to uninstantiated variables, like `flag`, as *symbolic* and variables that hold values, like the string ".txt", as *concrete*.

3.2. Yahoo! Pipes Mashups

Often, existing search capabilities of domain-specific languages are even more limited than those for more mainstream languages. Yahoo! Pipes is a mashup language with over 90,000 users [Jones and Churchill 2009], and a public repository of over 100,000 artifacts [Pipes 2012]. These programs combine, filter, sort, annotate, and manipulate RSS feeds. To write a Pipes program, programmers use the Pipes Editor, dragging and dropping predefined modules and connecting them with wires to define the data- and control flow. Example pipes are shown in Figure 2(a) and Figure 2(b). A pipe can have multiple sources (e.g., a *Fetch Feed* module) that access external data sources (e.g., URLs), and exactly one sink, the *Pipe Output* module shown at the bottom.

Each *Fetch Feed* module provides a list of RSS items to the pipe. Each item is a map data structure with key-value pairs. An example item, also called a record, from an RSS feed is shown in Figure 1. The keys are `Title`, `Description`, `Link`, and `Date`. The first three keys map to values of type string, and the last key maps to an integer (i.e., the date is converted to an integer). Mashup programs perform operations on the lists of RSS items. The operations are defined by the modules that connect the source(s) and the sink. In Figure 2(a), the pipe performs a head operation on the list (the *Truncate* module), and then a tail operation (the *Tail* module). In Figure 2(b), the pipe concatenates two data sources with a *Union* module and then retains items in the RSS feeds that contain the word "tennis".

In the state-of-the-practice, programmers can search for pipes by URLs accessed, tags, keyword, or program components. To illustrate the challenges programmers face with current search support, we performed five searches by URL (see Table VII in Section 5). The number of matches can be in the thousands, which is not surprising as many mashups include common URLs. The precision among the top ten results (P@10 [Craswell and Hawkings 2004]), determined by the behavior of the pipe, is

Module	Encoding as Constraints
Fetch_1	c1: (assert (in(Fetch_1) = URL_1)) c2: (assert (out(Fetch_1) = in(Fetch_1)))
Fetch_2	c3: (assert (in(Fetch_2) = URL_2)) c4: (assert (out(Fetch_2) = in(Fetch_2)))
wire_1	c5: (assert (in(Union_1) = out(Fetch_1)))
wire_2	c6: (assert (in(Union_2) = out(Fetch_2)))
Union	c7a: (assert (for ($0 \leq$ i $<$ size(in(Union_1))) recOf(out(Union), i) = recOf(in(Union_1), i))) c7b: (assert (for (size(in(Union_1)) $\leq$ i $<$ (size(in(Union_1)) + size(in(Union_2)))) recOf(out(Union), i) = recOf(in(Union_2), (i - size(in(Union_1)))))) c8: (assert (for ($0 \leq$ i $<$ size(out(Union))) (hasRec(in(Union_1), recOf(out(Union), i)) = true) $\vee$ (hasRec(in(Union_2), recOf(out(Union), i)) = true)))
wire_3	c9: (assert (in(Filter) = out(Union)))
Filter	c10: (assert (for ($0 \leq$ i $<$ size(in(Filter))) ((recOf(in(Filter), i) = r) $\wedge$ contains(field(r "descr"), "tennis")) $\Rightarrow$ (hasRec(out(Filter), r) = true))) c11: (assert (for ($0 \leq$ i $<$ size(out(Filter))) (hasRec(in(Filter), recOf(out(Filter), i)) = true))) c12: (assert (for ($0 \leq$ i $<$ size(out(Filter))) (for (i $<$ j $<$ size(out(Filter))) ((recOf(out(Filter), i) = r_1) $\wedge$ (recOf(out(Filter), j) = r_2)) $\Rightarrow$ ($\exists$ k, l ((k $<$ l) $\wedge$ ($0 \leq$ k $<$ size(in(Filter))) $\wedge$ ($0 \leq$ l $<$ size(in(Filter))) $\wedge$ (recOf(in(Filter), k) = r_1) $\wedge$ (recOf(in(Filter), l) = r_2))))))
wire_4	c13: (assert (in(Output) = out(Filter)))
Output	c14: (assert (out(Output) = in(Output)))

Definitions: Let l be a *List*, i be an *Integer*, r be a *record*, and s_1, s_2 be strings
$contains(s_1, s_2) = true \iff s_2 \subseteq s_1$
$recOf(l, i) = r \iff l[i] = r$
$hasRec(l, r) = true \iff \exists i \mid ((0 \leq i < size(l)) \wedge (l[i] = r))$

Fig. 3. Mapping the pipe in Figure 2(b) onto constraints.

0.06. Using other built-in search capabilities does not fare much better. Searching by components retrieves even more results and requires the programmer to know the pipe implementation details. The effectiveness of searching with tags is highly dependent on the community's ability and decision to systematically categorize their artifacts.

The external data sources accessed by the *Fetch Feed* modules are the inputs to the pipe. In our approach instantiated in the Pipes domain, the programmer provides the URLs for RSS feed(s) as input, just as he/she would to build a pipe from scratch. Like the Pipes Editor environment, our framework fetches the RSS feed; this produces the input list. The programmer modifies this list by reordering, removing, or modifying items to form the output list. An example of this process is shown in Figure 4. The programmer provides the URL, such as the *Input* in Figure 4, and our framework retrieves the RSS feed, which has n items. The programmer selects item(s) as the desired output.

In this domain, entire programs are encoded as constraints. The URL information is abstracted away so the pipe can be solved for any URL provided as input; this abstraction is imperative to find pipes that behave as desired, given an example input and output.

The encoding process is briefly illustrated in Figure 3 for the pipe in Figure 2(b). Each module is mapped to a set of constraints, and each connector (called wires) defines the relationships between the modules. The module constraints are expressed in terms of the input to and output from the module (e.g., `in(Filter)` refers to the list that enters

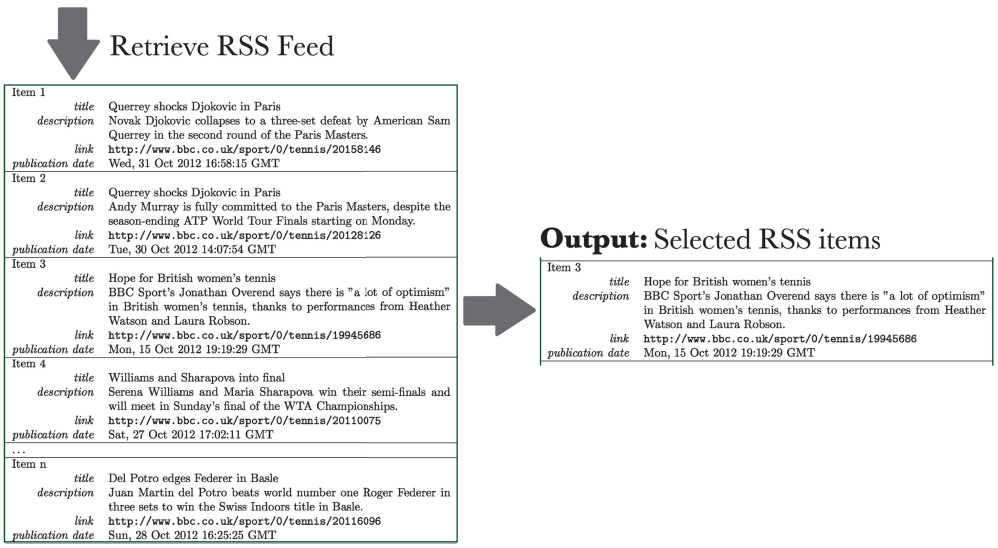


Fig. 4. Yahoo! Pipes input/output example.

the *Filter* module, and `out(Filter)` refers to the list that exits the *Filter* module). Constraints $c1$ and $c3$ assign input variables to each of the *Fetch Feed* (succinctly, *Fetch*) modules. Constraints $c2$ and $c4$ ensure that the output from the *Fetch* modules is the same as the input. Constraints $c5$ and $c6$ connect the output from the *Fetch* modules to the *Union* module as inputs. The *Union* module concatenates its input lists, which are described by constraints $c7a$, $c7b$, and $c8$. The first, $c7a$, ensures that all the items at the front of the output list, `out(Union)`, come from the first input list, $\texttt{in(Union}_1\texttt{)}$, and the second constraint, $c7b$, ensures that the next items are from $\texttt{in(Union}_2\texttt{)}$. This is called *inclusion*. The next constraint, $c8$, ensures that all items in the output list from the module exist in one of the two input lists, and in this way no extra items are appended to the end, enforcing *exclusion*. The output from the *Union* module goes to the *Filter* module per $c9$. Representing the *Filter* module requires three constraints that enforce *inclusion*, *exclusion*, and *order* properties. The first, $c10$, ensures that all items in `in(Filter)` that contain "tennis" in the description also exist in the `out(Filter)` list. The *exclusion* constraint, $c11$, ensures that all records in the output are also from the input (i.e., $\texttt{out(Filter)} \subseteq \texttt{in(Filter)}$). The final constraint, $c12$, ensures that if two records exist in the output list, their ordering is the same as in the input list. In this way, the module is order preserving. Constraint $c13$ ensures that the output from the *Filter* module goes to the input of the *Output* module, and $c14$ ensures that the output of the pipe, `out(Output)`, is the same as `in(Output)`.

Example 5. Say a programmer wants to collect news about tennis from a Web site, and created the specification shown in Figure 4 (the selected item for the output list contains "tennis" in the description). Searching a repository of programs (Section 5) reveals two possible matches, both shown in Figure 2. The first solution performs head and tail operations on the list to extract the third item, whereas the second solution joins two RSS feeds and permits items that have "tennis" in the description. While both match the specification, the first is likely a coincidental match and could be pruned by adding another input/output example, as demonstrated in Example 2.

Much like in Example 4 where the variable `flag` was symbolic, since the specification only had one input URL, the pipe solution in Figure 2(b) has an uninitialized input, URL_2. In this domain, instead of leaving the RSS feed symbolic, we assume unbound fetch modules have empty input lists. That is, URL_2 is set to an empty list. This is done because the RSS feeds are external resources and, if left symbolic, the solver may identify a program as a match but require an RSS feed that does not exist.

Example 6. When a matching program cannot be found, we can apply abstractions to the encodings to find code that does not exist as such, but is a close-enough match that can be instantiated to meet the user specifications. For example, say a programmer wants to filter an RSS feed based on "volleyball" rather than "tennis". The inclusion constraint for the *Filter* module in Figure 2(b), $c10$, contains as part of the implication, `contains(field(r "descr"), "tennis"))`. At a concrete abstraction level, the string "tennis" is encoded as-is, which would not satisfy a specification that requires "volleyball". However, with a weaker encoding consisting of constraint `contains(field(r "descr"), s))` for some string s, an SMT solver could determine that for $s =$ "volleyball", this program is a match.

This form of abstraction allows the search to identify programs that are approximate matches for the desired behavior, and can be modified systematically to satisfy the input/output specifications, similar to Example 4 where `flag` was instantiated. We implement and evaluate two abstraction levels within the pipes domain (Section 5).

3.3. SQL Select Statements

SQL select statements support data retrieval, operating on their own or being embedded into other languages. Given the simplicity of the SQL syntax and its popularity, even well-conceived syntactic searches for examples will return many irrelevant results.

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 that are the programs for which programmers search. Given example tables as input and output, the SMT solver determines, for each encoded SQL select statement, if it matches the specification.

