Toward Semantic Search via SMT Solver

Kathryn T. Stolee, 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 reuse. 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 syntactic searches the query cost is quite low, but the results are often irrelevant, so the examination cost is high and matches may be missed. 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. We propose an approach for semantic search in which developers specify lightweight specifications and an SMT solver identifies matching programs from a repository. A program repository is automatically encoded offline so the search is efficient. Programs are encoded at various abstraction levels to enable partial matches when no, or few, exact matches exist. We instantiate this approach on a subset of the Yahoo! Pipes mashup language. Preliminary results show promise for the feasibility of the approach.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

Semantic code search, SMT solvers, lightweight specifications

1. INTRODUCTION

Developers are increasingly turning to search to find solutions to their programming problems, and general search engines are the most common and often effective way to find suitable code [8]. 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. Specialized search engines (e.g., Koders, Krugle, Merobase) also incorporate filtering (e.g., by programming language) and program syntax into the query to better guide the search. Other approaches add natural language processing to increase the potential matching space [2]. Such syntactic approaches have a low query cost and may be effective in identifying functionality that can be captured with structural components like function names, but cannot capture behavior that is not tied to source code syntax or documentation. Some more sophisticated approaches have improved precision by incorporating semantics, but are not common in practice. They often require developers to incur additional costs, such as writing complex specifications [11]. Partial specifications using test cases [6, 7] mitigate that cost, but are too specific, meaning that *close* matches may be ignored.

In this work we propose a novel approach to semantic search. First, it allows developers to specify a lightweight, incomplete specification in the form of input/output pairs to keep the query cost low. Second, instead of having the search engine just index program syntax, it encodes programs as constraints representing behavior, so the matches are based on semantics. 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 identify *close* matches. **This idea of code search via SMT solver is the core of the new idea**. We do not fully evaluate it, but we present a preliminary instantiation of the approach and evaluation on a subset of the Yahoo! Pipes dataflow mashup language. Many questions remain regarding its feasibility in a broader context, and we hope to obtain feedback from the forum on many of the pending challenges and suggestions on how to address them.

2. SAMPLE APPLICATION OF APPROACH

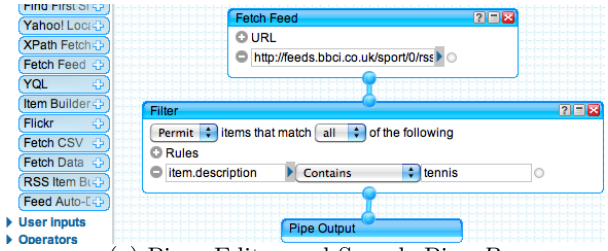
Here, we introduce the initial instantiation of our approach and the targeted domain, Yahoo! Pipes.

2.1 Target Domain

Since 2007, thousands of users have programmed with Yahoo! Pipes, forming a public repository of over 100,000 artifacts [5]. Motivated by the popularity of Yahoo! Pipes, the number of available artifacts, the components interfaces that make them amenable for encoding, and the limitations of existing search in this domain (discussed later), we instantiate our approach on a subset of this language.

To program these mashups, developers use the Pipes Editor, a proprietary development environment accessible through the browser, shown in Figure 1(a). Mashups are composed by dragging and dropping predefined modules (e.g., a *fetch* module, a *filter* module) from the module library on

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.



(a) Pipes Editor and Sample Pipe P

Module	Type	Constraint Def
1: fetch	equality	$out1 = i$
wire(1,2)	equality	$in2 = out1$
2: filter	inclusion	$(contains(in2, r) \wedge substr(descr(r), s)) \rightarrow contains(out2, r)$
	exclusion	$contains(out2, r) \rightarrow contains(in2, r)$
	order	...
wire(2,3)	equality	$in3 = out2$
3: output	equality	$in4 = o$

(b) Constraints for Pipe, C_P

Figure 1: Example Constraint Mapping

the left onto the canvas on the right. Module behavior is configured by setting their fields (e.g., URLs, strings). Control and data flow are defined with wires that connect the modules. Data flow from top to bottom, forming a directed graph with one or more sources and one sink. Data are generally input to the pipe by fetching RSS feeds from URLs, and the output is a unified and modified list of the records. A record is a data structure with name/value pairs. We refer to the names as *fields*; typical records have at least these five fields: *title*, *description*, *author*, *date*, and *link*. In Figure 1(a), the *fetch* module provides a list of records from the URL, the *filter* module retains records with "tennis" in the description, and the *output* module is the sink of the program.

