6-2012

Solving the Search for Suitable Code: An Initial Implementation

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

Sebastian Elbaum
University of Nebraska-Lincoln

Solving the Search for Suitable Code: An Initial Implementation

Kathryn T. Stolee and Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE, U.S.A.
{kstolee, elbaum}@cse.unl.edu

***Abstract*—Searching for code is a common task among programmers, with the ultimate goal of finding and reusing code or getting ideas for implementation. While the process of searching for code – issuing a query and selecting a relevant match – is straightforward, several costs must be balanced, including the costs of specifying the query, examining the results to find desired code, and *not* finding a relevant result. For the popular syntactic searches the query cost is quite low, but the results are often vague or irrelevant, so the examination cost is high and matches may not be found. Semantic searches may return more relevant results, but current techniques that involve writing complex specifications or executing code against test cases are costly to the developer, and *close matches* cannot be easily identified.**

In this work, we address these limitations and propose an approach for semantic search in which developers specify lightweight, incomplete specifications and an SMT solver automatically identifies programs from a repository that match the specifications. The program repository is automatically encoded as constraints offline so the search for programs is efficient. The program encodings cover various levels of abstraction to enable partial matches when no, or few, exact matches exists. We present empirical evidence showing the lightweight specifications can be accurately defined by developers, instantiate this approach on a subset of the Yahoo! Pipes mashup language, and outline extensions to other programming languages.

I. Introduction

"Hasn't anyone already written some code to solve this problem?" This is a perennial question asked by every developer facing a new problem, trying to get a grasp of the task ahead by exploring existing solutions or to accelerate development by reusing code from others. With the increasing number of large and publicly accessible code repositories (e.g., by 2007, Google Code Search, Krugle, and Merobase had each indexed over 10 million files [11]), one would be tempted to at least answer "very likely," especially for smaller problems that can be solved with a snippet of code, an algorithm, or a class. The frequency with which developers search for code, evidenced in our findings and recent studies, shows that developers agree. The challenge, however, lies in finding cost-effective ways for a developer to specify the problem precisely enough and for a search engine to identify suitable code that matches the problem.

Today, general search engines are the most common and often effective way for developers to find suitable code [26]. To define their problem, developers provide a textual query describing, for example, the name or description of a function they desire, and the search engine attempts to find a match among the indexed programs' pages. More specialized search engines (e.g., Google Code Search, Krugle, Merobase) incorporate various filtering capabilities (e.g., language, domain, scores) and program syntax into the query to better guide the matching process [26]. Some recent approaches also add natural language processing to increase the potential matching space [8], [18].

Such syntactic approaches have a low entry cost for specifying the problem and may be effective in identifying functionality that can be concisely captured with structural components like function names, but often return irrelevant results that developers must peruse before finding something useful or giving up, and cannot capture behavior not directly tied to source code syntax or documentation [21]. Some more sophisticated approaches have improved precision by incorporating semantics, but are not common in practice, in part because they often require developers to incur additional costs, such as writing complex specifications [31]. Partial specifications defined through test cases [16], [24], [25] mitigate that cost, but result in inefficiencies that do not scale (e.g., running tests against all programs) and are too specific, which means that some suitable matches may be ignored.

Through this work we propose a novel semantic search approach that addresses these limitations. First, it allows developers to specify a lightweight, incomplete specification in the form of input/output pairs and allows for incremental refinement of the specification. Second, instead of just indexing crawled programs, it encodes programs as constraints representing behavior. Third, it employs an SMT solver to identify code in a repository, encoded as constraints, that matches the specifications, using different levels of abstraction to relax the matching criteria. The success of the proposed approach hinges on overcoming three main challenges.

First, the cost of defining the problem must be reasonable for the developer. Providing input/output pairs requires more effort and may be more fault prone than providing a keyword, so the effectiveness of the search must compensate for that extra effort. In general, the cost of specifying the input/output needs to be less than the cost of building the desired code from scratch. We provide evidence that in some domains developers can quickly and effectively provide such lightweight specifications and that they already do so in many forums when asking

for help. Further, the incremental nature of our approach means that the developer can provide as many input/output pairs as they can budget and get matches that are guaranteed to fit the partial specifications.

Second, we must address the feasibility of mapping programs as a set of constraints that can be solved against the specifications. The level of granularity for encoding must balance the cost of search (a level too fine could result in constraint systems that cannot be resolved in a reasonable amount of time) with the precision of matches (a level too coarse could return a plethora of matches). We are not as concerned about the efficiency of this mapping; as is the case with traditional search engines, the process of crawling and encoding is performed automatically and offline. Instead, we are concerned with the ability to instantiate the approach for real programming domains. In this work we instantiate and implement the mapping for one domain, the popular Yahoo! Pipes (repository of over 100,000 mashup programs to process RSS feeds and a community of 90,000 users [13]), sketch the mapping to `SQL`, and discuss extensions to others.

Third, the efficiency and effectiveness of the approach must provide enticing tradeoffs for the developer. Efficiency is determined by the constraints' complexity, the input size, the level of abstraction considered, and the solver speed. Our assessment uses a state-of-the-art solver [30] and explores artifacts with different complexities, sizes of the input/output, and two levels of abstraction. In terms of effectiveness, we analyze the returned matches and highlight how semantic matches can find unexpected yet useful solutions, and how by simply weakening or strengthening the specifications or the encodings, developers can quickly prune irrelevant matches or find matches that are *close enough* semantically to be reused. The contributions of this work are:

- Provide characterization of how developers use search to find code based on a survey of 100 participants, and the types of questions programmers ask that are not easily answered by search based on an analysis of 100 questions from stackoverflow.com
- Define an approach to search for code with lightweight specifications using an SMT solver to identify matching code, and illustrate the feasibility and success of this approach using a subset of the Yahoo! Pipes language
- Assess the approach from several perspectives: the cost of providing the specifications, number of matches, time for retrieval, and effectiveness of the search in identifying matches
- Discuss further instantiations of this approach in other domains, including `SQL` and more traditional programming languages, and the applicability of our approach in a broader reuse context

II. Motivation

Our work is built on the premise that developers search for code often but the results are noisy, and that syntactic searches are not enough for developers to express what they want to find. In this section we offer support for that premise based on the results of a survey and a preliminary analysis of the questions and answers posted online by developers.

To explore how people search for code, we conducted a survey with ten questions on 109 participants, asking about their programming and search activities, how they search for code, and what they do with useful code once found. [1] Of the 109 participants, 43 came from junior/senior undergraduate classes at UNL while the remaining came from Mechanical Turk [20] (a qualification test was used to throw out inconsistent survey results and ensure response quality). Less than a quarter of participants reported to have less than a year of programming experience, half reported to have between 2 and 5 years, and the rest had more than 5 years.

Table I summarizes our findings in terms of programming and search frequency. Among the participants, 43 (39%) reported that they program daily and 49 (45%) program weekly. Of those who program daily, over half (25 of 43) also search for code daily, and of those who program at least weekly, 85% also search for code at least weekly (78 of 92). Two-thirds mentioned using *the web*, *the internet*, or specifically *general Google* to search for code online. A fifth (21%) mentioned searching stackoverflow.com specifically, and only 17% mentioned using a code-specific search engine like *Google Code Search*. The most popular methods used for code search (in a multi-select question) were keyword (72%), function name (45%), and libraries used (35%).

The participants reported that they must explore an average of 3.4 snippets of code before something useful is found. Half of the participants reported, in a multi-select question, that once useful code is found, they would copy/paste and modify it, 67% would use it to get ideas for implementation, and 13% would copy/paste the code as is.