Example 7. Consider the programmer who asked the question on stackoverflow, "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$?"[9] The programmer knew the desired behavior and described it through a concrete input example table.

```
 id  | username  |    balance  | status
-----+-----------+-------------+--------
 145 | rekin76   |   469370.44 |   0
  56 | avcio     |   466921.90 |   0
 705 | shantee   |   149160.09 |   0
5725 | ter       |    93004.45 |   0
3414 | rut1999   |    80944.80 |   0
 ... | ...       |     ...     |  ...
```

Based on the accepted answer from stackoverflow, we created an output table.

```
 username
-----------
 rekin76
 avcio
 shantee
```

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

With this input/output specification in the form of tables, our approach identifies as a match the recommendation of three positively voted responses in stackoverflow.

```
SELECT username FROM table WHERE balance >= 1000000 ORDER BY balance DESC
    LIMIT 10;
```

An interesting aspect of this domain is that the input/output specifications can be large since they may come from live databases. It becomes important to understand the impact of large specifications on solver time. As we explore in Section 5.4, it is not just the size of specification that matters but also the complexity of behavior exhibited in the specification.

Example 8. Consider a programmer who wants to extract salary information for employees from a database. The programmer has two database tables, one called `employee` with fields `[id, name, address]`, and another called `payroll` with fields `[id, account, salary]`. His/her desired output table contains `[name, salary]` for each employee `id`, which requires combining the two input tables as is done in the following query.

```
SELECT name, salary FROM employee, payroll WHERE employee.id = payroll.id
    ORDER BY salary;
```

This query requires an implicit join on the `id` field for the two input tables in order to create the output table. Our approach supports the case when multiple inputs form a single output. Such a query is possible in any of the supported domains (e.g., multiple strings in Java, multiple URLs in Yahoo! Pipes, or multiple tables in SQL), and is common in database queries that require merging multiple tables.

3.4. Summary

At this point, we have discussed several interesting aspects of our approach to semantic code search in three domains, illustrating the generality of the approach, showing how it can overcome many of the limitations of state-of-the-practice syntactic searches, and addressing challenging issues associated with state-of-the-art semantic searches. Specifically, our search approach is semantic rather than syntactic, returning results that match a behavioral example rather than a set of keywords. Our approach is not limited to complete programs, but can also work with incomplete code. We have also shown how a programmer can identify relevant code when there are many coincidental matches by adding additional input/output examples and how our search can use abstraction to identify and instantiate matching code that did not previously exist.

4. APPROACH

We present the general definitions of each piece of our approach, followed by details on our instantiation of the approach in each of the three supported domains: Java String library, Yahoo! Pipes programs, and SQL select statements. We also discuss the performance and effectiveness of the approach.

4.1. Components

Our general approach is illustrated in Figure 5 and Figure 6. The offline process of building the repository is depicted in Figure 5 and the online search process is depicted in Figure 6. The gray boxes indicate the key components and technical challenges: defining input/output specifications (Figure 6), encoding programs (Figure 5) and specifications (Figure 6) as constraints, abstracting program encodings when too few matches are found (Figure 6), and refining specifications when too many matches are found (Figure 6). The crawling and program encoding processes happen offline, whereas the query specification, query encoding, and solving for relevant code happen online.

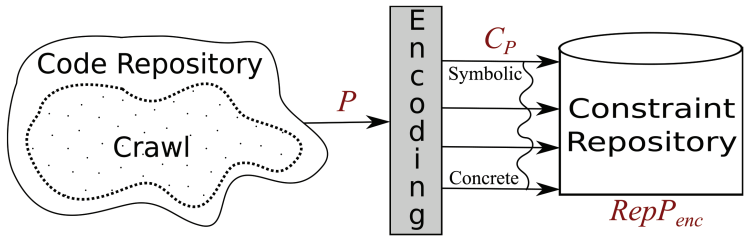


Fig. 5. Building and encoding the repository (offline).

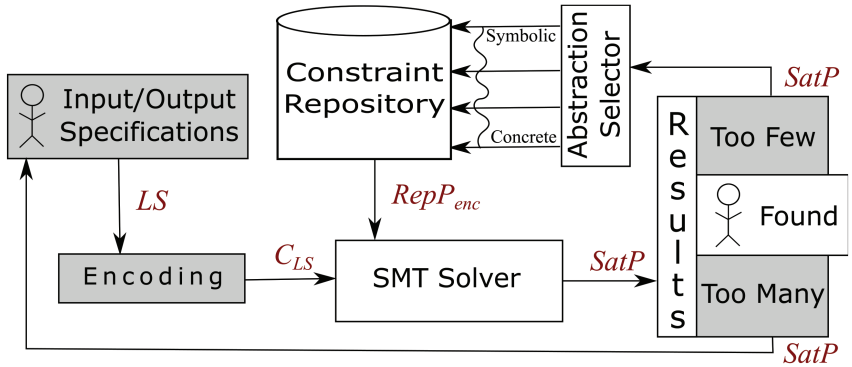


Fig. 6. Search process (online).

4.1.1. Specifying Behavior. Instead of keyword queries, our approach takes behavioral specifications that characterize an example of the code behavior (*Input/Output Specifications* in Figure 6). These are lightweight specifications (LS), in that they are incomplete and weak. As illustrated in Section 3, the inputs and outputs take different forms depending on the domain. 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, as was illustrated with Example 2 in Section 3.1. The size of k defines, in part, the strength of the specifications and hence the number of potential matches.

The last step of this process is the automated encoding of LS into constraints, C_{LS}, for the solver to consume when the search starts (recall the transformed specifications, $c4$ and $c5$, from Section 1).

4.1.2. Encoding. In our approach, encoding and solving are analogous to crawling and indexing performed by search engines [Langville and Meyer 2006]. Offline, a repository (*Code Repository* in Figure 5) is crawled to collect programs. These programs are parsed and encoded as constraints using symbolic analysis [Clarke 1976; Clarke and Richardson 1985; King 1976]; the constraints are stored in a *constraint repository*.

More formally, given a collected set of programs $RepP = \{P_1, P_2, \ldots P_n\}$, our encoding engine first uses a grammar to parse the programs. Since our approach is meant to support many languages, a different grammar is required for each language (see Section 4.2). For each parsed program P, we identify its input and output, which are encoded symbolically so the program can be matched against any arbitrary LS with matching types. We then use a symbolic analysis on P to create a symbolic summary of the behavior of the code, represented in conjunctive normal form as $C_P = c_1 \wedge c_2 \wedge \cdots \wedge c_m$. Encoding occurs at a given abstraction level, as discussed with Example 6 in

Section 3.2 and described in Section 4.1.5. In the end, the encoding process maps every program to a set of constraints such that $RepP_{enc} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$.

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. In Section 4.2, we explore encoding at the component level (Yahoo! Pipes), query level (SQL), and library level (Java).

4.1.3. Solving. The constraint repository, $RepP_{enc}$, is used by the solver in conjunction with the encoded specifications, C_{LS}, to determine matches (*SMT Solver* in Figure 6). Given C_{LS}, for each $C_P \in RepP_{enc}$ such that the types on the inputs and outputs match the types in C_P, the approach invokes $Solve(C_P \wedge C_{LS})$. The potential return values are *sat*, *unsat*, or *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 search results, or *SatP*, consist of all programs that return *sat*.

In practice, 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, which is set using the *Abstraction Selector*, shown in Figure 6. We begin by trying to solve for the strictest (most concrete) level, but this 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 5.4, it can take several minutes for the solver to return *sat* or *unsat*, so setting a short maximum solver time can lead to an efficient search, though it can miss some matches and impact recall.

4.1.4. Strengthening and Weakening the Specifications. If the specifications or the encoded program constraints are too weak, many matches may be returned (*too many* in Figure 6). Refinement is a process that helps to address these situations by tuning the Lightweight Specifications (LS'). A programmer may strengthen the specifications by providing additional (i, o) that further demonstrate the desired behavior, similar to query reformulation [Fischer et al. 1991; Haiduc et al. 2013]. 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 precision in Yahoo! Pipes in Section 5.3 and on search efficiency for SQL in Section 5.4.

4.1.5. Abstraction on Program Encodings. If the program encodings are too strong, the solver may not yield any results (*too few* in Figure 6). Abstraction is a process that uses weakened program encodings to find solutions that are close enough when no exact solutions exist, as was illustrated by Example 6 in Section 3.2. These approximate solutions can be instantiated to match the user's specifications by resetting the abstracted variables' values. Selecting weaker encodings for the search process is controlled by the feedback loop to the *Abstraction Selector* in Figure 6.

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*return*⟩

⟨*assignment*⟩ ::= ⟨*type*⟩? ⟨*ident*⟩ '=' ⟨*expr*⟩ ';'

⟨*return*⟩ ::= 'return' ⟨*expr*⟩ ';'

⟨*expr*⟩ ::= ⟨*ident*⟩ ('.' ⟨*method*⟩)*
| ⟨*stringLiteral*⟩
| ⟨*charLiteral*⟩
| ⟨*integer*⟩
| ⟨*booleanLiteral*⟩

⟨*method*⟩ ::= ⟨*noparam*⟩ '(' ')'
| ⟨*oneparam*⟩ '(' ⟨*expr*⟩ ')'
| ⟨*twoparam*⟩ '(' ⟨*expr*⟩ (',' ⟨*expr*⟩)? ')'

⟨*noparam*⟩ ::= 'length'

⟨*oneparam*⟩ ::= 'charAt' | 'concat' | 'contains' | 'endsWith' | 'equals' | 'startsWith'
| 'substring'

⟨*twoparam*⟩ ::= 'indexOf' | 'lastIndexOf'

⟨*type*⟩ ::= 'char' | 'String' | 'int' | 'boolean'

⟨*ident*⟩ ::= (LETTER | '_' | '$')(LETTER | '_' | DIGIT | '$')*

Fig. 7. Java supported grammar.

To weaken the encodings, we exploit the fact that most languages contain constraints over multiple datatypes (e.g., strings, characters, integers, booleans) for which the variable values can be relaxed and encoded as symbolic. 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 [Penix and Alexander 1999; Zaremski and Wing 1997]. *Weakening* : $C_P \rightarrow C'_P$ means that $(Solve(C_P \wedge C_{LS}) = unsat) \wedge (Solve(C'_P \wedge C_{LS}) = sat)$ for some relaxation of C_P that yields C'_P. We explore the impact of various abstraction levels on search time in Yahoo! Pipes in Section 5.3.

4.2. Implementation

For each of the three supported languages, we present the grammar that is used in the encoding process and domain-specific details about the symbolic analysis required for each implementation.[10] Our encoding engine transforms programs into SMT-LIB2 [2012] format. Solving is performed by Z3 v.4.1 [De Moura and Bjørner 2008].

4.2.1. String Manipulations in Java. Our implementation supports the subset of the Java language shown in Figure 7. Following the ANTLR syntax, all terminals are identified using single quotes. Angle-brackets are used to denote nonterminals. Some nonterminals are not defined here, specifically `<stringLiteral>`, `<charLiteral>`, `<integer>`, and `<booleanLiteral>`, as these follow the standard definitions in the Java grammar. From `java.lang.String`, we support the following library calls: `charAt, concat, contains, endsWith, equals, indexOf, lastIndexOf, length, startsWith`, and `substring`. To efficiently support these operations, we consider bounded strings where the bound is configurable (in line with recent work on solving string constraints [Bjørner et al. 2009; Kiezun et al. 2009]).

[10]The programs we currently support contain only a single program path, so full symbolic execution is unnecessary for the work presented here; see Section 6 for a more thorough discussion on this.