To illustrate the challenges for Yahoo! Pipes developers using existing search mechanisms, we performed five searches for mashups by querying for the URLs used in each. The number of matches can be staggering (often more than 1,000) but not surprising as many mashups include common websites. The average number of relevant results among the top ten, determined by behavior, is 0.9. The other built-in search capabilities do not fare better; searching by language constructs retrieves more results and requires implementation knowledge, and searching for tags is dependent on the community's categorization of their artifacts.

2.2 Implementation

We now describe an instantiation of our proposed search in the Yahoo! Pipes domain, which involves three steps: defining a specification (query), encoding programs to be searched (indexing), and finding matches via an SMT solver.

Lightweight Specifications. With Yahoo! Pipes, the input specification is one or more URLs that reference RSS feeds and provide lists of records to a pipe, such as the *fetch* module in Figure 1(a). Each list of records, identified by a URL, is assigned to input i. Next, the developer chooses records to be retained in the output o, giving form to the lightweight specification $(i, o) \in LS$. Next, LS is automatically transformed into constraints by the framework and later used as the search query. For example, if a record r is at index 2 in i, and index 1 in o, constraints would assert, $i[2] = r \wedge o[1] = r$. To assign a string value to the title field of r, a constraint would assert, $title(r) =$ "A clever title."

Encoding. Encoding a pipe is an automated process that performs a code transformation, mapping the program constructs to constraints; in this domain, it requires no annotations by the developer. The program components have well-defined interfaces and behaviors, and since the average pipe has about 8 modules, it is natural to map each module and wire onto constraints.

We use the example in Figure 1 to illustrate a pipe's encoding, with the structure, P, in Figure 1(a) and constraints, C_P, in Figure 1(b). For this data-flow language, module constraints are classified as inclusion, exclusion, and order, and wires are equality constraints. *Inclusion* constraints ensure completeness; in the example, the inclusion constraint for the *filter* module ensures that if a record r is in the input to the module and $descr(r)$ contains s, then r is in the output from the module. *Exclusion* constraints ensure precision; if a record r is in the output of the *filter* module, then r in the input. *Order* constraints (omitted from Figure 1(b) for brevity) ensure that records are ordered properly in the list. That is, if two records exist in the *filter* module's input and output, then their ordering is the same in both lists. *Equality* constraints ensure the output of a source module is equivalent to the input of the destination module.

The pipe in Figure 1(a) illustrates only three modules, yet our instantiation includes a larger subset of the language representing the most common constructs, covering five of the top ten most used modules. This subset performs filter (the *filter* module), permute (the *sort* module), merge (the *union* module), copy (the *split* module), and head/tail (the *truncate* and *tail* modules) operations on lists of records. (The *fetch* and *output* modules are also among the supported modules). Encoding the supported operations requires six data types: characters $\mathcal{C}$, strings $\mathcal{S}$, integers $\mathcal{I}$, booleans $\mathcal{B}$, records $\mathcal{R}$, and lists $\mathcal{L}$. Implementing this language subset requires many complex constraints, for example, the *filter* module requires support for substring identification. The constraint mappings for the supported subset of the Yahoo! Pipes language are available in a technical report [9].

Our encodings also support two levels of strength for the constraints on the field values, concrete and symbolic, where the latter permits *close* matches when exact matches cannot be found or do not exist. For the *filter* module in Figure 1, a concrete constraint requires $substr(descr(r), \text{"tennis"}) = true$, whereas a symbolic constraint requires $substr(descr(r), s) = true$ for some string s. The inclusion constraint in Figure 1(b) is symbolic on the description field value.

Solving. Once all programs have been encoded, we can perform the search using LS as the query. An SMT solver will be invoked, pairing each encoded program with LS to determine which programs, if any, match LS. For the matches, inserting the URL into the *fetch* module will result in the desired output. To perform the search, we use the Z3 SMT Solver v3.2 from Microsoft Research [10].

3. GENERALIZING THE APPROACH

The previous section showed how our approach can be applied to Yahoo! Pipes. Here, we describe more generally the building blocks of the approach, illustrated in Figure 2.

3.1 Lightweight Specifications

In this approach, the search query takes the form of lightweight specifications that characterize the desired be-

havior of the code (*Lightweight Specifications* in Figure 2). These specifications, LS, are represented as input/output pairs, and take different forms depending on the domain. The size of LS (i.e., the number of pairs) 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 behavior.

It is important that the query cost be reasonable for the developer. Providing input/output pairs requires more effort and may be more fault prone than providing keywords, so the effectiveness of the search must compensate for the extra effort. We have observed that developers already use examples in forums when asking for help; in a preliminary study of 100 questions about SQL from `stackoverflow.com`, for example, 76 provided examples of desired behavior [9].