These results are in line with recent findings from a study investigating the effectiveness of different code search approaches [26]. In that work, all 36 study participants had some programming experience, and 50% reported to search for code "frequently" while 39% did it "occasionally", again showing that code searches are a common practice among developers. The study indicated that developers looking for code found approximately 3 out of the first 10 matches useful, which seems to align with our finding. This study did not focus on the cost of examining spurious search results, but it is worth pointing out that participants were allowed to perform preliminary searches and to refine their query to better explore the space of solutions. These preliminary searches and examination of the results, although unaccounted, clearly increase the search costs.

To provide a reference for the domain that we later target, Yahoo! Pipes, and to further illustrate the challenges for developers using existing search mechanisms, we performed five searches for mashups (the rationale for the choice of these mashups is justified in Section IV-A),[2] querying for the

[1] The survey and UNL Institutional Review Board approved process can be found at cse.unl.edu/~kstolee/fse2012/).

[2] Search results reflect the state of the Yahoo! Pipes repository on February 22, 2012.

TABLE I
PROGRAMMING AND SEARCH FREQUENCY

Activity	Daily	Weekly	Monthly	Never
Programming	43	49	8	9
Code Search	25	53	22	9

URLs used in each mashup. The number of matches for the searches can be staggering (more than 1,000 for two examples) but not surprising as many mashups may include common feeds or feed aggregator websites. The average number of relevant matches among the top ten results, determined by the pipe behavior, is 0.9. Using other built-in search capabilities does not fair much better. Searching by particular components retrieves even more results and would require for the developer to know not just what the mashup should do, but also how it would be built, and searching for tags is highly dependent on the community's ability and decision to systematically categorize their artifacts.

The last motivating piece of support comes from the an analysis of the questions posted on stackoverflow.com, a free question and answer site, where developers have posted over 2.5 millions questions and responses. To constrain our analysis and to motivate the selection of the `SQL` domain, which we later explore, we searched for questions tagged with "mysql" and "select" keywords. Of the 1,500 questions returned, we examined the 100 questions with the most votes (and their corresponding answers and annotations) returned on February 19, 2012. Some clear themes on the question types emerged; we counted the frequency of questions that match each type (some questions fit multiple) and show the results in Table II. Also reported is the number of questions that include an example of the desired behavior either in the textual description or by sample table or record.

The dominant question type is *"How do I . . .,"* for example: *"I really can't find a simple or even any solution via `SQL` to get unique data from DB (MySQL). I will give a sample (simplified): TABLE t [sample input table] And now I want output - something like distinct(apple) and distinct(color) together and order by weight desc: [sample output records] . . ."* [3] Syntactic search mechanisms are not well equipped to answer this type of question as the developer does not know what `SQL` query or query components may be used to solve the problem; the developer asking this type of question only knows the behavior that is desired. These types of questions usually come with examples (76 out of the 81, 38 being snippets of tables that serve as inputs and records they expect as outputs, which is something we leverage later in our approach) that help developers better specify the required behavior. Corroborating the previous observation, *"What is ..."* questions are not asked often, further indicating that this type of question is well handled by existing mechanisms like tutorials or other syntactic search engine results.

There are several threats to validity of the evidence presented thus far (specifically, the pools of survey participants and questions selected, participants self-reporting on their activities, our biasing in classifying the questions) that can only be addressed through more extensive studies. Still, the results are consistent with previous findings. Furthermore, there seem to be clear trends indicating that developers rely extensively on search engines to find code to reuse or ideas on how to approach a problem, they must examine several pieces of code before finding something relevant, and for many questions associated with program behavior, current syntactic search mechanisms are inadequate.

[3] stackoverflow.com/questions/7639830

TABLE II
`SQL` QUESTION TYPES IN STACKOVERFLOW

Question Type	Frequency	Examples
Best way to do ...?	11	10
Can I do x with/without y?	5	4
How do I ...?	81	76
How does foo work?	8	4
X versus Y?	4	4
What is ...?	1	0
What's wrong with ...?	27	27

III. APPROACH

In this section we describe the key building blocks of the approach that enables an incremental semantic search in which a developer provides lightweight specifications, an encoder maps programs to constraints, and a solver identifies which encoded programs match the specifications. We first explain the attributes and components of the framework.

A. Framework

Our approach operates within the framework illustrated in Figure 1. As mentioned in Section I, the success of our approach depends on effectively addressing three critical challenges: defining cost-effective lightweight specifications, encoding programs as constraints and solving the constraint systems, and the efficiency and effectiveness of the search, which we can refine through strengthening and weakening the specifications and encoding. Using Figure 1, we discuss each piece of the approach in detail.

1) Lightweight Specifications: Instead of textual queries, this approach takes lightweight specifications that characterize the desired behavior of the code (*Lightweight Specifications* in Figure 1). These specifications, LS, are represented as input/output pairs, $LS = \{(i_1, o_1), \ldots, (i_k, o_k)\}$, for k pairs, and take different forms depending on the domain. The size of LS defines, in part, the strength of the specifications and hence the number of potential matches. This approach allows a developer to provide specifications incrementally, starting with a small number of pairs and adding more to further constrain the behaviors and increase the precision of the search [21].

We recognize that specifying each $(i, o) \in LS$ requires more effort than a standard keyword search and that generally, the cost of writing specifications can be a barrier to adoption in semantic search. Earlier work in semantic search required developers to write complex specifications of the behavior using first-order logic or specification languages (e.g., [7], [22],

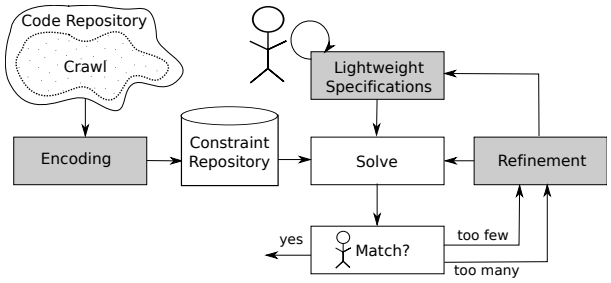


Fig. 1. General Approach

[25], [31]), which can be expensive to develop and error-prone for the programmer. The cost of writing specifications can be reduced by using more incomplete behavioral constructs, such as test cases to describe desired behavior [16], [24], [25], but these approaches require that the code be executed to find matches. For those approaches the search efficiency is defined by the runtime of the reuse code [21], making them expensive and not scalable. Further, since the test cases are executed against the code, these methods cannot identify approximate behavioral matches. Previous work has proposed the use of semantic networks [2] to identify approximate matches, but it requires manual annotations on the code.

2) Encoding and Solving: In our approach, encoding and solving are analogous to the crawling, indexing, and matching processes performed by many information search engines [15]. Offline, a repository (*Code Repository* in Figure 1) is crawled to collect programs. These programs are encoded as constraints (*Encoding*, analogous to indexing), and stored in a repository (*Constraint Repository*). The constraint repository is used by the solver, in conjunction with lightweight specifications, to determine matches (*Solve*, analogous to matching). The *Solve* stage is performed by an SMT solver.

More formally, given a collected set of n programs $SP = \{P_1, P_2, \ldots, P_n\}$, the encoding process is $Encodes : P \rightarrow C_P$, where $P \in SP$ and $C_P = \{c_1 \wedge c_2 \wedge \ldots \wedge c_m\}$ is the set of m constraints that describe P. 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. At the coarsest granularity the encoding would capture none of the program behavior so $C_P = \{\}$. These extremes correspond to the least and the greatest abstraction level (number of matches) and the worst and the best search speeds respectively, but there is a spectrum of choices in between such as encoding at the component level, which we explore in Section III-B.