$\langle pipe \rangle$ ::= 'output' $\langle composition \rangle$

$\langle composition \rangle$ ::= $\langle operator \rangle ?$ $\langle grouping \rangle$
| $\langle init \rangle$

$\langle grouping \rangle$ ::= 'union' $\langle segment \rangle +$

$\langle segment \rangle$ ::= '(' $\langle init \rangle$ ')'
| '(' $\langle operator \rangle ?$ $\langle grouping \rangle$ ')'
| '(' ('(' $\langle operator \rangle$ ')')* ')' 'split' $\langle composition \rangle$

$\langle init \rangle$ ::= $\langle interior \rangle ?$ 'fetch'

$\langle operator \rangle$ ::= ('filter' | 'sort' | 'truncate' | 'tail')+

Fig. 8. Yahoo! Pipes supported grammar.

Two types of statements are supported in this grammar, *assignment* and *return* statements. For assignment statements, the LHS constitutes the program output. For both assignment and return statements, the receiving object on the expression of the RHS is the input. For snippets that contain multiple statements, as with Example 1 in Section 3.1, the statements are in-lined to form a single assignment statement. To illustrate, the snippet in Example 1 becomes the following.

```
int len = s.substring(s.lastIndexOf("."), s.length()).length();
```

After a search, 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.

4.2.2. Yahoo! Pipes Mashups. We support the subset of the Yahoo! Pipes grammar shown in Figure 8. As Yahoo! Pipes is a visual language, we transform each program into a parallel-serial graph [Stolee et al. 2012] for recognition by the grammar. To illustrate, the program in Figure 2(a) would be represented as `output tail truncate fetch`, and the program in Figure 2(b) would be represented as `output filter union (fetch) (fetch)`. Our encoding supports the following modules: `fetch`, `filter`, `output`, `sort`, `split`, `tail`, `truncate`, and `union`, representing six of the top 10 most commonly used constructs. Encoding this language fragment requires evaluating substring and equality relations over strings, and enumeration over all elements in a list; as with Java, we consider bounded strings and additionally bound the lists. In the Yahoo! Pipes language, specific modules are associated with inputs (e.g., *Fetch* modules) and the output (the *Output* module), so binding the input/output to programs is straightforward. To reduce 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. These refactorings focus on decreasing the size of the pipes and standardizing them according to the community standards, and the programs were refactored using a tool developed as part of our previous work [Stolee and Elbaum 2011]. This is not a necessary step for the encoding process though it may have led to performance gains. since the sizes of the programs (and thus the number of constraints) are smaller. Measuring such gains is left for future work.

4.2.3. SQL Select Statements. We can encode SQL select statements according to the grammar in Figure 9. Our encoding supports SQL queries with the `distinct` function and with `limit, order by`, and `where` clauses, covering three of the seven most common

⟨*select_statement*⟩ ::= 'SELECT' ⟨*modifier*⟩? ⟨*column_ref*⟩ 'FROM' ⟨*table_ref*⟩
⟨*where_clause*⟩ ⟨*orderby_clause*⟩? ⟨*limit_clause*⟩? ';'

⟨*modifier*⟩ ::= 'DISTINCT'

⟨*column_ref*⟩ ::= ⟨*column*⟩ (, ⟨*column*⟩)*
| ASTERISK

⟨*column*⟩ ::= ⟨*table_ident*⟩ DOT ASTERISK
| ⟨*table_ident*⟩ DOT ⟨*column_ident*⟩
| ⟨*column_ident*⟩

⟨*table_ref*⟩ ::= ⟨*table_ident*⟩ (',' ⟨*table_ident*⟩)*

⟨*where_clause*⟩ ::= 'WHERE' ⟨*factor*⟩ ⟨*op*⟩ ⟨*factor*⟩ (⟨*andor*⟩ ⟨*factor*⟩ ⟨*op*⟩ ⟨*factor*⟩)*

⟨*factor*⟩ ::= ⟨*column_ref*⟩ | ⟨*integer*⟩

⟨*op*⟩ ::= '<' | '>' | '<=' | '>=' | '=' | '!='

⟨*andor*⟩ ::= 'AND' | 'OR'

⟨*orderby_clause*⟩ ::= 'ORDER BY' ⟨*column_ident*⟩ ('ASC' | 'DESC')?

⟨*limit_clause*⟩ ::= 'LIMIT' ⟨*integer*⟩

Fig. 9. SQL supported grammar.

MySQL select clauses.[11] Joins are also supported, but are implicit and can occur when the user specifies multiple input tables, as illustrated in Section 3.3, Example 8.

For SQL select statements, the program inputs are tables and the output is a table. We consider bounded table sizes in terms of the number of rows, similar to the bounded string and list sizes in the other domains. During encoding, the table names and column names used in the select statements are assigned symbolic names. For example, consider the following SQL query.

```
SELECT deduction, person FROM discounts ORDER BY deduction;
```

The table `discounts` is assigned a symbolic name, `sym_tbl1`, and the columns `deduction` and `person` are also assigned symbolic names, `sym_col1` and `sym_col2`. This produces the following, more general select statement.

```
SELECT sym_col1, sym_col2 FROM sym_tbl1 ORDER BY sym_col1;
```

During the search, the table name(s) and column name(s) from the input table(s) are bound to symbolic names in each encoded select statement. For example, given an input table `Payroll` with fields `[name, salary]`, these would be bound to the symbolic names as follows.

```
c1. (assert (sym_tbl1 = Payroll))
c2. (assert ((sym_col1 = name ∧ sym_col2 = salary) ∨ (sym_col1 = salary ∧
    sym_col2 = name)))
```

Constraint $c1$ binds the input table name, `Payroll`, to the symbolic table name. Constraint $c2$ binds the input column names to the symbolic column names. There are two possible bindings for the columns, which requires a disjunction. The output table in the specification binds to the result of the select statement. This allows the SQL query to be an eligible result for any arbitrary input/output example, similar to how URLs are abstracted away from Yahoo! Pipes programs.

[11]The other four clauses are `group by`, `having`, `procedure`, and `into`, per the reference: `http://dev.mysql.com/doc/refman/5.0/en/select.html`.

length restrictions on s, t	$(\text{length}(s) \geq \text{length}(t))$
value of s ends with t	$(\exists j$ $((j \geq (\text{length}(s) - \text{length}(t))) \wedge (j < \text{length}(s))$ $\wedge(\forall i$ $(((i < \text{length}(s)) \wedge (i \geq j))$ $\rightarrow (\text{charAt}(s, i) = \text{charAt}(t, (i - j)))))))$

Fig. 10. Transformation Rules for `java.lang.String.endsWith(t)`.

Table IV. Basic Operations for Current Implementation

Term	Java Strings	Yahoo! Pipes	SQL Select
Accessor	*charAt*: S x I ↦ C		value: Row x Col ↦ I
	indexOf: S x S x I ↦ I	field: R x S ↦ I | S	getCol: T x S ↦ Col
	lastindexOf: S x S x I ↦ I	recordOf: L x I ↦ R	index: T x R ↦ I
Join	*concat*: S x S ↦ S	*union*: L x L ↦ L	join: T x T x Col ↦ T
Filtering	*substring*: S x I x I ↦ S	*truncate*: L x I ↦ L	*limit*: T x I ↦ T
		tail: L x I ↦ L	*where*: T x Col x Op ↦ T
		filter: L x S x Op ↦ L	*distinct*: T x Col ↦ T
Copy		*split*: L ↦ L x L	
Permute		*sort*: L x S ↦ L	*order by*: T x Col ↦ T
Size	*length*: S ↦ I		height: T ↦ I
		size: L ↦ I	
Operators (Op)	<, ≤, >, ≥: I x I ↦ B	equals: I x I ↦ B	equals: B x B ↦ B
	contains: S x S ↦ B	equals: C x C ↦ B	equals: T x T ↦ B
	equals: S x S ↦ B		
	startsWith: S x S ↦ B	equals: L x L ↦ B	containsRow: T x Row ↦ B
	endsWith: S x S ↦ B	equals: R x R ↦ B	
		hasRec: L x R ↦ B	containsCol: T x Col ↦ B

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

4.2.4. Language Mapping. The effort to map a programming language to constraints involves several steps. These include determining which parts of the language grammar are worth supporting, mapping those grammar elements of interest to constraints, and defining the input/output model for the domain. For example, in the Java implementation, we chose to focus on the `java.lang.String` library, which is among the most common Java libraries. One method in this library that we support, $s.endsWith(t)$, is mapped to constraints by analyzing the API semantics and representing these semantics in first-order-logic, as shown in Figure 10. If, in fact, s ends with t, then these constraints will be satisfied. The converse is also true. The input/output model is defined as variables and their values with the types supported by the grammar; for our Java implementation, these are booleans, characters, integers, and Strings.

One thing to note is that many of the data structures and libraries are commonly found across many programming languages, so some of the effort can be harnessed for multiple language implementations. For example, string manipulation, list/array manipulation, and arithmetic operations are quite common across many programming languages. Table IV provides a classification of the operations supported by the current implementation. Using these basic datatypes, 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 IV, followed by a mapping to the supported language subsets.

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 is also used by Yahoo! Pipes. The `operator` functions all return booleans based on some criteria, such as two booleans being `equal` (supported in all languages), two strings being `equal` (supported in Java and Yahoo! Pipes), or determining if one string `startsWith` another (supported in Java). The `concat` method in Java joins strings, like the `union` module in Yahoo! Pipes, joins lists and the implicit join in SQL joins tables. The `sort` module in Yahoo! Pipes reorders list elements like the `order by` clause does in SQL.

Building support for a language can be incremental, as we have done it. Less support means fewer matches in the search, but growth can be incremental according to a community's needs. In our current implementation, 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. As mentioned in Section 3.1, constructs that modify the control flow, such as loops and predicates, are not part of the current implementation (Section 4.2.1). Symbolic execution [Clarke 1976; Clarke and Richardson 1985; King 1976] seems promising for handling such constructs as a means of identifying distinct paths through a program for encoding; this is left for future work.

4.3. Effectiveness and Performance

Several factors can influence search efficiency and effectiveness in terms of precision and recall. In our approach, the primary factors include the solver speed and supported theories, query complexity, repository size and complexity, and the developer's context. We explore each in turn.

4.3.1. Solver Speed and Sophistication. The performance of our approach is bound in part by the performance of SMT solvers and supported theories. A slow solver will result in slow performance, directly impacting usability. Our current implementation uses the Z3 SMT solver [De Moura and Bjørner 2008] and the UFNIA: Nonlinear integer arithmetic with uninterpreted sort and function symbols theory in the encoding of all programs, which requires strings to be represented as composite datatypes with two properties, value and length. Recent research has adapted the Z3 SMT solver to support part of the theory of strings, treating strings as primitives [Zheng et al. 2013], which may increase solver performance in the presence of string constraints.

Although we performed the search serially in our studies, we have designed this search approach to be highly parallelizable, where several solver invocations could happen in parallel. Further performance improvements may be possible by caching and reusing duplicate constraints [Visser et al. 2012]. We could also improve performance by setting a maximum solver time, forcing the solver to return *unknown* in some cases. Treating the *unknown* programs as results sacrifices precision as there may be some false positives; ignoring those programs sacrifices recall as there may be some missed matches. We evaluate our approach performance in a Yahoo! Pipes program with complex specifications in Section 5.3 and in SQL with large specifications in Section 5.4.

4.3.2. Query Complexity. Our approach supports multiple input/output pairs in a specification, as illustrated in Section 3.1 and defined in Section 4.1.1. The size of the query input or the query output, such as the length of a string or the number of

input/output pairs, all impact efficiency. As with any search approach, the quality of the query impacts the quality of the results. Redundant queries can lead to slow performance since each input/output pair needs to be checked. A good query will use examples that are succinct but illustrate the behavior of the desired code, potentially including edge cases. Three of the queries in the Java evaluation contain multiple input/output pairs. We briefly explore the impact of the number of input/output pairs on the number and quality of search results in Section 5.1.2.

