University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Conference and Workshop Papers | Computer Science and Engineering, Department of

1-1-2012

Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications

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

Follow this and additional works at: http://digitalcommons.unl.edu/cseconfwork

Part of the Computer Sciences Commons

Stolee, Kathryn T., "Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications" (2012). *CSE Conference and Workshop Papers*. Paper 202.
http://digitalcommons.unl.edu/cseconfwork/202

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Conference and Workshop Papers by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

2012 34th International Conference on Software Engineering (ICSE)
Digital Object Identifier: 10.1109/ICSE.2012.6227034

Finding Suitable Programs: Semantic Search with Incomplete and Lightweight Specifications

Kathryn T. Stolee
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE, USA, 68508
kstolee@cse.unl.edu

***Abstract*—Finding suitable code for reuse is a common task for programmers. Two general approaches dominate the code search literature: syntactic and semantic. While queries for syntactic search are easy to compose, the results are often vague or irrelevant. On the other hand, a semantic search may return relevant results, but current techniques require developers to write specifications by hand, are costly as potentially matching code need to be executed to verify congruence with the specifications, or only return exact matches. In this work, we propose an approach for semantic search in which programmers specify lightweight, incomplete specifications and an SMT solver automatically identifies programs from a repository, encoded as constraints, that match the specifications. The repository of programs is automatically encoded offline so the search for matching programs is efficient. The program encodings cover various levels of abstraction to enable partial matches when no or few exact matches exists. We instantiate this approach on a subset of the Yahoo! Pipes mashup language, and plan to extend our techniques to more traditional programming languages as the research progresses.**

***Keywords*-semantic search; program composition; code reuse; SMT solvers; constraints**

I. Introduction

Programmers face many challenges when approaching a new problem, including learning new languages, APIs, and environments. Often, the problem being solved is not novel. With the increasing number of large and publicly accessible code repositories [1], the existence of similar, if not exact solutions is likely. Yet, the costs of code reuse in terms of finding and integrating code often outweigh the benefits [2].

Search engines are the most common way for developers to find suitable code [3], but frequently return irrelevant results that still must be evaluated by the developers. Semantic search by specification is effective for finding relevant code but often requires developers to write complex specifications (e.g., [4] [5]). Other semantic search techniques that support partial specifications, such as test cases (e.g., [6] [7]), are costly and unscalable since code needs to be executed to identify matches, and also unable to identify approximate matches when no code meets the partial specification.

To address these limitations, we propose an approach for semantic search in which developers *specify lightweight, incomplete specifications in the form of input/output pairs and/or partial program fragment*, and incrementally refine the specifications as needed. Instead of executing the code to determine a match, we *automatically encode repositories of existing programs as constraints* and use an SMT solver to identify code in the repository that matches the specifications. To identify *close enough* matches when no exact matches exist, we relax the matching criteria using lattices over levels of abstraction. The programs in the repository are automatically encoded offline, so the only costs are defining the lightweight specifications and solver time.

In this paper, we describe our approach to finding suitable code by solving with incomplete specifications. The expected contributions are:

- Approach for semantic search via lightweight specifications and program fragments using SMT solvers that provides full, partial, and composed matches
- Prototype(s) to automatically translate programs and specifications into constraints
- Evaluations of the aforementioned approach in at least two programming languages

A. Related Work

Queries for syntactic code search are easy to specify, and more sophisticated techniques leverage natural language processing to identify matches [8], yet they ignore the structure and behavior of code, so results may not be relevant. Early work in semantic search involved specification matching over component signatures and behavior, expressed using pre/postconditions written in first-order logic [4] [5]. While the results from these searches may be relevant, writing specifications is non-trivial, error-prone, and often requires a developer to learn a specification language.

To increase the practicality of semantic search, performing transformations on the code (e.g., reordering parameters) can make code relevant to more queries, but explodes the search space [6]. Other techniques allow developers to create partial specifications in the form of input/output pairs [9], test cases [7], constraints, or a combination thereof [6]; potential code matches are found by executing the code. While this is effective in finding relevant code, it is costly and unscalable as the code must be executed to identify matches. Further,

these methods are only effective if the repository contains an *exact match* given the specification, albeit in some cases after a transformation.