In the end, the encoding process maps every $P \in SP$ to a set of constraints such that $SP_{enc} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$. Given LS, for each $C_P \in SP_{enc}$, the approach invokes $Solve : (C_P, LS) \rightarrow (sat, unsat, unknown)$ to identify matches. The $Solve$ function returns *sat* when a satisfiable model is found or *unsat* when no model satisfies the constraints. When the solver is stopped before it reaches a conclusion or it cannot handle a set of constraints, *unknown* might be returned.

It is important to point out that the developer is not responsible for transforming the input/output pairs in LS for the solver. The transformation is an automated process performed by the framework in which the encoded input/output pairs are provided to the solver as additional constraints. Thus, the developer is only responsible for generating the lightweight specification in the form used in the domain. Also note that previous work in the area of program synthesis also makes use of solvers to derive a function that maps to an input/output [9]. The key difference is that our approach uses the solver to find a match against existing encoded programs which makes it much more scalable.

3) Refinement: The effectiveness of a search is defined in terms of the relevant matches, which must be found efficiently. Often the solver may return too many or too few results. If the specifications or the encoded program constraints are too weak, many matches may be returned; if they are too strong, the solver may not yield any results. In these cases, refinement is needed to find *close matches* (*Refinement* in Figure 1). We approach refinement from two perspectives: tuning the lightweight specifications and changing program encodings.

a) Tuning Lightweight Specifications.: Two scenarios can result from specifications that are too strong and could benefit from weakening. In the first, since we are dependent on SMT solvers, the content and size of the input/output may require an impractical amount of time for solving. In the second, if LS is too strong, then *sat* may not be returned by any program. One way in which the specifications can be weakened is to use $LS' \subset LS$. Another way is, for some $(i, o) \in LS$, to limit the size of the input i, which implies a subset of o. For example, consider an (i, o) where i and o are lists such that $size(i) = 10$ and $o = [i[1], i[3], i[9]]$ (with zero-based indexing). Reducing the input size so that $size(i) = 9$ causes $o = [i[1], i[3]]$. In these ways, we relax the specifications to either increase the number of matches or more quickly prune out programs that do not match.

For specifications that are too weak, our approach allows refinement and additions to the specifications. We support two solutions for this, complementary to the processes for weakening. In the first, the developer provides additional $(i, o) \in LS_{new}$ to further demonstrate the desired behavior, similar to query reformulation [5]; instead of LS, $LS' = LS \cup LS_{new}$ is used. In the second, the developer can change an original input/output pair to provide more information.

b) Changing Program Encodings.: For encoded program constraints that are too strong, the approach can systematically relax the constraints. Unlike previous approaches that relax matching on pre/postconditions [31], we exploit the fact that most languages contain constraints over multiple data types (e.g., strings, floats, integers, booleans, lists) and relax matching on the specific variables by treating their values as symbolic. $Weakening : C_{P_i} \rightarrow C_{P_i}{}'$ means that

$$Solve : (C_{P_i}, LS) \rightarrow \neg sat \wedge$$
$$Solve : (C_{P_i}{}', LS) \rightarrow sat \wedge$$
$$\exists (i,o) \in LS \mid Solve : (C_{P_i}, LS - (i,o)) \rightarrow sat$$

For program encodings that are too weak, the process is inverted. Strengthening the constraints works in the opposite direction, and we define it as follows: $Strengthening$: $C_{P_i} \rightarrow C_{P_i}{}'$ means that

$$Solve : (C_{P_i}, LS) \rightarrow sat \wedge$$
$$Solve : (C_{P_i}{}', LS) \rightarrow \neg sat \wedge$$
$$\exists (i,o) \in LS \mid Solve : (C_{P_i}{}', LS - (i,o)) \rightarrow sat$$

Encoding weakening and strengthening are performed by traversing an abstraction lattice (see Figure 4(a)), similar to the pre/postcondition lattices in previous work on specification matching [22], [31].

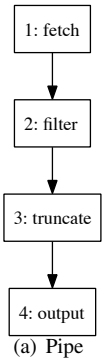
B. Instantiation

Motivated by the popularity of the Yahoo! Pipes mashup environment, the vast number of artifacts available for studying in the online repository, and the clear interfaces that surround the components making them amenable for encoding, we instantiate our approach using the Yahoo! Pipes language. Since 2007, over 90,000 users have created mashups with Yahoo! Pipes [13], forming a public repository of over 100,000 artifacts [23]. To program these mashups, developers use the Pipes Editor, which is accessible through the browser. Mashups are composed by dragging and dropping predefined modules (e.g., a *fetch* module to access an RSS feed, a *filter* module to select records), configuring the modules by setting their fields (e.g., URLs, expressions to specify a filter criterion), and connecting the modules with wires to define the data and control flow. The structure of a mashup forms a directed graph with one to many sources and one sink (the *output* module). Data is generally provided to the pipe by fetching RSS feeds from URLs, and the output is a unified and modified list of the records from the feeds. We show a *filter* module from the language in Figure 4(c) and abstract representations of several pipes structures in Figures 3(a), 5(a), 5(b), 5(c), 5(d), and 5(e), described later.

1) Lightweight Specifications: With Yahoo! Pipes, the input/output specification takes the form of URLs that reference RSS feeds and provide lists of records (input) and the desired record(s) from the feeds (output). In practice, RSS feeds are accessed when the mashup is executed, and so the developer is only responsible for specifying the URLs. Since $i \in (i,o)$ is defined in terms of a list of records, we define $T : URLs \rightarrow i$ to automatically transform the URLs into a list of records from which the output can be derived. From i, the developer chooses the records or parts of records that should be retained in the output, giving form to the lightweight specification $(i,o) \in LS$. For illustration, one example record from an RSS feed that is part of the input to a program used in evaluating the approach (Figure 5(a), described later), is shown in Figure 2.

Field	Value
`Title`	Your Local Doppler Radar
`Descr.`	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.
`Link`	http://www.weather.com/weather/map/93012
`Date`	Fri Jan 13 11:15:22 CST 2012

Fig. 2. Record from an RSS Feed (part of input)



(a) Pipe

Module	Type	Constraint Def
1	equality	$out1 = i$
link(1,2)	equality	$in2 = out1$
2	inclusion	$(contains(in2, r) \wedge substr(field(r), c)) \rightarrow contains(out2, r)$
	exclusion	$contains(out2, r) \rightarrow contains(in2, r)$
	order	...
link(2,3)	equality	$in3 = out2$
3	inclusion	$\forall i (0 \leq i < n) \rightarrow record(in3, i) = record(out3, i))$
	exclusion	$contains(out3, r) \rightarrow contains(in3, r)$
	order	...
link(3,4)	equality	$in4 = out3$
4	equality	$in4 = o$

(b) Constraints for Pipe Example

Fig. 3. Example Constraint Mapping

Once the developer is satisfied with the specification LS, it is automatically transformed into constraints by the framework and later sent to $Solve : (C_P, LS)$.

2) Encoding and Solving: The process of encoding and solving a pipe P given a specification LS is as follows. First, the pipe is refactored for size and simplicity using an infrastructure built for a previous study [27] to reduce the number of modules that need to be encoded, and then the input/output information (i.e., the URLs) is abstracted out of the pipe so the constraints can be solved for any arbitrary LS. Second, each module and wire in the pipe is systematically mapped onto constraints ($Encodes : P \rightarrow C_p$). Third, an SMT solver evaluates $Solve : (C_P, LS)$.