The performance of the approach can be controlled to some extent at the cost of precision by bounding the sizes of the encoded data structures representing variables in the programs or the input/output pairs. For example, the lengths of strings in Java and Yahoo! Pipes, lengths of lists in Yahoo! Pipes, and number of rows in tables in SQL have a configurable maximum bound. In Section 5.3.3 with Figure 11, we explore the impact of various specification sizes on the search precision in the Yahoo! Pipes domain; in Section 5.4.3, we explore the impact of specifications sizes on the search performance in SQL.

4.3.3. Repository Size, Complexity, and Abstraction. The content of the repository has a profound impact on the efficiency and effectiveness of any search approach. A small repository may not have diverse-enough code to meet the needs of a user query, while a large repository may lead to long search times and relevant code may not be found efficiently. For encoded programs, those with higher complexity may take longer for the solver to process whereas programs with lower complexity may be too trivial and not worth searching for. Programs encoded at the most concrete abstraction level may be too specific, but more abstract programs may require too many modifications to be useful. These factors will each require thorough experimentation to measure the sensitivity of the approach to changes in each dimension.

Presently, our encoding supports strings, lists, tables, booleans, characters, and integers, so the approach implementation is limited in this way. Previous work on symbolic execution indicates that, as long as the data structures can be modeled, then their symbolic analysis is feasible, although more costly. The main challenge we foresee is with objects that live on the heap. Other challenges include handling of predicates (as mentioned in Example 3 in Section 3.1) and loops.

4.3.4. Developer Context. This search approach is particularly useful and effective when the programmer has a concrete idea of what they want the code to do and can illustrate this with an example. Such a situation may manifest during general development activities, but may be particularly common during test-driven development where the programmer creates stubs and test cases for their desired code. Using the test cases as input/output examples, the code search would identify potential code candidates to fill in the stubs. In this way, the search would operate behind the scenes and the developer could continue designing and developing code as the search finds candidate source code.

As with the preceding scenario, depending on how the approach is used, performance issues may not be a problem. Specifically, the matched results will behave as specified, which is not the case for most matched results returned by state-of-the-practice syntactic searches. Thus, programmers may be inclined to trade speed for quality. For example, novice programmers may know what their desired code should do but not how to code it. A more experienced programmer who is new to a language may run into the same situation while learning new syntax and libraries. In these cases, it may be useful to provide an example of the desired behavior and see how, in the new language, such behavior can be achieved.

While these situations provide scenarios when slower search performance may be tolerated, studying this tolerance is left for future work.

4.4. Summary

In this section, we have defined and provided the implementations details for our code search approach in subsets of three languages, namely, Java, Yahoo! Pipes, and SQL, providing the grammars and describing the language coverage. We have also identified challenges and opportunities for this approach as the research moves forward. Next, we evaluate each language by manipulating several of the factors and exploring the impact on effectiveness and performance.

5. EVALUATION

The study is designed to provide a preliminary assessment of the approach across multiple dimensions while highlighting some key aspects in the three supported domains: Java, Yahoo! Pipes, and SQL. It is not exhaustive, but rather is designed to explore the potential of this approach to serve the diverse needs of programmers across many domains and outline potential areas of future exploration. Comparing our search to state-of-the-art and state-of-the-practice searches is difficult since the query models are heterogeneous across approaches (e.g., keyword, formal specification, input/output example, etc.) and the content of the repositories may vary significantly. We took a mixed approach to mitigate these challenges, using the opportunities provided by each domain to explore various aspects of the approach more fully. Thus, the evaluations are different for each domain.

To evaluate our approach in Java, we begin by comparing our approach to the state-of-the-practice code searches by searching a local repository using our search and a Google-powered keyword-based search engine pointed to a local repository. The relevance of the search results was judged by programmers in an empirical study. This evaluation is designed to address our first research question:

RQ1. How do our search results compare to those found using a keyword-based approach, from the perspective of the programmer?

While *RQ1* aims to compare our approach to syntactic searches, we hypothesized that these search approaches are complimentary and can be used together. The goal of *RQ2* is to explore the benefits of combining the two search approaches in Java; we evaluate how Google and our approach can work together by evaluating our second research question:

RQ2. How much can existing search approaches be improved by augmenting results with our search approach?

As discussed in Section 3.2, programmers can search the Yahoo! Pipes repository by URL, which is also the input used by our search approach. However, our search includes another piece of information, the output. In order to obtain specifications to evaluate our approach in this domain, we identify representative pipes from which we extract input/output queries. We first explore the shortcomings of existing search techniques to identify the potential for gains. Second, we use our approach to search a local repository for relevant pipes and explore the impact of tweaking the search parameters on the effectiveness of the search. This evaluation aims to address our third research question:

RQ3. What is the impact of tweaking search parameters, specifically abstraction, specification size, and solver time, on the search effectiveness?

The Yahoo! Pipes domain is better suited than Java to evaluate *RQ3* for two reasons. First, in our current Java evaluation, the specifications and snippets are small, so the solver times are fast. In Yahoo! Pipes, the specifications can get quite large and solving can take minutes, leaving much opportunity for improvement through specification

manipulation (e.g., considering partial specifications, trading precision for efficiency). Second, the Java code snippets we encode are taken out of context, so many variables are already symbolic. The Yahoo! Pipes programs, on the other hand, are encoded in their entirety, which leaves more opportunity to gain from abstraction.

Scalability is a concern as the size of a specification can have a big impact on the search time. Increasing the bounds for the specification sizes (i.e., strings, lists, tables) can give an idea of how our approach scales with respect to specification size. We manipulate the size and content of specifications in SQL to better understand the impact of specification size and complexity on search time. This evaluation aims to address our fourth research question:

RQ4. What is the impact of query complexity on search time?

For each research question, we describe how the repositories were built, the metrics we use, and the results. All of the study artifacts are available online.[12] For the studies related to *RQ3* and *RQ4*, our data were collected under Linux on a 2.4 GHz Opteron 250s with 16GB of RAM. For *RQ1* and *RQ2*, our data were collected under OS X on a 2.4 GHz Intel Core 2 Duo with 4GB of RAM.

5.1. RQ1: Comparing Our Search to Syntactic Searches – Java

RQ1 aims to compare a keyword-based search approach against our approach in Java. We use a local repository that we created and control as a common baseline to compare the results obtained by our search against the results obtained by a Google-powered syntactic search engine pointed at the same local repository. Performing a similar comparison of our approach to a general Google search is impractical, since it would require us to index the same scope of programs and Web pages that Google has indexed and our encodings are limited. Instead, we opt to compare the results of a syntactic approach with our approach using a common baseline repository.

5.1.1. Metrics. To compare the results across the search techniques on the local repository, we use the number of results and *P@10*, which represents the precision, or relevance, of the top 10 results. To calculate relevance, we performed an empirical study where each of the top 10 results was shown to programmers who determined whether or not the code was relevant to the problem. The average relevance among the top 10 search results forms the P@10 metric.

5.1.2. Artifacts. Comparing the search approaches requires the same query and the same repository so the results can be compared. We could control the space of potential matches by pointing both search approaches at the same repository. The following describes how we formed the repository, gathered queries, and obtained search results for evaluation.

Local repository. We built a local repository by issuing syntactic searches on Koders.com [Koders 2012] for each of the `java.lang.String` functions supported by our encoding. We scraped all lines of Java source code that contained a call to at least one of the supported functions, totaling 5,192 lines. We pruned out duplicates, lines that contained functions we do not support, and those that are not assignment or return statements, per the grammar in Figure 7. This left 713 unique snippets of code that form the Java code repository used in this evaluation. By making this repository available online, we were able to create a custom Google search engine that points to the local repository, which was used as the keyword-based search approach.

[12]https://sites.google.com/site/semanticcodesearch.

Table V. Java Artifacts Specifications for *RQ1* and *RQ2*

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

Search queries. Our search and a keyword-based search use different query models, input/output examples, and textual queries, respectively, so we found a resource that would allow us to extract both input/output examples and keywords to perform the searches. Using questions posted on stackoverflow, we use the posting title as the keyword query and the input/output example(s) as the query for our approach. Of the 67 questions tagged in stackoverflow with `java`, `string`, and `substring`, 40 (60%) contain some form of explicit example. For 17 of these cases, our current Java implementation supports encoding the input/output example. The remaining 23 involve constructs we do not currently support, such as regular expressions or arrays. The titles and input/output for the 17 questions are shown in Table V. The *Q* column identifies the question number, and *Title* is as it appears in stackoverflow. Each of these questions has an input and output example which are shown in the *Input String* and *Output String* columns. For *Q1*, *Q9*, and *Q13*, multiple examples were provided.

Search results. Results for the keyword-based approach were obtained by issuing each title from Table V as a keyword query against the local repository, using a Google-powered keyword-based search engine. The top 10 results were retained for evaluation.

For our approach, we encoded the input/output as C_{LS} for each of the 17 stackoverflow questions and searched our local repository for matches. The top 10 results were

retained for evaluation. When multiple input/output examples were given, as was the case with *Q1*, *Q9*, and *Q13*, the examples were considered simultaneously to identify results. This means that if there exists a free variable in a solution, such as the upper bound on a substring, that variable needs to be set to the same value for all examples in order for the result to be considered. For example, with the first specification in *Q1*, the input is "Animal.dog" output is "Animal". The following code is among the 51 code snippets identified as a match.

```
String fieldname = line.substring(0, idx);
```

With the input bound to *line* and the output bound to *fieldname*, the variable *idx* is symbolic. The solver determines this example matches when $idx \mapsto 6$. However, considering the second specification, "World.game" and "World", this match is eliminated since $idx \mapsto 5$. Other results set the upper bound based on a property of the input variable. For example, the following code matches both specifications since the upper bound is set to be the index of the string "." in the input variable, *typel*.

```
packagename = typel.substring(0, typel.lastindexof(''.''));
```

5.1.3. Human Evaluation. For each of the top 10 results returned by either search approach, we asked programmers if the source code was relevant to the programming task described by the original stackoverflow title. The following describes the experimental setup and implementation.

Experimental tasks. An experimental task presents a participant with a programming task (i.e., the title from Table V) and five source-code snippets. Only the titles were presented since the descriptions also contained the input/output examples and we did not want to bias the participants against results that did not match the example but may still be relevant to the task (that is, results from a keyword search). The snippets are the search results from the keyword-based search and the input/output search, alternating. For half of the tasks, a result from the keyword-based search appeared first; the other half had results from the input/output search appearing first. Participants were not made aware of which search approach was responsible for which snippet. Then, participants state whether or not each code snippet is relevant to the task (yes/no response) and why (free response). Relevance was defined by *"source code* [that] *is directly applicable except for variable renaming or resetting."* For example, for *Q1*, the following snippet was determined to be relevant if the variable, *querystring*, is set to ".".

```
url = url.substring(0, url.indexof(querystring));
```

We created 64 different tasks; this is calculated by the 17 questions * 10 search results * 2 search approaches = 340 snippets. For *Q6* and *Q10*, our approach returned zero results, reducing this to 320 snippets. With five snippets per experimental task, there were 320/5 = 64 experimental tasks available. When there were fewer than 10 responses (e.g., *Q1* and our approach, *Q5* and the keyword-based approach), the search results were repeated, starting with the first result, to fill up 10 slots; otherwise, each search result appeared exactly once. This maintained the alternating pattern of responses in the experimental task design.

Deployment. As with part of the survey in Section 2.1, this study was deployed on Amazon's Mechanical Turk [2010]. This platform has been effective for gathering programmer opinions regarding source code in our previous projects [Stolee and Elbaum 2010]. Each experimental task is implemented as a human intelligence task, or HIT. In order to perform HITs in the study, participants had to correctly answer at least two of four multiple-choice Java competency questions correctly (details available

[Stolee 2013, Appendix C]). These questions required the potential participants to read and analyze the behavior of Java methods.