3.2 Encoding

Encoding is analogous to crawling and indexing performed by information search engines. Offline, a repository (*Code Repository* in Figure 2) is crawled to collect programs. In an automated process, these programs are encoded as constraints (*Encoding*, analogous to indexing) and stored in a repository (*Constraint Database*). The encoding process uses a mapping of program constructs to constraints. Creating this mapping is one of the key challenges to this approach, as the levels of granularity and strength for encoding are important considerations for the cost of search.

The finest granularity corresponds to encoding the whole program behavior but could result in constraint systems that cannot be resolved in a reasonable amount of time. At the coarsest granularity the encoding would capture none of the program behavior, which could return a plethora of irrelevant matches. With the encoding process, generalizability is certainly a concern. Section 2.2 presents the list of operations supported by our Yahoo! Pipes instantiation. Other languages or parts of languages that perform similar list and string manipulations could be covered by this, but further study is needed to understand the extent to which our current encodings extend to other domains.

The strongest encodings use a more concrete representation of the constraints, but may not return enough matches; weaker encodings relax some of the constraints and treat variables as symbolic to allow *close* matches to be identified. Like previous approaches that relax matching on pre/postconditions [11], we exploit the fact that most languages contain constraints over multiple data types (e.g., strings, integers, booleans) and relax matching on specific variables (e.g., the string "tennis" from Section 2.2). For richer languages like Java, these current relaxations may not be sufficient. Certain constructs like loops will likely be approximated by necessity, and the question of how to relax or tighten those encodings remains. In general, finding sufficient encodings, abstractions, and relaxations is one of the most interesting, albeit difficult, challenges in this approach.

3.3 Solving

Solving (*SMT Solver* in Figure 2) is analogous to matching in search engines. $Solve(C_P \wedge LS) \rightarrow (sat, unsat, unknown)$ returns *sat* when a satisfiable model is found (i.e., a match) or *unsat* when no model satisfies the constraints (i.e., no match). When the solver is stopped before it reaches a conclusion or it cannot handle a set of constraints, *unknown* might be returned. In the case of symbolic constraints, if the solver returns *sat*, extracting the satisfiable model reveals values for the symbolic variables to guide the creation of a suitable program P' from a matching program P.

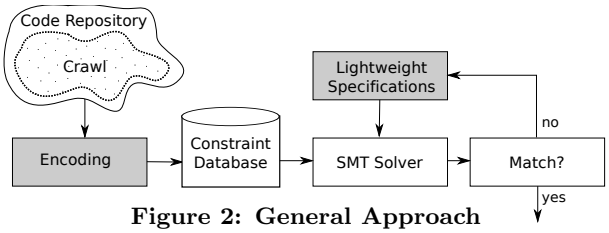


Figure 2: General Approach

The efficiency of this process is determined by the constraints' complexity, the input size, and the solver speed. The constraint complexity can be controlled, to some degree, by changing the granularity and strength of the encodings (e.g., concrete or symbolic). Recall is dependent on how long the solver is allowed to run; the longer the allowed solver time, the higher the recall. For precision, the programs returned by the search for a given specification will always be relevant as defined by LS, by design, and so the precision of the search is 100% if matches are found.

4. FEASIBILITY STUDY

Section 2.2 presents an instantiation of the approach in the Yahoo! Pipes language, which we now evaluate.

Study Setup. From a pool of 2,859 Yahoo! Pipes, we selected five example pipes,[1] executed each to derive LS, and then performed searches using the specifications.[2] Bounds of 100 characters on the strings (i.e., the record titles and descriptions) and five records per URL in the input list were imposed, though these are configurable.

In the search, two factors are manipulated for each example: the strength of the field encodings (*Concrete* and *Symbolic*) and the maximum runtime for each call to $Solve(C_P \wedge LS)$ ($maxT = \{5, 300\}$ seconds). For each example pipe and combination of factors, we search the pool and report the number of matches, $precision = \frac{relevant \cap retrieved}{retrieved}$, $recall = \frac{relevant \cap retrieved}{relevant}$, and *Time to First Sat* (TFS), which represents the time until a match is found. (averaged over 250 runs). Relevant results are those that eventually return *sat* for LS and a given encoding strength, if allowed enough solver time. Retrieved results are those for which the solver returns *sat* within $maxT$. The TFS reported represent sequential iteration of the programs, but this process can be parallelized to improve efficiency. Our data were collected under Linux on 2.4GHz Opteron 250s with 16GB of RAM.

Results. In Table 1, the results for the concrete program encodings are presented first, followed by the symbolic encodings. Each example is reported in a row. The first three columns show the example number (Ex) and size, in terms of number of records, of the input and output in LS. The following columns show the number of matching pipes that returned *sat* (#), recall (*Rec.*), and TFS given $maxT = 5$ and 300. A '+' before the time means that no satisfiable result was found, so the time displayed is a lower bound in those cases. If any matches are found, $precision = 1.00$ since all results are relevant, hence precision is not reported in the

[1] In clustering the pool by structural similarity, these pipes were selected from the median five clusters.