To illustrate the process, we introduce a simple example of an encoded Yahoo! Pipes program in Figure 3; the structure is shown in Figure 3(a) and the constraints are in Figure 3(b). The *fetch* component provides a list of records to the program for some URL, *filter* removes records based on some criteria c, *truncate* performs a *head* operation on the list given a length n, and *output* is the sink of the program.

Abstracting the input/output from the pipe is an important first step in the encoding process. In Yahoo! Pipes, this means removing all URL information so that the program can be solved given any URL; this is shown in the constraint definition for the first module of Figure 3, where $out1 = i$. The original i was a concrete URL before, but after abstraction it is a symbol assigned to i for some $(i,o) \in LS$.

In the second step, the encoding is performed. Since the programs are defined in terms of modules with well-defined

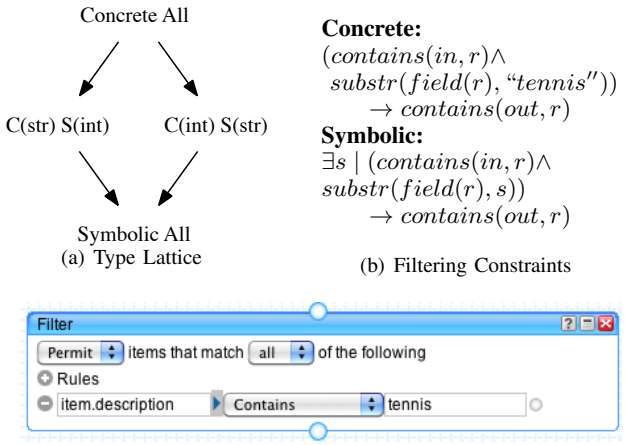
TABLE III
SUPPORTED LIBRARY FUNCTIONS

Function	Type	Definition
$length(s)$	$\mathcal{S} \rightarrow \mathcal{I}$	length of s
$char(s, i)$	$\mathcal{S} \times \mathcal{I} \rightarrow \mathcal{C}$	char of s at position i
$substr(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{B}$	true if s_2 is a substring of s_1
$equals(s_1, s_2)$	$\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{B}$	true if $s_1 = s_2$
$size(l)$	$\mathcal{L} \rightarrow \mathcal{I}$	length of l
$record(l, i)$	$\mathcal{L} \times \mathcal{I} \rightarrow \mathcal{R}$	record of l at position i
$contains(l, r)$	$\mathcal{L} \times \mathcal{R} \rightarrow \mathcal{B}$	true if $r \in l$
$equals(l_1, l_2)$	$\mathcal{L} \times \mathcal{L} \rightarrow \mathcal{B}$	true if $l_1 = l_2$
$field(r)$	$\mathcal{R} \rightarrow \mathcal{S}$	the field string value of r
	$\mathcal{R} \rightarrow \mathcal{I}$	the field integer value of r
$equals(r_1, r_2)$	$\mathcal{R} \times \mathcal{R} \rightarrow \mathcal{B}$	true if $r_1 = r_2$

interfaces and behaviors, and the average pipe has about 8 modules [28], it was natural to consider encoding at the module level. Each module and wire is mapped onto a set of constraints. Since Yahoo! Pipes is a data-flow language, we classify the constraints in terms of inclusion, exclusion, and order, and the links are mapped onto equality constraints. *Inclusion* constraints ensure completeness; all relevant records from the input exist in the output from the module. *Exclusion* constraints ensure precision; all records in the output are relevant and from the input. *Order* constraints (omitted from Figure 3(b) but defined later) ensure that the records are ordered properly in the list. The *equality* constraints ensure that the output of the source module is equivalent to the input of the destination module (e.g., in Figure 3(a), $link(1, 2)$ is encoded as $in2 = out1$ and means the input to the second module (*2: filter*) is the same as the output from the first module (*1: fetch*)).

While the pipe in Figure 3(a) illustrates only four modules, our instantiation includes a larger subset of the Yahoo! Pipes language representing many of the most common constructs. This language subset performs filter, permute, merge, copy, and head/tail operations on lists of records, where a record is a datatype with fields that contain values (e.g., Figure 2). Encoding all supported operations requires six data types: characters ($\mathcal{C}$), strings ($\mathcal{S}$), integers ($\mathcal{I}$), booleans ($\mathcal{B}$), records ($\mathcal{R}$), and lists ($\mathcal{L}$). The functions used in our encoding are summarized in Table III. Table IV provides the constraint descriptions that are at the core of the encoding process. The operation being mapped is in the *Operator* column with the corresponding Yahoo! Pipes module (i.e., for the *Generate* operator, *(yp:fetch)* refers to the *fetch* module in Yahoo! Pipes). Next is a textual description and representation of the constraints using first-order logic (except for the *order* constraint on the *Merge* operator, in which we deviate from the logic representation for brevity).

Encoding the Yahoo! Pipes language fragment requires evaluating substring and equality relations over strings, and enumeration over all records in a list, as outlined in Table III. To efficiently support these operations, we consider bounded strings and list, where the bounds are configurable (in line with recent work on string constraints [1], [14]).



(a) Type Lattice

(b) Filtering Constraints

(c) Filtering Component in Yahoo! Pipes

Fig. 4. Type Lattice Example

Once encoded, the third step is to solve the constraint system, $Solve : (C_P, LS)$. In our implementation, we use the Z3 SMT Solver v3.2 from Microsoft Research [30].

3) Refinement: We apply refinements in two ways: tuning the lightweight specifications and relaxing the program encodings.

a) Tuning Lightweight Specifications.: In many cases, the time required to evaluate $Solve : (C_P, LS)$ can be expensive. To control the cost, we constrain $T : URLs \rightarrow i$ to limit the number of records in the input retrieved from each URL. In Section IV-C, we show the effects of changes in the input size from $[1, 2, \ldots, size(i)]$ on the number of matches.

b) Changing Program Encodings.: We define constraints over two datatypes that can hold concrete or symbolic values: integers and strings. The type lattice we use is illustrated in Figure 4(a). Stronger constraints utilize concrete values and identify exact matches (*Concrete All*), while weaker constraints utilize symbolic values (*Symbolic All*). The lattice also relaxes the concrete values of either the integers (i.e., *C(str) S(int)*) or the strings (i.e., *C(int) S(str)*). To illustrate, we focus on the filter component in Figure 3(a) and show it in the Yahoo! Pipes language in Figure 4(c). For the *inclusion* constraint, the concrete encoding in Figure 4(b) requires $substr(field(r), \text{“}tennis\text{”}) = true$, whereas the symbolic constraint requires $substr(field(r), s) = true$ for some string s. *Filter* components also handle integer constraints using equality, less than, and greater than comparisons. Additional integer constraints are used in *head/tail* components to define lengths of the resulting lists.

IV. STUDY

The evaluation of our technique aims to address two of the challenges outlined in Section I: the cost of defining the specifications and the efficiency and effectiveness with which the approach identifies matches. We perform two studies, one to address each challenge, using the same set of artifacts.