Each HIT paid $0.50 and participants could complete all 64. For replication, five participants performed each HIT. The study was available for one month from September 22, 2013 until October 22, 2013.

Subjects. Our study involved 19 participants. The average participant had over four years of Java experience; 68% of the participants reported to program daily while the remaining six participants programmed less frequently. Approximately half the participants reported to search for code daily or *"whenever* [they] *code"*.

The median number of HITs completed per participant was 7, with a maximum of 62 and a minimum of 1. While the quantity of HITs performed by a couple of participants was high, the impact on the overall P@10 was minimal. For example, removing the responses from the participant who completed 62 HITs had a -0.017 impact on P@10 for our approach and a +0.003 impact for the keyword-based approach.

Each HIT took participants approximately 4.5 minutes to complete for an effective hourly rate of $6.52.

5.1.4. Results. The results for both *RQ1* and *RQ2* (see Section 5.2.3) are shown in Table VI. The *Q* column matches the specifications shown in Table V. The next sets of columns, *Our Approach* and *Keyword Approach*, show results for *RQ1*.

On average, our approach found 20.5 matches for each query, ranging from zero (in two cases, *Q6* and *Q10*) to 49 results. As an example, for *Q2* in Table V, given the input "23 14 this is random" and output "this is random", our search approach finds 24 matches including, for example, the following.

```
String message = name.substring(6);
```

Although queries can be refined by adding extra examples, some refinements are better than others. Adding the second input/output pair reduces the number of results for *Q*1 from 51 to four. For *Q*9 and *Q*13, there is no reduction in the search results when considering all the input/output pairs in the specification compared to considering just the first input/output pair. This may be because the examples were too similar, or because the results actually capture the intention of the programmer. Specifically, for *Q*9, at least one result matched the community-approved result on stackoverflow.

Using the keyword-based search, on average, 48.5 matches were found for each query, ranging from two to 100. These results are under the *Keyword Approach* column in Table VI. In *Q2*, for example, we see that the keyword-based approach returns 34 results.

Comparing all results to the solutions proposed and positively voted by the stackoverflow community, our approach returns results that match the community solutions for five of the 17 searches (*Q4*, *Q9*, *Q11*, *Q12*, and *Q14*, each marked with the *), and keyword-based results matched for two searches (*Q4* and *Q9*). A match was determined if all API calls were the same between two snippets.[13] For example, consider *Q9*. Our search returns 49 snippets, including relevant snippets `s1` and `s2`.

```
s1. repository = location.substring(0, colon_index);
s2. String fieldname = line.substring(0, idx);
```

The keyword-based approach returned 41 results, including relevant snippets `s3` and `s4`.

[13]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 our result sets.

Table VI. Java Results for *RQ1* where P@10 is Based on an Assessment Using 19 Programmers and for *RQ2* where Our Approach is Combined with Google

	RQ1				*RQ2*			
	Our Approach		Keyword Approach		Google Global + Us			
Q	#	P@10	#	P@10	S@10	Discarded	S'@10	% Reduction
1	4	1.00	99	0.28	25	18	7	72%
2	24	0.84	34	0.36	17	0	17	0%
3	48	0.90	37	0.30	0	0	0	–
4	13	*0.88	100	*0.28	36	12	+24	33%
5	48	0.98	5	0.14	3	0	3	0%
6	0	0.00	99	0.42	37	6	31	16%
7	21	0.82	42	0.28	16	4	12	25%
8	20	0.54	99	0.26	38	7	31	18%
9	49	*0.86	41	*0.20	0	0	0	–
10	0	0.00	40	0.42	9	2	7	22%
11	23	*0.72	70	0.42	6	3	3	50%
12	13	*0.92	38	0.22	7	2	+5	29%
13	24	0.86	2	0.04	29	11	17	38%
14	22	*0.76	38	0.24	0	0	0	–
15	13	0.96	42	0.26	0	0	0	–
16	14	0.94	2	0.08	26	14	12	54%
17	13	0.90	34	0.16	8	7	1	88%
Average	**20.5**	**0.76**	**48.4**	**0.26**	**15**	**5**	**10**	**34%**

Key:
#: The number of results from the search.
P@10: Relevant results from the search (according to programmers).
(* indicates some results match Stackoverflow responses).
S@10: Count of 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.
Reduction: The reduction in snippets that need to be evaluated.
(+ indicates a results returned *sat*).

```
s3. String = string.substring(0, end);
s4. String axispart = mdxquery.substring(mdxquery.indexOf(select), mdxquery.
    indexOf(from));
```

Stackoverflow suggests snippet `s5` as a result.

```
s5. String.substring(0, maxLength);
```

While `s4` could be instantiated to fit the specification in *Q9*, snippets `s1`, `s2`, and `s3` match the API calls used in `s5`.

The ultimate oracle for the relevance, however, is a human judge. For this reason, we also turned to programmers to determine the relevance of the search results with respect to the problems and to calculate P@10.

The average relevance among the top 10 search results for our approach was 0.76 versus 0.26 for the keyword-based approach. The breakdown per question is shown in Table VI. For our approach, the best results came from *Q1* where all the results were found to be relevant (P@10 = 1.00). This may have been due, in part, to the fact that multiple input/output examples were given, leading to highly relevant results. For the other questions that had multiple input/output pairs, *Q9* and *Q13*, the relevance was slightly lower at 0.86 each. For the keyword-based approach, the highest relevance

came from *Q6*, *Q10*, and *Q11* with P@10 = 0.42 for each. While the keyword-based results' relevance was always lower, this demonstrates a complementary nature among the search approaches. For two of the highest-performing questions for the keyword-based approach, our approach was not able to find any search results (i.e., *Q6* and *Q10*).

In summary, we observed that, using the same repository, the keyword-based approach returns over twice as many results as our approach, but among the top ten, our approach is nearly three times more effective at returning relevant results. For four of the 17 searches (*Q5*, *Q8*, *Q13*, and *Q16*), our approach provides matches when the keyword-based approach does not find any as the syntactic query was not good enough to identify results.

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 VI and all searches, the average solver time to determine *sat* is 0.0483 seconds and to determine *unsat* is 0.0051 seconds. However, given that the snippets and the specifications are small, this may represent a best-case scenario. In the presence of larger and more complex programs and larger and more complex specifications, these performance measures drop, as we observe with *RQ3* and *RQ4*.

5.2. RQ2: Combining Our Search with Google – Java

Rather than treating our search approach as an alternate to a keyword-based search engine, we hypothesized that these two approaches are complementary. For *RQ2*, we perform Google searches on the Web and explore how the results could be filtered and improved by also using our search approach.

5.2.1. Metrics. A syntactic search returns many Web pages that could contain several snippets of code. To capture the space of code that must be evaluated by a programmer, we define new metrics, *S@10* and *S'@10*. The metric *S@10* represents the number of code snippets returned in the top ten results from a general Google search.

To capture *S@10*, we issue Google queries, then scrape and count the Java code snippets from the top 10 page results. The metric *S'@10* represents the number of snippets the programmer must evaluate after applying our search technique on top of the Google results. To capture *S'@10*, we first attempt to encode all the snippets in *S@10*. Next, we run our search technique using the encoded *S@10* snippets as a repository and the example input/output as a specification, and discard snippets that return *unsat*. This set of *discarded* snippets represents those that the programmer does not need to evaluate by hand. *S'@10* is calculated as the difference (*S@10* − *Discarded*), representing the reduced space of snippets for the user to evaluate.

5.2.2. Artifacts. We use the same artifacts gathered for *RQ1*, shown in Table V. The initial queries to Google were formed using the titles. For each of the top 10 page results, we collected the source-code snippets. This formed a temporary repository for the input/output search, which used the input/output examples from Table V as the query.

5.2.3. Results. The results for *RQ2* are shown in Table VI in the last set of columns, *Global Google + Us*. On average, 15 snippets were gathered from the top 10 search results per search, with a range from zero to 38 (zero occurred when none of the retrieved pages was in the Java language).

By trying to encode each of these snippets, we were able to check the input/output pairs from Table V against the retrieved snippets as a way to identify matches and prune the result set. The number of snippets for which the SMT solver returns *unsat*

given the input/output specification is shown in the *Discarded* column. 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, though the search time would clearly increase. In two cases, *Q4* and *Q12* (marked with + in Table VI), at least one snippet returned *sat*, indicating that the snippet matches the specification and would be a solution. 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.

While our approach can assist syntactic searches by removing irrelevant results, it should be noted that if a syntactic query misses a possible solution (i.e., a snippet of code that would provide a solution is not in the Google result set), then our search would not have the opportunity to evaluate that solution. Here again, the effectiveness of the search is dependent on the programmer's ability to write a query tied to documentation or syntax, a limitation that is addressed when our search is used in isolation, as was done for *RQ1*. Still, integrating syntactic search capabilities may be useful for programmers who know a little about the implementation they desire, though clearly the programmer would sacrifice some performance over just a syntactic search.

5.3. RQ3: Impact of Tweaking Search Parameters – Yahoo! Pipes

In Yahoo! Pipes, the specifications can be quite large and complex, and tweaking the search parameters can have a profound impact on the results. We begin by looking at the effectiveness of the state-of-the-practice search approach, to show when the existing search succeeds and when it fails. Then the results for *RQ3* are presented, exploring the impact of tweaking search parameters, specifically solver time, specification size, and the abstraction level of program encodings, on the effectiveness of the search in Yahoo! Pipes. We measure effectiveness using precision and recall, where the baseline is our search at the most abstract encodings. This is different from the previous study because we are not comparing our results to another search engine, but rather are comparing against an oracle.

5.3.1. Artifacts. To evaluate *RQ3*, we require a repository of Yahoo! Pipes programs and specifications to search the repository. In a previous study with Yahoo! Pipes [Stolee et al. 2012] we scraped 32,887 pipes programs from the public repository by issuing approximately 50 queries against the repository and removing all duplicates. Among these pipes, 2,859 are supported by our encoding (Section 4.2.2), which forms the local repository.

To perform the searches for the study, we gathered specifications from five representative pipes in the repository. These pipes were identified as follows: the pipes were clustered based on their structural similarity (i.e., modules and wires match in topology, but the field values within modules can differ). The clusters were ordered according to size and one pipe was selected from each of the median five clusters. Each specification was obtained by retrieving the RSS feeds to form the input list(s) and executing the pipe to capture the output list.

The pipes used to generate the specifications are described in the *Structure* column in Table VII. The first pipe has one URL that gathers weather information. The specification retrieved from this pipe is *specification 1*. The second pipe has one URL and retains records that contain "hotel" in the description field, then sorts the list and retains the first three records. The retrieved specification is *specification 2*. The third pipe has three URLs and forms *specification 3*. The fourth pipe has one URL and is similar to the pipe shown in Figure 2(a); the retrieved specification is *specification 4*. The fifth pipe creates *specification 5*. It aggregates and sorts the items from two URLs.

Table VII. State-of-the-Practice Search by URL Query on Global and Local Repositories

Pipe	Structure	URLs	Global Search Matches	Global Search P@10	Local Search Matches	Local Search P@10
1	`output union ((filter) (filter)) split fetch`	rss.weather.com	71	0.2	1	0.1
2	`output truncate sort filter fetch`	feeds.feedburner.com	16,990	0.0	881	0.0
3	`output sort union (truncate fetch) (truncate fetch) (truncate fetch)`	anunturi-gratis.ro anunturi-utile.ro feedproxy.google.com	1,281	0.0	220	0.0
4	`output tail truncate fetch`	ocregister.com	38	0.0	1	0.1
5	`output sort union (fetch filter)(fetch)`	feeds.gawker.com lifehacker.com.au	4	0.1	1	0.1

5.3.2. Metrics. To explore the effectiveness of the state-of-the-practice search, we searched repositories of Yahoo! Pipes programs using the URLs from each of the derived specifications. We report the number of *matches* returned by the search, and *P@10*, which is determined by executing each pipe and evaluating the results. This search is preformed on the global Yahoo! Pipes repository and on our local repository.