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

Table 1: Preminimary Results from Study

Concrete

Ex	*LS*		*5sec.*			*300sec.*		
	i	*o*	#	**Rec.**	**TFS**	#	**Rec.**	**TFS**
1	10	2	0	0.00	+95.60	17	1.00	125.39
2	10	3	0	0.00	+111.52	1	1.00	217.72
3	15	9	0	0.00	+115.19	0	0.00	+645.40
4	5	1	1	1.00	32.19	1	1.00	32.81
5	10	7	0	0.00	+94.45	1	1.00	146.75

Symbolic

Ex	*LS*		*5sec.*			*300sec.*		
	i	*o*	#	**Rec.**	**TFS**	#	**Rec.**	**TFS**
1	10	2	0	0.00	+275.47	81	1.00	224.90
2	10	3	0	0.00	+213.39	0	0.00	+1,232.53
3	15	9	0	0.00	+342.80	16	0.62	598.85
4	5	1	3	0.03	60.99	89	1.00	51.82
5	10	7	0	0.00	+196.58	1	1.00	609.54

table. 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*.

In the concrete search given $maxT = 300$, all relevant pipes are found with the exception of Ex 3, in which no relevant pipes are found. For Ex 3, it takes longer than 300 seconds for the solver to return *sat* for each of the three relevant pipes; the large size of *LS* may play a role in the runtime. Only Ex 4 yielded any results with $maxT = 5$.

In the symbolic search with $maxT = 5$, 3% of the relevant results are found for Ex 4, but the recall was 0.00 for the others. Setting $maxT = 300$ yeilds results for all searches except Ex 2, which finds none of the three relevant pipes. In Ex 3, only 16 of the 26 relevant pipes are identified.

Overall, the longer the solver can run, the higher the recall, and the more relaxed the encodings, the higher the number of relevant pipes. Yet, for some *LS*, even 300 seconds might be too short to find a relevant pipe. The overall efficiency can be increased by introducing concurrency in the search, which might allow us to explore longer solver times.

An interesting observation not immediately apparent from Table 1 is that when matches are not found (i.e., the search is stopped at $maxT$), the solver returns *unknown* for any relevant pipes. Treating the set of *unknown* pipes as actual results boosts the recall to 1.00 when $maxT = 300$ while maintaining precision at 1.00. Recall also boosts to 1.00 for $maxT = 5$, however, we lose precision as some *unsat* pipes may return *unknown* without sufficient solver time. Further study is needed to identify the optimum $maxT$ for precision and recall, and also to compare the performance of our search to state-of-the-art and state-of-the-practice search engines.

5. RELATED WORK

Early work in semantic search required developers to write complex specifications of behavior using first-order logic or specification languages (e.g., [4, 11]), which can be expensive and error-prone for the programmer. This cost can be reduced by using incomplete behavioral constructs, such as test cases to describe behavior [6, 7], but these approaches require that the code be executed, which is not scalable and cannot identify approximate behavioral matches. Some previous work has proposed the use of semantic networks to identify approximate matches [1], but it requires manual annotations on the code.

Our use of symbolic encodings broadens the search space beyond what is defined by the program semantics, in essence providing a framework for program synthesis. Previous work in program synthesis also makes use of solvers, but derives a function mapping an input to an output [3]. The key difference is that our approach uses existing programs as skeletons to constrain the search, which makes it more scalable.

6. CONCLUSION

We have defined a semantic approach to search that uses an SMT solver to match lightweight specifications against programs encoded as constraints. We have shown that with suitable encodings matching programs can be found, and that using symbolic constraints can identify matches that are *close enough* to be modified for the provided specifications.

Our preliminary instantiation on the Yahoo! Pipes language is promising, but many challenges remain. Selecting an appropriate level of granularity for encoding is a key challenge, and complexities such as loops and side effects remain unaddressed. Additionally, approximating behavior is a challenge for finding *close* results. Thus far, closeness has been identified by treating string and integer values as symbolic in the search. More sophisticated approximations will be necessary as we extend this work in a broader context.

Acknowledgments

This work is supported by in part NSF Award CCF-0915526, NSF GRFP CFDA-47.076, and AFOSR #9550-10-1-0406.

7. REFERENCES

[1] 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.

[2] 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.

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

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

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

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

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

[8] 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.

[9] K. T. Stolee and S. Elbaum. Solving the Search for Suitable Code: An Initial Implementation. Technical report, University of Nebraska-Lincoln, June 2012.

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

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