TABLE IV
INCLUSION/EXCLUSION/ORDER/EQUALITY CONSTRAINT DESCRIPTIONS FOR SUPPORTED CONSTRUCTS

Operator	Type	Description	Constraint
Generate (yp:fetch)	inclusion	all records in input are in the output	$contains(in, r) \rightarrow contains(out, r)$
	exclusion	all records in output are in the input	$contains(out, r) \rightarrow contains(in, r)$
	order	ordering of records in input and output is same	$\forall i(0 \leq i < size(out) \rightarrow record(in, i) = record(out, i))$
Filter (yp:filter)	inclusion	for all records in input that have criteria, they are in output	$(contains(in, r) \wedge field(r) = c) \rightarrow contains(out, r)$
	exclusion	all records in output are in input	$contains(out, r) \rightarrow contains(in, r)$
	order	for all records in output, their ordering is preserved from the input	$\forall r_1, r_2((contains(out, r_1) \wedge contains(out, r_2) \wedge (\exists i, j(record(out, i) = r_1 \wedge record(out, j) = r_2 \wedge i < j)) \rightarrow (\exists k, l(k < l \wedge record(in, k) = r_1 \wedge record(in, l) = r_2)))$
Permute (yp:sort)	inclusion	all records in input are in output	$contains(in, r) \rightarrow contains(out, r)$
	exclusion	all records in output are in input	$contains(out, r) \rightarrow contains(in, r)$
	order	all records in the output are sorted according to the property.	$\forall i, j((0 \leq i, j < size(out) \wedge i < j) \rightarrow field(record(out, i)) \leq field(record(out, j)))$
Head/Tail (yp:truncate) (yp:tail)	inclusion	all records in input up to the cutoff n are in the output	$\forall i(0 \leq i < min(n, size(in)) \rightarrow record(in, i) = record(out, i))$
	exclusion	all records in output are in input	$contains(out, r) \rightarrow contains(in, r)$
	order	order of records in input and output are same	$\forall i(0 \leq i < size(out) \rightarrow record(in, i) = record(out, i))$
Merge (yp:union)	inclusion	all records in the n input(s) are in the output	$contains(in_1, r) \vee contains(in_2, r) \vee \ldots \vee contains(in_n, r) \rightarrow contains(out, r))$
	exclusion	all records in output are in one of the n input(s)	$contains(out, r) \rightarrow contains(in_1, r) \vee contains(in_2, r) \vee \ldots \vee contains(in_n, r)$
	order	the order of records in n input list(s) are preserved in output	$out = [in_1 \cdot in_2 \cdot in_3 \cdot in_4 \cdot in_5]$
Copy (yp:split)	inclusion	all records in input are in the output	$contains(in, r) \rightarrow contains(out_1, r) \wedge contains(out_2, r)$
	exclusion	all records in output are in the input	$contains(out_1, r) \vee contains(out_2, r) \rightarrow contains(in, r)$
	order	ordering of records in input and output is same	$\forall i(0 \leq i < size(out_1) \rightarrow record(in, i) = record(out_1, i) \wedge record(in, i) = record(out_2, i))$
Output (yp:output)	inclusion	all records in input are in the output	$contains(in, r) \rightarrow contains(out, r)$
	exclusion	all records in output are in the input	$contains(out, r) \rightarrow contains(in, r)$
	order	ordering of records in input and output is same	$\forall i(0 \leq i < size(out) \rightarrow record(in, i) = record(out, i))$
Link (yp:wire)	equality	the source and destination of the link have equal values	$equals(source, destination) = true$

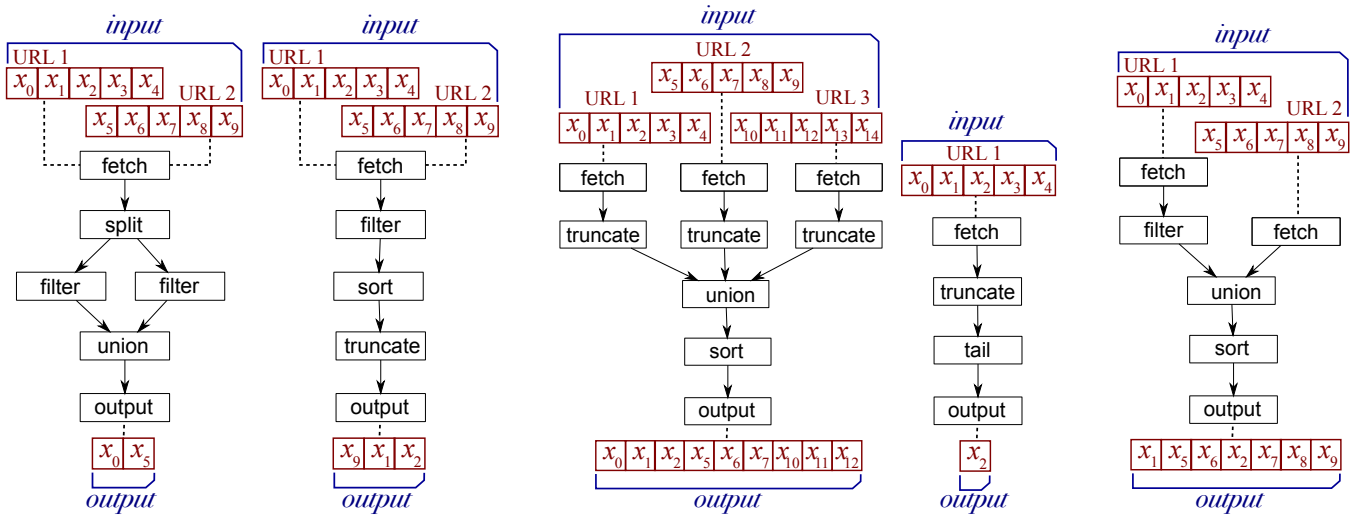
A. Artifact Selection

We obtained a pool of Yahoo! Pipes programs to form the code repository (Figure 1). In a previous study [28], we scraped 32,887 pipes from the public Yahoo! Pipes repository between January and September 2010 by issuing approximately 50 queries against the repository (each of which returned a maximum of 1,000 pipes) and removing all duplicates; this forms the code repository.

We identified five representative pipes from which the example lightweight specifications were formed. These are intended to be "typical" pipes in the repository based on structural uniqueness and their popularity (measured by the number of clones - or explicit copies - of the pipes), and were selected as follows. The pipes were grouped by similar structures (i.e., the modules and wires but not necessarily the field values, are the same for every pipe in each cluster), yielding 2,483 clusters. Among those, 154 clusters (2,859 pipes), used only the language supported by our encodings (Section III-B2). Clusters that were equivalent post-refactoring or had fewer than five pipes were removed, as were the most and the least popular clusters. From each of the remaining five clusters, we randomly selected one pipe to serve as the five example pipes used for evaluation.

For each example pipe P, we generated the lightweight specifications LS by extracting the URLs from P, using $T : URLs \rightarrow i$ to generate i, and then executing the pipe, setting the output to o. To capture the behavior of the pipes while keeping the solver time reasonable, we tuned T to limit the number of records from each URL to five, though this bound may change in practice; o was modified based on the records retained in i. An additional bound of 100 characters was imposed on the string lengths, though this, too, is configurable. The following describes the structure, behavior, and $LS = \{(i, o)\}$ for each example pipe P.

Example 1, P_1, and the lightweight specification, LS_1, are shown in Figure 5(a). In the structure of P_1, the fetch retrieves RSS feeds as input and the *split* performs a copy on the input, sending one copy along each output wire. Each *filter* looks for a different substring ("10-Day" or "Current") in the title field for each record, and the *union* concatenates the



(a) Example 1 (b) Example 2 (c) Example 3 (d) Example 4 (e) Example 5

Fig. 5. Example Pipes and their Lightweight Specifications

lists from the two *filter* modules. The lightweight specification $LS_1 = \{(i, o)\}$ is labeled as *input* and *output* in Figure 5(a). Within the input, each box labeled X_j represent a distinct record at index j in the input list. There are ten records in the input coming from two URLs (i.e., $i[0..4]$ are from $URL1$ and $i[5...9]$ are from the $URL2$). In the output, two records, $X_0 = i[0]$ and $X_5 = i[5]$, are retained; the order of the records in the output is illustrated in Figure 5(a); in this example, $o[0] = X_0$ and $o[1] = X_5$.