For the input/output search, we manipulate three search parameters, namely the abstraction of the program encodings, the specification size, and the maximum solver time. We report the number of pipes in the local repository that return *sat*, *unsat*, and *unknown* (?) at each of four solver times, 1, 10, 100, and 1,000 seconds (*sec.*), considering four sizes of specification and two levels of abstraction on the program encoding. The specification sizes are measured as a percentage of the full specification from which the precision and recall are computed. The levels considered are 25%, 50%, 75%, and 100%. For example, if a specification has 10 RSS items in the input, as is the case with specifications 1, 2, and 5, then 75% of the input size would consider the first 8 items, and 50% would consider the first 5 items. The output is adjusted according to the input. That is, the ninth item in specification 2 is included in the output, but when considering the first 75% of the specification, this item is dropped also from the output. Abstraction has two levels, all concrete and all symbolic, on the string and integer fields. In the *symbolic* encoding, all configurable string and integer fields in the operator modules (`<operator>` in Figure 9) are relaxed.

We also calculate precision and recall, where

$$precision = \frac{relevant \cap sat}{sat} \quad \text{and} \quad recall = \frac{relevant \cap sat}{relevant}.$$

Using the results of our own search as a baseline, relevant results are those that will eventually (given infinite time) return *sat* with a symbolic encoding, which represents the pipes for which an instantiation of the module field values can achieve the desired behavior.

5.3.3. Results. To explore the impact of tweaking search parameters, we use our approach to search our local repository using each of the five input/output specifications, given the solver times, specification sizes, and abstraction levels described. These search parameters are relevant to our semantic approach only, so we cannot directly compare our results to a syntactic search on the local repository. To gain a better understanding of how programmers currently search in this domain and the potential

for improvement with our proposed input/output search, we first provide the results of state-of-the-practice searches.

State-of-the-practice. The results for the state-of-the-practice searches on the global repository are shown in Table VII in the *Global Search* columns, and varied substantially among the five example specifications. For each search, the number of matches and *P@10* are reported.[14] For two of the searches, *specification 2* and *3*, thousands of matches were returned in the search. *Specification 5*, on the other hand, only returned four results, where one was relevant; two pipes were relevant for *specification 1*. Repeating this search on our local repository (*Local Search* in Table VII), which is much smaller (approximately 3,000 pipes), yields one match for *specification 1*, *specification 4*, and *specification 5*, with and $P@10 = 0.1$ (the pipe from which the specifications were generated). *Specification 2* returns 881 results and *specification 3* returns 220 results, with $P@10 = 0.0$ for both. What this illustrates is that for the more common URLs, programmers must sift through a lot of irrelevant results, and a pipe that behaves as they want might not be easy to find. In all cases, to determine relevance, each pipe must be either executed or manually inspected.

Impact of abstraction on recall. We consider two levels of abstraction: a concrete level where the programs (sans URLs) are encoded as-is and a symbolic level in which the strings and integers in the program encodings are made symbolic. The results of our experiments are shown in Table VIII and Table IX for the impact of abstraction and solver time on recall.

In Table VIII, the first set of columns reports the results for the *Concrete* abstraction level, and the second set for the *Symbolic* abstraction level. Each row represents the results given a specific maximum solver time. The number of matches is shown in the *Sat* column, and the number of discarded programs is shown in the *Unsat* column. If the solver was stopped before it could make a decision, then the solver returned *unknown*, which is shown in the *?* column, followed by the recall metric.

For all searches 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 our local repository. Symbolic encodings yield as many or more results than concrete. For instance, with *specification 4* in Table VIII 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.

Even though the concrete encodings yield fewer results, in all cases, our search on the local repository (Table VIII) returns at least as many relevant results as the syntactic searches on either the global or the local repository (Table VII). For *specification 3*, we find three results with the concrete encoding whereas neither of the syntactic searches return any relevant results among the top 10. For *specification 2*, one result is found, compared to zero relevant results in the syntactic searches. We observe that it does take some time to find results, yet, for the other three specifications, a result is returned in the concrete encoding within 100 seconds, and in the case of *specification 4*, within 10 seconds.

Based on the differences in the number of results for symbolic versus concrete encodings, the impact of abstraction on the solver time warrants further investigation. In the *Symbolic* encodings reported in Table VIII, both integers and strings were abstracted. Using the five example pipes from which the specifications were derived, we teased apart the symbolic encodings to create two additional levels of abstraction between *Concrete* and *Symbolic*; these are SICS (Symbolic Integers, Concrete Strings)

[14]Search results reflect the state of the repository on February 22, 2012.

Table VIII. Pipe Specification Search Results

Specification 1

	Concrete				Symbolic			
sec.	Sat	Unsat	?	Recall	Sat	Unsat	?	Recall
1000	17	2842	0	0.165	100	2756	3	0.971
100	16	2842	1	0.155	24	2756	79	0.233
10	0	2836	23	0.000	0	2723	136	0.000
1	0	2794	65	0.000	0	2572	287	0.000

Specification 2

	Concrete				Symbolic			
sec.	Sat	Unsat	?	Recall	Sat	Unsat	?	Recall
1000	1	2858	0	0.333	2	2856	1	0.667
100	0	2858	1	0.000	0	2853	6	0.000
10	0	2836	23	0.000	0	2785	74	0.000
1	0	2783	76	0.000	0	2567	292	0.000

Specification 3

	Concrete				Symbolic			
sec.	Sat	Unsat	?	Recall	Sat	Unsat	?	Recall
1000	3	2856	0	0.143	18	2838	3	0.857
100	0	2856	3	0.000	0	2833	26	0.000
10	0	2835	24	0.000	0	2651	208	0.000
1	0	2798	61	0.000	0	2554	305	0.000

Specification 4

	Concrete				Symbolic			
sec.	Sat	Unsat	?	Recall	Sat	Unsat	?	Recall
1000	1	2858	0	0.011	89	2770	0	1.000
100	1	2858	0	0.011	86	2770	3	0.966
10	1	2858	0	0.011	3	2770	86	0.034
1	0	2795	64	0.000	0	2758	101	0.000

Specification 5

	Concrete				Symbolic			
sec.	Sat	Unsat	?	Recall	Sat	Unsat	?	Recall
1000	1	2858	0	1.000	1	2858	0	1.000
100	1	2858	0	1.000	0	2857	2	0.000
10	0	2851	8	0.000	0	2773	86	0.000
1	0	2799	60	0.000	0	2607	252	0.000

and SSCI (Symbolic Strings, Concrete Integers). We paired the specification with its original pipe at each of the four abstraction levels and invoked the solver, measuring the runtime. Table IX presents the runtime results for the concrete encoding in the *Concrete Encoding* column, averaged over three runs. The next three columns show the *Slowdown* of the runtime for the various levels of abstraction. The final two columns of the table indicate the number of strings and integers that were *Abstracted* in each pipe.

For pipes where only integers were ever abstracted, pipe 3 and pipe 4, a speedup was observed when the integers were removed (i.e., indicated by negative slowdown in *SICS* and *Symbolic*). For the SICS encoding, a 58% speedup is observed for pipe 3, and a 27%

Table IX. Comparative Runtime in Seconds after Applying Abstraction

	Concrete	Slowdown			Abstracted	
	Encoding	SICS	SSCI	Symbolic	Strings	Integers
Pipe 1	181 sec.	−1%	+908%	+960%	2	0
Pipe 2	131 sec.	+1000%	+247%	+1763%	1	1
Pipe 3	511 sec.	−58%	+3%	−52%	0	3
Pipe 4	4.9 sec.	−27%	+0%	−31%	0	2
Pipe 5	78 sec.	+1%	+237%	+212%	1	0

speedup is observed for pipe 4. On the other hand, when strings are abstracted, which happened for pipe 1, pipe 2, and pipe 5, there is a slowdown of at least 200% in the *SSCI* and *Symbolic* columns. Combining abstracted strings with abstracted integers, which happens for pipe 2, causes the largest slowdown we observed. In terms of complexity, pipe 2 includes *filter*, *sort*, and *truncate* modules, whereas each of the other pipes contains only one or two of those module types; we hypothesize that the combination of complex modules contributes to the poor performance. Using this information will be important when optimizing the performance of the search approach in the presence of abstraction.

Impact of solver time on recall. The concrete encodings can discard irrelevant programs faster than the symbolic encodings, in part because the constraint systems are tighter and the solver has fewer decisions to make. 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 for its symbolic counterpart. Since cutting the solver time before it has reached a conclusion returns *unknown*, the recall is reduced as only *sat* pipes are considered results. Treating the *unknown* pipes as results will increase recall to 1.00, but at the cost of precision. Studying this trade-off is left as future work.

Impact of specification size on precision. Larger specifications seem to have a bimodal profile, either returning nothing because they run out of time or returning a precise match if they are allowed to run longer. Figure 11 shows the impact of modifying the specification size on precision for each of the specifications and two abstraction levels, with a range of solver times from 1 to 1000 seconds. Each combination of specification and abstraction level is presented in a graph. The x-axis shows the solver time in seconds on a base-10 logarithmic scale (i.e., 0.6 represents $10^{0.6} \approx 4.0$ seconds, 3.0 represents $10^{3.0} = 1000$ seconds). The y-axis shows the search precision. Each line in each graph represents a percentage of the specification size used in the search, either 25%, 50%, 75%, or 100%. For example, with specification 1, the concrete encoding, 100 seconds (2.0 on the x-axis), and 50% of the specification, the search precision is 0.719. In this case, 32 results are found in total, and only 23 of those are relevant based on pipe behavior.

Overall, we see that a smaller specification can yield results faster, but at the cost of precision. For example, with specification 3, a concrete encoding, and a full specification, no results are found until 600 seconds (2.78 on the x-axis). However, considering 25% of the specification yields 13 results in 6 seconds (0.8 on the x-axis), but only with a precision of 0.231. Considering 50% of the specification yields perfect precision at 20 and 30 seconds, but the precision drops to 0.600 after 40 seconds (this happens because the incomplete specification can return false positives; for this specification, and those do not appear until after 40 seconds). Considering 75% of the specification returns results within 200 seconds (2.3 on the x-axis) with 100% precision. For specification 1 and a concrete encoding, 50% of the specification reaches a plateau of precision at 0.719 after 50 seconds. Using a full specification requires a wait of 60 seconds and yields precision of 1.000. With the symbolic encoding of specification 4, 75% of the specification yields

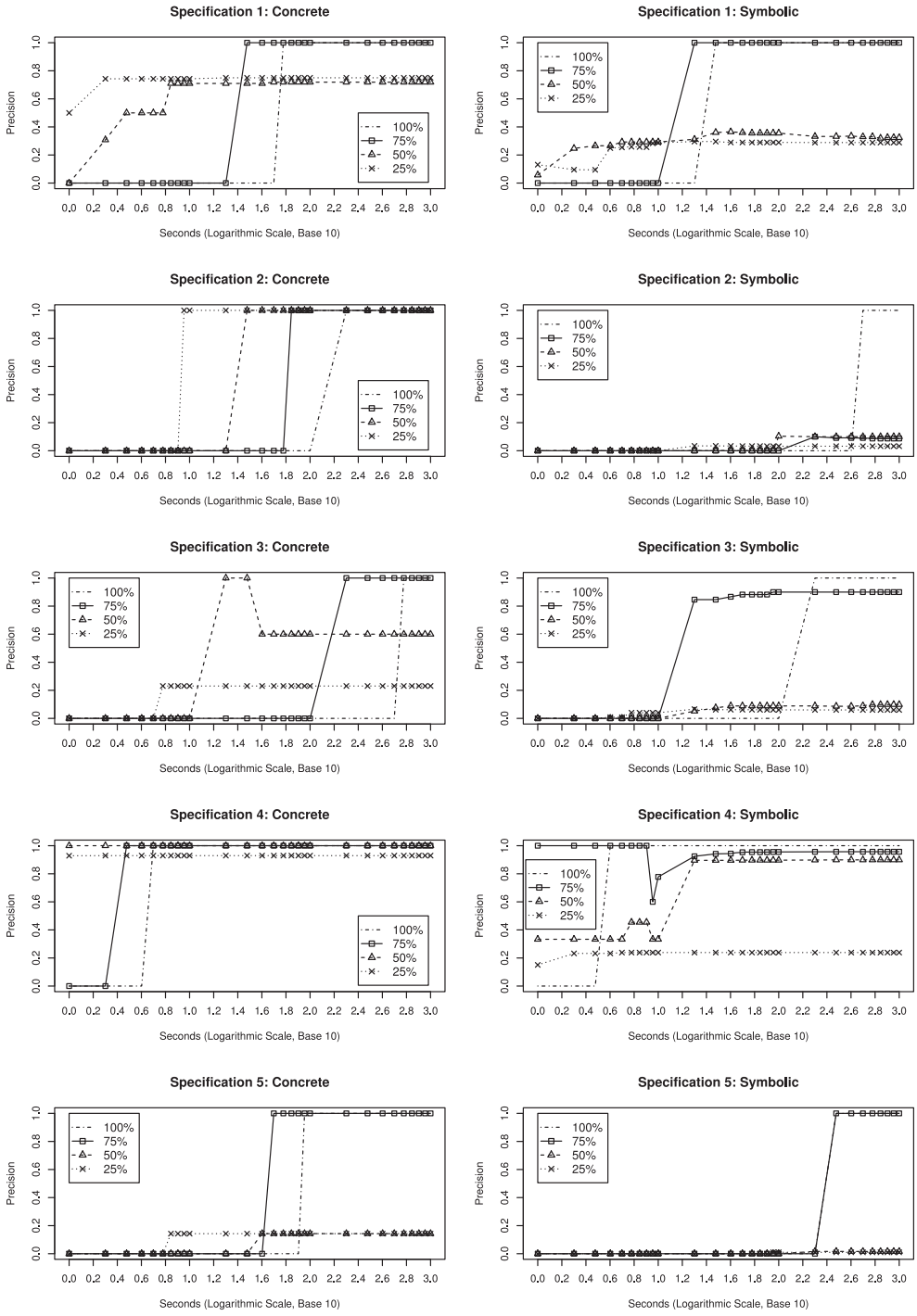


Fig. 11. Impact of specification size and solver time on precision. The x-axis represents time in seconds and the y-axis represents precision. Each line represents a specification size. The full specification is represented as 100%. A three-quarter specification is represented by the 75% line, and so forth.

precision of 1.00 within 1 second. Results are also found with 50% and 25% of the specification, but the precision is much lower; the precision for 25% never rises above 0.239. Given the potential for parallelization of the approach, it may be worthwhile to launch solvers with different maximum times in parallel with different spec lengths.

While the smaller specifications can yield results faster, care must be taken. In specification 5 and a symbolic encoding, considering 25% of the specification returns 45 results in 10 seconds, yet none is relevant and precision is 0.0. Understanding when it is appropriate to consider a reduced specification size is left for future work, but we see an opportunity to decrease the search time at the cost of precision, which could make the approach more amenable to being combined with other approaches (e.g., syntactic searches as explored in *RQ2*).

5.4. RQ4: Impact of Query Complexity on Search Time – SQL

In our search approach, we have two primary concerns with respect to scalability: increasing the size of the repositories and increasing the size and complexity of the programs and specification. In the former case, scalability may be improved by introducing parallelization, more clever heuristics, and higher-level encodings, which we leave for future work. *RQ4* explores the impact of size and complexity of specifications on the time for the solver to return *sat*, indicating a match. Since SQL tables can become very large in practice, it was the natural domain for evaluating this question.

5.4.1. Artifacts. To address *RQ4*, we required a careful manipulation of the specification to vary size and complexity. We selected a program (SQL select query) from stackoverflow[15] 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 selected the first one when ranked by number of votes that also had an input/output example.

```
SELECT username FROM table WHERE balance >= 1000000 ORDER BY balance DESC;
```

To vary specification complexity, we decompose the statement into component clauses, `where` and `order by`, and generate four SQL statements using the combinations. These statements are as follows.

```
s1. SELECT username FROM table;
s2. SELECT username FROM table WHERE balance >=1000000;
s3. SELECT username FROM table ORDER BY balance DESC;
s4. SELECT username FROM table WHERE balance >= 1000000 ORDER BY balance DESC;
```

To vary the specification size, we generate input tables with 10 to 100 rows in increments of 10, for each of the component combinations. The values of `balance` in the input tables were pulled from a normal distribution $\mathcal{N}(\mu = 1{,}000{,}000, \sigma^2 = 200{,}000)$. For each decomposed select statement and each specification size, the output tables were generated from the input table to satisfy the query and caused the solver to return *sat*.

5.4.2. Metrics. We report the time to return *sat*, averaged over ten runs, for each decomposed select statement and each specification size. On each run, a new input table was pulled from the normal distribution and a new output table was generated.

5.4.3. Results. Figure 12 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 is represented by a symbol on the graph.

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

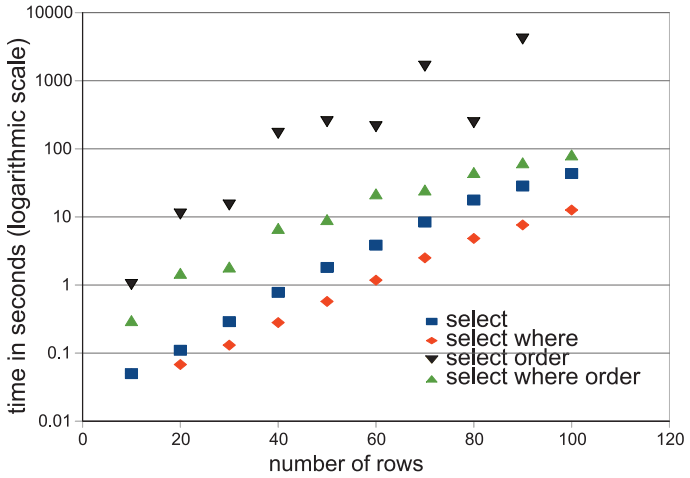


Fig. 12. Specification size versus solver time in SQL.

The solving time increases exponentially with the number of rows for all the specifications. In all cases, the output table size is 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 since the `balance` values were pulled from a normal distribution where μ was equal to the critical value in the query (i.e., 1,000,000). When the `where` clause is omitted, the sizes of the input and output tables are equal. In the graph, the performance of the `select where` statement and `select where order` statement are very close to the performance of the `select` statement. This is not entirely intuitive since the output table size for the former two is approximately half the input table size, and for the latter the table sizes are equal. The `select order` statement is the least efficient, and the input and output table sizes are the same. Regardless, there is a clear relationship between the size of the input table and solver time.

It is more subtle 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 with `select order` takes more time than the one with `select where order`, as the expensive sorting constraints from `order` 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.

5.5. Threats to Validity

Our evaluation explores different aspects of the approach in each of the three languages, and each comes with its own limitations and threats to validity.

In the Java study for *RQ1*, we show that our search approach finds more relevant results than a keyword-based search when using our local repository. In practice, however, syntactic searches thrive in large repositories in terms of finding matches at the cost of precision. By applying our technique on top of snippets gathered from general Google searches, as in *RQ2*, we are able to quickly discard many irrelevant snippets and also identify some matches. We recognize four 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, our local repository is small, and some queries may require a solution that we have not encoded. By our

current methodology, those potential solutions are ignored.[16] Third, our encodings are limited to a small subset of the Java language that handles single-path programs. Regardless, these small programs were still found relevant to the programming tasks according to the opinions of 19 participants in an empirical evaluation. Fourth, by pointing a keyword-based search engine at a local repository, the ranking capabilities may have been handicapped, which may have artificially reduced the relevance of the top 10 results captured by the P@10 metric. As we move forward, comparing against generic Google will be necessary, and necessitate the development of our own ranking algorithms.

In the Yahoo! Pipes study for *RQ3*, symbolic encodings found more relevant examples, but the concrete encodings could more quickly discard 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 at a different time can yield a different input/output, and consequently a different set of relevant results.

With the SQL study for *RQ4*, solving time increased with input size. Our instantiation of SQL only works on integers, and it is likely that the time would be much longer in the presence of more complex datatypes; further study is needed.

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

As this approach has been implemented in only three languages, applicability beyond these languages is yet to be explored so generality is a concern. Adapting the approach to a new language takes an effort, but we did not evaluate it from that perspective at this exploratory stage.

The query model of input/output pairs may not be representative of a general and realistic programming model for programs. Based on the evaluation of questions asked on stackoverflow, the input/output model seems reasonable (Section 2.2). Combining the input/output with keywords, allowing partial programs, or negative examples may be useful and a part of our future work.

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 beyond the evaluation of search results in *RQ1*. To show the benefits in practice requires an empirical study with actual programmers, which will require a robust prototype with complete interface. Still, 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.

The final point of discussion here comes from the legal implications of encouraging code reuse. In the evaluations, the source-code repositories we have used come from open-source, publicly available repositories. However, within a company, reuse of public resources may be discouraged or constrained to specific types of licenses. Implementing the search within a company where the repository is built from company code would skirt these issues. It is also worth pointing out that these issues are not just faced by us, but by any researcher or practitioner involved with search.

5.6. Summary

To summarize the studies, we revisit the research questions and discuss the findings. In general, although our approach covers only a limited amount of each language, the results are promising. As our language support increases and our implementation

[16]To alleviate this threat, we can apply abstractions to expand the space of matching program behavior, which is evaluated for Yahoo! Pipes.

is able to handle larger and more complex programs, we anticipate that this search approach will become even more effective when compared to the state-of-the-practice.

5.6.1. RQ1: How Does Our Search Compare to Syntactic Searches from the Perspective of the Programmer? In this study, we used the Java language and evaluated how well our search results compare to the results found using a keyword-based search from the programmer's perspective. We found that, when using the same repository, a keyword-based search returns over twice as many results as our approach, but among the top ten, our approach is nearly three times more effective at returning relevant results based on the opinions of 19 programmers in an empirical study.

5.6.2. RQ2: How Much Can Existing Search Approaches Be Improved by Augmenting Results with Our Search Approach? In this study, we conducted a Google search and extracted all code snippets from the top 10 results. Using those as a local repository, we conducted a search with input/output examples to prune the space of results. By using our search on top of Google, the number of snippets returned could be reduced by 34%.

5.6.3. RQ3: What Is the Impact of Tweaking Search Parameters on the Search Effectiveness? In this study, we used Yahoo! Pipes and evaluated the impact of tweaking three search parameters, namely solver time, abstraction, and specification size, on precision and recall. The results showed that a search using concrete pipe encodings can discard irrelevant programs faster than with symbolic encodings. The maximum allowed solver time has a clear impact on recall where lower solver times lead to lower recalls. A similar effect was observed with manipulations on the specification size, where smaller specifications led to lower precision. However, smaller specifications also returned results faster.