Example 2, P_2, and the lightweight specification, LS_2, are shown in Figure 5(b). The *filter* looks for "hotel" as a substring in each record's description field and the *sort* is based on the records' publication dates. The *truncate* module permits only three records. In the specification, there are ten records in the input from two URLs (i.e., $i[0..4]$ are from $URL1$ and $i[5...9]$ are from $URL2$). The output has three records, but the order of the records in the output is different from that in the input (i.e., $o = [i[9], i[1], i[2]]$).

Example 3, P_3, and the lightweight specification, LS_3, are shown in Figure 5(c). The *sort* is based on publication date and each *truncate* permits three records. This example has three URLs with one assigned to each input path (*fetch* module). The specification has 15 records in the input and nine in the output (the first three records from each URL).

Example 4, P_4, and the lightweight specification, LS_4, are shown in Figure 5(d). The *truncate* and *tail* modules perform head and tail operations on the input list to identify the third record. This example has one URL, and the specification shows just one record, X_2, in the output.

Example 5, P_5, and the lightweight specification, LS_5, are shown in Figure 5(e). The *filter* looks for "au" as a substring in the description field and the *sort* is based on publication date. This example has two URLs; ten records are in the input and seven records are in the output.

B. User Study

Toward the goal of evaluating if input/output pairs are a cost-effective specification model, we designed a user study around ten tasks that, given an input and a behavioral description of the desired program, ask the participants to select the output. Five of the tasks were related to Yahoo! Pipes and used the LS from the five example pipes, where the i was provided in the task and the o was used as the oracle to check participants' responses. The descriptions for each task, shown in Table V for the pipes tasks, were generated by the researchers based on each example pipe's behavior. The other five tasks were related to `SQL` queries and are discussed in detail in Section V.

Participants selected the output from each task using checkboxes next to each record in the input. In scoring, if a participant selected a record that was supposed to be unselected, or vice versa, a point was not awarded. The score awarded was a percentage out of the total points possible in the task.

Since the participants only specified the output in the tasks, this introduces a threat to construct validity. However, for the domains being evaluated, the input can easily be obtained (from a URL or database table), so having the participants specify only the output mimics how our approach could be used in practice. In other domains, the input may need to be user-generated, which could be more expensive.

We used the same participants from the study introduced earlier (Section II), except that only a subset performed each task due to time constraints and self-selection. The Mechanical Turk participants performed the tasks online and the students used a pencil/paper method.[4] Table V summarizes the accuracy and timing data for the pipes tasks. The *n* column shows the

[4]There is no difference in accuracy between the student and mechanical turk participants at $\alpha = 0.10$ on all tasks except *Pipes Task 3* in which the student results are significantly lower ($p = 0.0212$). We suspect this results from differences in instrumentation (i.e., paper vs. online) for the groups.

TABLE V
USER STUDY WITH PIPE TASKS

Task	Textual Description	Accuracy			Timing (m:ss)		
		n	Mean	Median	n_2	Mean	Median
1	Select all records that show the Current Weather Conditions or the 10-Day Forecast for Malibu, Exeter, or Camarillo	63	90%	94%	30	2:30	1:48
2	Select the four most-recent records from the list that contain information about a hotel	60	90%	90%	24	3:48	2:55
3	Select the first three records from each source, where the sources are indicated using different background colors	65	93%	100%	29	1:20	0:46
4	Select the the third most-recent record from the list	72	96%	100%	30	1:14	0:47
5	Select all records with the pink background, and those items from the grey background with "au" in the description	70	95%	100%	30	2:26	2:05

number of participants per task followed by the mean and median accuracy. Timing data was only measured per task for those performed online; there was a fixed time limit for the pencil/paper method considering all tasks. In Table V, n_2 indicates the participants from whom timing data was gathered per task.

Interestingly, the median accuracy for three of the five tasks was 100% and the average accuracy for each task was over 90%. In terms of timing, the median time ranged from 0:46 to 2:55; we find this to be efficient as the timing includes getting familiar with the input and the goal, which is non-trivial for more time-consuming tasks like the second task that involves filtering based on date and content. These findings show that using input/output pairs in this domain is a cost-effective and accurate specification model.

C. Artifact Study

To evaluate the efficiency and effectiveness of our approach, we use LS derived from the example pipes and search a pool of programs for matches. Three factors are manipulated for each example: the size of LS by modifying the size of the input lists as described in Section III-A3, the abstraction level (*Concrete All* and *Symbolic All* from the lattice in Figure 4(a)), and the maximum runtime for each call to $Solve : (C_P, LS)$ ($max\_time = \{5, 30, 300\}$ seconds). For each example pipe and each combination of factors, we search a pool of 2,859 programs for matches (i.e., $Solve : (C_P, LS) \rightarrow sat$).[5] We report the number of matching pipes and the *Time to First Sat*, or *TFS*, which represents the average time until a match is found (calculated by averaging over 250 orderings on the encoded pipes SP_{enc}, with shuffling performed by the Fisher–Yates shuffle algorithm). A '+' before the time means that no satisfiable result was found within the allotted time, so the time displayed is a lower bound in those cases. Our data were collected under Linux on 2.4GHz Opteron 250s with 16GB of RAM.

The results of our experiments are shown in Table VI. There are two tables for each example, one for each the concrete and symbolic abstractions. The first two columns of each table show the sizes of LS, where each row was formed using a different size of (i, o) (some rows are omitted for space). The next columns show the number of matching pipes that returned sat from the pool of 2,859 (*Pipes*) and *TFS* given $max\_time = 5, 30$, and 300.

As expected, using symbolic constraints yields more results than concrete, but it takes longer to find matches since the solver usually decides $unsat$ faster than sat. More matches are found when the size of the input is smaller (weaker specifications) rather than larger (stronger specifications), but many matches are coincidental. For all examples and abstraction levels except for *Example 2: Symbolic* in Table VI(d), at least one match is found when $max\_time = 300$. Comparing against the maximum of $size(i)$ with $max\_time = 300$ for each example, cutting $size(i)$ in half reduces *TFS* six-fold on average; this can help by discarding pipes early that have a long run-time but ultimately return $unsat$. Next, we explore the results per example.

Example 1: Matches in Various Topologies. Table VI(a) shows the results of the concrete search on Example 1. In the top row with $size(i) = 10$ and $max\_time = 300$, the 17 satisfiable pipes have three general topologies. The first is the same as P_1 (Figure 5(a)). The second involves pipes with multiple input paths. Since LS_1 was generated from two URLs, when provided to a pipe with multiple input paths, the input i is split by URL (i.e., $i[0, 4] \rightarrow path1$ and $i[5, 9] \rightarrow path2$). When this happens, the output can be achieved by grabbing the first record from each input list (i.e., using a *truncate* with $n = 1$) since just the first record from each list is in o; three of the matching pipes have this structure. The other 13 matches permit records that contain the substring `http://` in the description, which happens to be the case for all records in o. It would seem, then, that LS_1 is a weak specification for P_1 (recall Task 1 in Table V).