5.6.4. RQ4: What Is the Impact of Query Complexity on Search Time? This study considered SQL, and the results show that the size of the specification has a clear impact on the solver time, and that the complexity of the specification likewise has an impact on solver time. This echoes some findings in *RQ3* where the size of the specification had an impact on precision. The differences are that with the SQL study, the specifications were designed to return *sat*, so precision was 1.00 by design. Additionally, the SQL specification considered only integers whereas the Yahoo! Pipes specification considered integers and strings. Understanding which factors lead some specifications to have a longer runtime is left for future work.

6. RELATED WORK

We have motivated, defined, instantiated, and evaluated a new approach to source-code search that uses input/output examples as specifications and an SMT solver to identify search results. In this section, we discuss the related work.

Our approach is related to recent work in code search, code reuse, verification and validation, and program synthesis.

6.1. Code Search

We have described an approach to code search that is semantic and uses input/output examples to define the queries, which is closely related to research in code search.

Recent studies have revealed that programmers frequently use general search engines to find code for reuse [Sim et al. 2011], and our own study confirms these findings [Stolee and Elbaum 2012a]. More specialized syntactic code search engines in the state-of-the-practice (e.g., Koders, Krugle) also incorporate filtering capabilities (e.g., language, libraries) and program syntax into the query to guide the matching process, such as type signatures of desired code [Sim et al. 2011]. These approaches search at an Internet scale, whereas our search approach operates over repositories. Other

approaches in the state-of-the-art add natural language processing to increase the potential matches [Grechanik et al. 2010; McMillan et al. 2011]. Our work is different in that the search is semantic, but as we show (Section 5.2.3), both approaches are complementary and can be combined.

Early work in semantic code search required developers to write complex specifications using first-order logic or specialized languages (e.g., Ghezzi and Mocci [2010], Penix and Alexander [1999], and Zaremski and Wing [1997]), 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) [Lazzarini Lemos et al. 2007; Podgurski and Pierce 1993; Reiss 2009], but these approaches require that the code be executed to find matches. Some approaches also require a keyword query to first prune the search space, which could miss some solutions [Reiss 2009]. Further, executing test cases only returns exact matches, missing many relevant matches that may have a slightly different signature (e.g., extra parameter). Other search approaches use sequences of API calls [Mishne et al. 2012] or sequences of textual statements [Chan et al. 2012] as queries to find code that performs the specified actions in a specified order, but implementation details are required for an effective search.

6.2. Code Reuse

In the code reuse process, there are two primary activities: finding and integrating. Our approach focuses on finding, which is what we have evaluated, but it has potential to be useful with integration.

For effective reuse, scope and dependencies must be understood for developers to effectively integrate code [Garlan et al. 1995]. Some recent work assists programmers with integrating new code by matching it to structural properties in their development environment (e.g., method signature, return types) [Cottrell et al. 2008; Holmes et al. 2006]. Real-time clone detection can promote reuse by identifying code clones as they are developed, but again this depends on a developer having a sense of how to implement code [Lee et al. 2010]. Further, while these approaches guarantee structural matching, the behavior of the integrated code may not be well understood.

6.3. Verification and Validation

In this work, we have talked about how symbolic analysis is used to generate constraints that represent the program behavior, and that this representation is used in the search process. Symbolic execution [Clarke 1976; Clarke and Richardson 1985; King 1976] is a technique that executes code with symbolic, rather than concrete, values, and can generate such symbolic summaries of source code. These are similar to the summaries that our implementation generates to represent code behavior. For two of our languages presented in this work, SQL and Yahoo! Pipes, symbolic execution tools are not readily available. For Java, however, tools like the symbolic execution extension [Khurshid et al. 2003; JPF-symbc 2012] to the Java PathFinder model checker [Visser et al. 2003] can generate symbolic summaries that we can use, but are limited in the datatypes that are supported. At this point, part of our ongoing work is to integrate our encoding process with such tools, taking advantage of their capabilities to generate summaries for certain complex code structures.

In validation, constraint 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 [Binnig et al. 2007]. Other work in test case generation for SQL queries has used SMT solvers to generate tables based on queries [Veanes et al. 2010]. 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).

6.4. Program Synthesis

Previous work in the area of automated program generation [Balzer 1985] 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 from input/output examples (e.g., Godefroid and Taly [2012], Gulwani et al. [2011], and Harris and Gulwani [2011]). 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 templates [Godefroid and Taly 2012] or a domain-specific grammar that can be traversed exhaustively [Gulwani et al. 2011; Harris and Gulwani 2011] to generate a program that matches the programmer's examples. A similar approach uses the source and destination (akin to input and output) objects to synthesize for finding code snippets based on types, as is done in Jungloid [Mandelin et al. 2005]. This approach is particularly useful for type conversion. Our search, on the other hand, returns results based on concrete examples of desired behavior.

7. CONCLUSION

We present an approach to source-code search that uses input/output examples as queries and searches a repository for source code that matches the defined behavior. The novelty of the approach resides in using an input/output example as a query and in using a constraint solver to assist with the matching process. This necessitates a transformation process on the source code and the specifications into first-order logic so the solver can identify matches.

To motivate the need for better code search, we surveyed 99 programmers about their search habits, finding that code search is a common task and that current search tools are often inadequate. To assess the viability of an input/output model for queries, we explored questions asked by the community on stackoverflow and found that questions are frequently accompanied by input/output examples, indicating that programmers already think in this way when looking for help online.

We discuss the potential and trade-offs of our search approach over the state-of-the-practice and the state-of-the-art, describe how to encode search queries and programs in three languages, namely the Java String library, Yahoo! Pipes, and SQL select statements, and explore the effectiveness of our approach in each of these domains. Generality and efficiency in the context of richer programs, such as those containing loops and other complex constructs, are concerns that still need to be addressed. Despite this, we have shown that this approach is applicable in a variety of languages, can handle nontrivial specifications, is flexible in finding programs that are *close* matches that can be easily modified to satisfy the user specifications, and can be used in lieu of or to complement the state-of-the-practice code searches. This is just one step toward our ultimate goal of leveraging existing resources, such as source-code repositories, to positively impact programmer productivity.

REFERENCES

Robert Balzer. 1985. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Engin.* 11, 11, 1257–1268.

Carsten Binnig, Donald Kossmann, and Eric Lo. 2007. Reverse query processing. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE'07).* 506–515.

Nikolaj Bjorner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 307–321.

Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected api subgraph via text phrases. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM Press, New York.

Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Engin. SE-2*, 3, 215–222.

Lori A. Clarke and Debra J. Richardson. 1985. Applications of symbolic evaluation. *J. Syst. Softw.* 5, 1, 15–35.

Rylan Cottrell, Robert J. Walker, and Jorg Denzinger. 2008. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 214–225.

Nick Craswell and David Hawkings. 2004. Overview of the trec 2004 webl track. In *Proceedings of the 13th Text Retrieval Conference (NIST'04)*. 1–9.

Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.

Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Theory and Practice of Software (TACAS'08/ETAPS'08)*. 337–340.

Gerhard Fischer, Scott Henninger, and David Redmiles. 1991. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the International Conference on Software Engineering*. 318–328.

David Garlan, Robert Allen, and John Ockerbloom. 1995. Architectural mismatch: Why reuse is so hard. *IEEE Softw.* 12, 6, 17–26.

Carlo Ghezzi and Andrea Mocci. 2010. Behavior model based component search: An initial assessment. In *Proceedings of the ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*. 4.

Patrice Godefroid and Ankur Taly. 2012. Automated synthesis of symbolic instruction encodings from i/o samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 441–452.

Mark Grechanik, Chen Fu, Qing Xie, Collin Mcmillan, Denys Poshyvanyk, and Chad Cumby. 2010. Exemplar: EXEcutable examples archive. In *Proceedings of the International Conference on Software Engineering*. 259–262.

Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 50–61.

Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE Press, 842–851.

William R. Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *SIGPLAN Not.* 46, 6, 317–328.

Reid Holmes, Robert J. Walker, and Gail C. Murphy. 2006. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Engin.* 32, 12, 952–970.

Jeff Huang and Efthimis N. Efthimiadis. 2009. Analyzing and evaluating query reformulation strategies in web search logs. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*. ACM Press, New York, 77–86.

M. Cameron Jones and Elizabeth F. Churchill. 2009. Conversations in developer communities: A preliminary analysis of the yahoo! Pipes community. In *Proceedings of the 4th International Conference on Communities and Technologies (C&T'09)*. 195–204.

JPF-Symbc. 2012. Symbolic pathfinder. http://babel_sh.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 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, 553–568.

Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: A solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'09)*. 105–116.

James C. King. 1976. Symbolic execution and program testing. *Comm. ACM* 19, 7, 385–394.

Koders. 2012. Koders. http://code.ohloh.net/.

Amy N. Langville and Carl D. Meyer. 2006. *Google Page Rank and Beyond*. Princeton University Press.

Otavio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. 2007. CodeGenie: A tool for test-driven source code search. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. 917–918.

Mu-Woong Lee, Jong-Won Roh, Seung-Won Hwang, and Sunghun Kim. 2010. Instant code clone search. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM Press, New York, 167–176.

David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. 2005. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM Press, New York, 48–61.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. 111–120.

Mechanicalturk. 2010. Amazon mechanical turk. https://www.mturk.com/mturk/welcome.

Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. 997–1016.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2004. *Principles of Program Analysis*. Springer.

John Penix and Perry Alexander. 1999. Efficient specification-based component retrieval. *Autom. Softw. Engin.* 6, 2, 32.

Pipes. 2012. Yahoo! pipes. http://pipes.yahoo.com/.

Andy Podgurski and Lynn Pierce. 1993. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Engin. Methodol.* 2, 3, 18.

Steven P. Reiss. 2009. Semantics-based code search. In *Proceedings of the International Conference on Software Engineering*. 243–253.

Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. 2013. Reverb: Recommending code-related web pages. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE Press, 812–821.

Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Engin. Methodol.* 21, 1, 4:1–4:25.

Rishabh Singh and Armando Solar-Lezama. 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 Press, New York, 289–299.

Smtlib2. 2012. SMT-LIB. http://www.smtlib.org/.

Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma. 2012. Discovering how end-user programmers and their communities use public repositories: A study on yahoo! pipes. *Inf. Softw. Technol.* 55, 7, 1289–1303.

Kathryn T. Stolee. 2013. Solving the search for source code. Ph.D. dissertation, University of Nebraska at Lincoln.

Kathryn T. Stolee and Sebastian Elbaum. 2010. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*.

Kathryn T. Stolee and Sebastian Elbaum. 2011. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the International Conference on Software Engineering*. 81–90.

Kathryn T. Stolee and Sebastian Elbaum. 2012a. Solving the search for suitable code: An initial implementation. Tech. rep. CSE 126, University of Nebraska-Lincoln.

Kathryn T. Stolee and Sebastian Elbaum. 2012b. Toward semantic search via smt solver. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

Kathryn T. Stolee and Sebastian Elbaum. 2013. On the use of input/output queries for code search. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*.

Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. 2010. Qex: Symbolic sql query explorer. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. 425–446.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM Press, New York.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model checking programs. *Autom. Softw. Engin.* 10, 2, 203–232.

Ian H. Witten and Dan Mo. 1993. TELS: Learning text editing tasks from examples. In *Watch What I Do,* MIT Press, 183–203.

Amy Moormann Zaremski and Jeannette M. Wing. 1997. Specification matching of software components. *ACM Trans. Softw. Engin. Methodol.* 6, 4, 333–369.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ES-EC/FSE'13).* ACM Press, New York, 114–124.

Received December 2012; revised December 2013; accepted February 2014