Table VI(b) shows the results of the symbolic search. With $size(i) = 10$ and $max\_time = 300$, the solver identifies 81 matches. For the second topology discussed above, the concrete search required that $n = 1$ for *truncate* modules. When n is made symbolic, pipes that held other values now match. Similarly with the third topology, the substring can be anything that exists just in o. Here, relaxing the string and integer values yields *close enough* matches that can be reused with few modifications (i.e., instantiating the symbolic

[5] Artifacts are available: cse.unl.edu/~kstolee/fse2012/

TABLE VI
RESULTS FROM ARTIFACT STUDY

(a) Example 1: Concrete

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	2	0	+95.60	0	+217.20	17	125.39
9	2	0	+82.93	0	+180.97	17	118.10
8	2	0	+73.74	16	50.85	17	79.23
7	2	0	+67.09	16	44.53	17	60.04
6	2	4	21.83	20	20.33	21	28.49
5	1	31	8.01	31	12.04	32	12.88
4	1	31	5.51	32	7.62	32	7.97
3	1	35	2.89	36	4.39	36	4.31
2	1	49	1.65	49	1.70	49	1.63
1	1	2775	0.26	2775	0.19	2775	0.19

(b) Example 1: Symbolic

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	2	0	+275.47	0	+1,141.17	81	224.90
9	2	0	+237.50	21	105.34	100	154.34
8	2	0	+226.15	24	96.16	100	131.66
7	2	0	+215.84	24	86.12	100	95.40
6	2	19	25.51	84	30.77	106	66.93
5	1	127	13.70	276	21.02	302	65.44
4	1	205	9.37	277	16.13	324	48.29
3	1	337	2.77	347	4.89	357	8.68
2	1	359	1.66	361	1.97	361	1.97
1	1	2821	0.39	2822	0.42	2822	0.42

(c) Example 2: Concrete

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	3	0	+111.52	0	+189.54	1	217.72
9	2	0	+118.78	0	+200.14	1	172.62
8	2	0	+110.70	1	133.02	1	132.75
7	2	0	+98.67	1	92.90	1	93.99
6	2	0	+71.47	1	69.07	1	68.45
5	2	0	+69.97	1	59.93	1	60.30
4	2	1	37.67	1	34.47	1	36.31
3	2	1	19.87	1	20.45	1	19.53
2	1	2	5.04	2	4.91	2	4.87
1	0	85	0.55	85	0.61	85	0.60

(d) Example 2: Symbolic

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	3	0	+213.39	0	+508.44	0	+1,232.53
9	2	0	+214.20	0	+885.37	20	522.36
8	2	0	+197.00	1	456.07	22	366.85
7	2	0	+191.13	1	392.88	23	258.99
6	2	0	+174.82	1	304.08	23	176.28
5	2	1	89.47	1	285.13	30	90.29
4	2	1	81.53	24	63.82	36	44.87
3	2	62	11.64	93	15.82	93	15.87
2	1	99	8.68	99	8.92	99	9.02
1	0	373	0.54	373	0.55	373	0.56

(e) Example 3: Concrete

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
15	9	0	+115.19	0	+298.34	0	+645.40
14	9	0	+122.05	0	+298.73	3	371.36
13	9	0	+126.23	0	+301.20	3	375.66
12	8	0	+109.60	0	+239.03	3	418.04
11	7	0	+111.95	0	+232.49	3	241.06
10	6	0	+93.42	3	106.62	3	113.03
9	6	0	+114.83	3	113.28	3	136.55
8	6	0	+94.27	3	112.45	5	85.19
7	5	6	33.36	9	46.87	9	46.69
6	4	2	41.20	5	36.82	5	36.42
5	3	8	27.48	13	22.16	13	21.98
4	3	13	11.49	13	12.42	13	12.30
3	3	2588	2.01	2591	2.15	2591	2.12
2	2	2596	0.62	2596	0.62	2596	0.61
1	1	2738	0.22	2738	0.22	2738	0.22

(f) Example 3: Symbolic

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
15	9	0	+342.80	0	+1,048.63	16	598.85
14	9	0	+330.77	0	+936.25	13	687.57
13	9	0	+344.65	0	+1,167.39	18	502.54
12	8	0	+312.14	13	175.07	20	317.64
11	7	0	+300.17	15	132.16	20	268.32
10	6	0	+302.57	15	119.49	20	268.99
9	6	0	+256.49	12	133.97	23	243.36
8	6	17	143.00	203	22.48	208	69.17
7	5	172	11.07	203	15.47	213	41.25
6	4	185	8.37	203	12.50	213	27.65
5	3	232	6.90	318	13.98	349	24.35
4	3	243	5.12	344	11.66	349	14.24
3	3	2679	3.08	2689	3.91	2689	3.88
2	2	2687	1.08	2689	1.18	2689	1.17
1	1	2821	0.41	2822	0.42	2822	0.42

(g) Example 4: Concrete

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
5	1	1	32.19	1	32.17	1	32.81
4	1	1	26.36	1	26.10	1	25.71
3	1	1	17.72	1	17.23	1	17.26
2	0	79	1.28	79	1.19	79	1.19
1	0	85	0.52	85	0.57	85	0.57

(h) Example 4: Symbolic

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
5	1	3	60.99	80	35.44	89	51.82
4	1	10	39.02	90	29.27	93	42.02
3	1	68	13.47	97	16.38	99	25.36
2	0	370	1.13	373	1.54	373	1.53
1	0	373	0.52	373	0.54	373	0.54

(i) Example 5: Concrete

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	7	0	+94.45	0	+162.40	1	146.75
9	6	0	+73.29	0	+133.65	1	114.07
8	5	0	+71.17	0	+126.69	1	88.76
7	4	0	+46.23	0	+73.11	1	69.61
6	3	0	+33.12	0	+61.49	1	54.18
5	2	0	+53.16	6	45.83	7	36.54
4	2	6	22.70	7	20.72	7	20.90
3	2	6	13.85	7	10.42	7	10.57
2	1	20	2.64	21	2.31	21	2.32
1	0	71	0.60	72	0.56	72	0.57

(j) Example 5: Symbolic

Sizes		5sec.		30sec.		300sec.	
i	o	Pipes	TFS	Pipes	TFS	Pipes	TFS
10	7	0	+196.58	0	+472.42	1	609.54
9	6	0	+178.94	0	+395.49	0	+829.41
8	5	0	+167.15	0	+405.29	1	432.35
7	4	0	+127.60	0	+196.03	1	557.31
6	3	0	+140.03	0	+220.42	0	+953.57
5	2	0	+159.78	8	173.59	61	117.22
4	2	1	78.63	60	44.90	62	86.57
3	2	13	24.89	97	15.23	99	22.18
2	1	104	6.53	105	7.12	105	7.14
1	0	372	0.53	373	0.58	373	0.56

variables with concrete values in the model found by the solver and setting the URLs to those specified by the developer).

Example 2: Early Identification of Match. In the concrete execution (Table VI(c)), only one pipe, P_2 (Figure 5(b)), satisfies LS_2. This pipe is also the only match when $size(i) = 3$ and $max\_time = 5$, and takes on average only 20 seconds to be identified. Given that same input size and solver time in the symbolic search, there are 62 matches.

In the symbolic search (Table VI(d)), there are no matches for $size(i) = 10$ and $max\_time = 300$. Three pipes return $unknown$ because it takes longer than 300 seconds for the solver to find a satisfiable model for pipes like P_2. However, tuning the lightweight specifications so $size(i) = 9$ with $max\_time = 300$ yields 20 matches that may be *close enough*; these have two topologies. One topology is P_4 (Figure 5(d)), since n in the *truncate* and *tail* modules can be set to 3 and 2, respectively, to grab $i[1, 2]$ and form the output. The second involves a *filter* module that asserts substring containment on the descriptions or titles of all records in o.

Example 3: Symbolic Sometimes More Efficient. For the concrete search (Table VI(e)), no matches are found when $size(i) = 15$ and $max\_time = 300$, but three pipes return $unknown$, including P_3 (Figure 5(c)). In the symbolic search (Table VI(f)), 16 matches are found with $size(i) = 15$ and $max\_time = 300$, including P_3. These pipes all have similar topologies, containing at least three input paths, each with a *truncate* module prior to a *union*, then possibly a *sort* prior to the output. The symbolic matches are possible because the solver finds a value for the *truncate* (i.e., $n = 3$) that yields the desired output. Even more interesting is that for $4 \leq size(i) \leq 8$ and $max\_time = 30$, or for $6 \leq size(i) \leq 8$ and $max\_time = 300$, the symbolic search identifies a match faster than the concrete search, likely because there are so many more symbolic matches.

If we consider the concrete search with $size(i) = 10$ and $max\_time = 30$ (just two of the three URLs, shown in Table VI(e)), a match is found within two minutes (106 seconds), and includes P_3. The same is true when $size(i) = 5$ and $max\_time = 30$ (just one of the three URLs), although in this case there are 13 matches returned in about 22 seconds. For some specifications like LS_3, an input/output pair can easily be broken down into smaller pieces that will be found satisfiable faster, and can possibly be generalized.

Example 4: Symbolic Often Trivial. In the concrete search (Table VI(g)), only one pipe, P_4 (Figure 5(d)), is a match for $size(i) = 5$ and $max\_time = 300$. This pipe is found quickly and could be identified in about 32 seconds given $max\_time = 5$. The symbolic search (Table VI(h)) yields more results. With $max\_time = 30$, 80 matches are found for $size(i) = 5$, and the first is found in 35 seconds. In addition to P_4, other matching pipes assert equality and substring containment over the title and/or description of the records. Since there is only one record in the output for $3 \leq size(i) \leq 5$, the pipes that assert such string properties are trivially satisfiable by assigning the symbolic string the value of the output record's title or description.

Example 5: Concrete and Symbolic Equal. In this example, we found that for the number of matches found, the symbolic search was not much better than the concrete search for larger input sizes. For $6 \leq size(i) \leq 10$ in the concrete search (Table VI(i)) and $max\_time \leq 30$, no matches are found. When $max\_time = 300$, one match is found, and it is P_5 (Figure 5(e)). On average, this pipe was identified within 55 seconds for $size(i) = 6$ and $max\_time = 300$. For symbolic search, the same pipe is identified as satisfiable for $7 \leq size(i) \leq 10$ and $max\_time = 300$. With smaller input sizes that come from just the first URL (i.e., $1 \leq size(i) \leq 5$), there are many matches found, a phenomenon we also observed with Example 2. In both cases, considering just $i[0, 4]$ yields many results, but as the size of i increases to include the second URL ($i[5, 9]$ in Example 2, and $i[5, 9]$ in Example 5), o also increases to include the additional records. For Example 2 and Example 5 only, the ordering of records in i is not preserved in o, which seems to make even symbolic matches hard to find.

V. Scope and Applicability

At this point, we have shown how our approach can be instantiated to support developers of Yahoo! Pipes. We now discuss our approach in a broader reuse context and the extent to which it may generalize to other languages.

Reuse. Our approach to semantic search has clear ties to code reuse. In the reuse process, there are two primary activities: finding and integrating. Our approach thus far has focused on the first part, finding, but we recognize that it has potential to be useful with integration. For effective reuse, scope and dependencies must be understood for developers to effectively integrate code [6]. Some recent work aims to assist the developer with integrating new code by matching the structural properties (e.g., method signature, return types) of their development context with the reused code [3], [10] or by querying APIs by keyword [19]. While these approaches guarantee structural matching, the behavior of the newly integrated code may not be well understood. In our approach, we guarantee the behavior of the matched code given LS, and structural properties of the code within the developer's context could be leveraged to generate $i \in (i, o)$.

Generalizability. We start by pointing out that the current instantiation of the approach is easily generalizable to other mashup languages based on predefined configurable modules that obtain and manipulate data (e.g., [4], [12]). For the approach to work on these languages and their repositories, it would just be necessary to re-map the modules of each language to the constraints defined in Table IV.

We have also started to explore the application of the approach for data manipulation languages like `SQL`, where the specifications take the form of a populated database (input) and the desired columns, rows, or a subset thereof (output). We provided evidence in Section II on how developers' questions in this domain are often stated in terms of desired behavior using sample tables and records, making it suitable for the proposed approach. We also performed an empirical study in

the same context described in Section IV-B to assess how quickly and correctly developers could provide input/output samples.[6] Each of the five tasks was completed by an average of 69 participants. On average, the tasks were performed correctly 92% of the time and took an average time of 78 seconds, indicating that providing input/output for this domain can be done correctly and quickly in most cases. Much of the SQL language has already been mapped onto a rich background theory used to generate input tables and parameter values given a SQL query [29]. This work is complementary to our efforts in supporting SQL, where the queries are encoded as programs (C_P) and the input/output tables as specifications LS. As such, the application of the approach to this domain seems within reach.

For our approach to be cost-effective in the context of more traditional programming languages like Java, we envision the component characterization could take the form of pre/postconditions or invariants. If such conditions are provided in annotation languages like JML [17], then the approach would only need to encode JML specifications as constraints. The richness of such specifications, including the presence of side-effects, and their quality will be the core challenges as we push our approach in that direction.

VI. Conclusion

The need for more powerful code search capabilities is evident as developers attempt to leverage large open code repositories. In this work, we have defined a semantic approach to search that matches lightweight specifications against programs encoded as constraints. We have shown that the lightweight specifications can be accurately and efficiently defined, that with suitable encodings matching programs can be found, and that refining the problem definition can tune the results in the presence of constraints that are too strong or too weak. With the success of our preliminary instantiation on the Yahoo! Pipes language, we are working to extend and assess our approach in a broader context.

Acknowledgments

Thanks to Daniel Dobos for his contributions. This work is supported in part by NSF Award CCF-0915526, NSF GRFP under CFDA-47.076, and AFOSR Award #9550-10-1-0406.

References

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

[2] S.-C. Chou, J.-Y. Chen, and C.-G. Chung. A behavior-based classification and retrieval technique for object-oriented specification reuse. *Softw. Pract. Exper.*, 26(7):815–832, July 1996.

[3] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 214–225, New York, NY, USA, 2008. ACM.

[4] DERI Pipes. http://pipes.deri.org/, August 2009.

[6]Task descriptions and detailed findings are online: cse.unl.edu/~kstolee/fse2012/

[5] G. Fischer, S. Henninger, and D. Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pages 318–328, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12:17–26, November 1995.

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

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

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

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

[11] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45 –52, sept.-oct. 2008.

[12] IBM Mashup Center. http://www.ibm.com/software/info/mashup-center/, August 2009.

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

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

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

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

[17] G. T. Leavens. Tutorial on jml, the java modeling language. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 573–573, New York, NY, USA, 2007. ACM.

[18] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Int'l Conf. on Soft. Eng.*, 2011.

[19] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25, New York, NY, USA, 2010. ACM.

[20] Amazon Mechanical Turk. https://www.mturk.com/mturk/welcome, June 2010.

[21] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. *Ann. Softw. Eng.*, 5:349–414, Jan. 1998.

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

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

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

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

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

[27] K. T. Stolee and S. Elbaum. Refactoring pipe-like mashups for end-user programmers. In *International Conference on Software Engineering*, 2011.

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

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

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

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