Some research in program synthesis aims to synthesize code based on specifications, using input/output pairs of objects and a solver to generate a program that meets the specification [10]. The difference from our approach is that we use the solver to find a match against existing programs which makes it much more scalable.

B. Initial Scope

Our ultimate goal is to apply and evaluate our approach on multiple languages (see Section II-C). Here, we illustrate an application of our approach to the domain of web mashups, using the Yahoo! Pipes [11] language for evaluation.

Yahoo! Pipes is a popular language that allows users to create mashups within a browser, boasting over 90,000 users [12] with a public repository of over 100,000 pipes programs (an example is shown in Figure 1). This component-based dataflow language can access multiple data sources (e.g., RSS feeds), manipulate the data (e.g., *filter*, *sort*, *concatenate*), and create a unified output. The input to the programs are data sources, typically RSS feeds, that appear at the top of the programs (there are two sources in Figure 1). The output is a list of records (e.g., the records that reach the *output* module at the bottom). Within this domain, we aim to help programmers find suitable code for a variety of scenarios, including the following:

1) Order articles from two blogs by publication date (involves *sorting*; a solution is shown in Figure 1)
2) Retrieve the five most-recent articles about tennis from three different web data sources (involves processing each data source individually with *sort*, *filter*, and *head* operations, and then *concatenation*)
3) Use US census information to order the US states by population (involves *extraction* of population information into a new variable, *reformatting* it with a regular expression, and *sorting* based on that variable)

II. Goals and Approach

The research problem we aim to address is to help programmers find suitable code by narrowing the gap between code that is desired and code that is available. In our proposed solution, programmers find desired code by specifying lightweight, incomplete specifications. With these specifications (automatically encoded as constraints) and an encoded code repository of programs (Section II-B2), an out-of-the-box SMT solver decides which programs match the specifications. If the solver returns no or few matches, we propose relaxations over the specifications and program encodings that help identify 1) a program that is *close enough* and can be *adapted* to the behavior specified, or 2) a set of programs that, when composed, match the partial specifications provided by the developer.

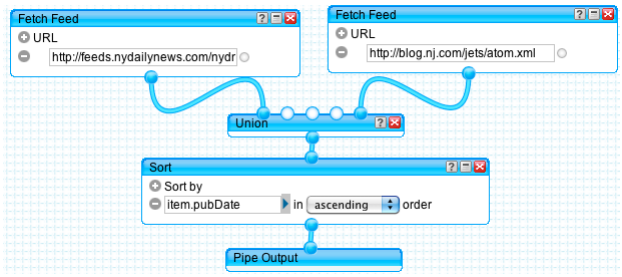


Figure 1. Pipe Solution for First Scenario

A. Research Methodology

Our research methodology generally follows a cyclic workflow in which we define an overall approach to address our research problem (as defined in Section II), select a domain (Section II-C), adapt parts of the techniques that are domain-specific (detailed in Section II-B), prototype an implementation, and perform an evaluation (Section II-D).

In the first iteration, we started with a domain, where, based on our experience, finding suitable code is challenging since syntactic matches are uncommon [13]. We built a proof-of-concept to encode Yahoo! Pipes programs (Section II-B2), which led to a prototype that shows SMT solvers can be used to identify matching programs from an input/output pair (Section II-B3). In a small evaluation (Section II-D), we observed that there were often too few matches for a given specification, leading us toward subsequent iterations in which we refine the techniques.

In the second iteration, we defined, prototyped, and evaluated lattices for relaxing the constraints representing programs for the case when too few matches are found (Section II-B4). In the third iteration, we will introduce a technique for composing existing programs when no single program matches the specifications (Section II-B5). An implementation and evaluation will indicate if this technique is sufficient, or if an alternate composition approach is needed to stitch together solutions when none exist. The fourth iteration will be a wrapper around the first three, adapting this approach to new domain(s).

B. Approach

Figure 2 illustrates the big picture of our approach, which involves five main activities; we describe each in detail.

1) ***Defining Specifications****:* Instead of textual queries, this approach uses lightweight, incomplete specifications that characterize the desired behavior of the code (*User Input* in Figure 2). These specifications are in the form of input/output pair(s) (e.g., two unsorted list / a combined sorted list for the first scenario in Section I-B) and/or partial program fragments (e.g., a *sort* component). The size of the specifications defines, in part, the strength of the specifications, and this approach allows a developer to provide specifications incrementally.

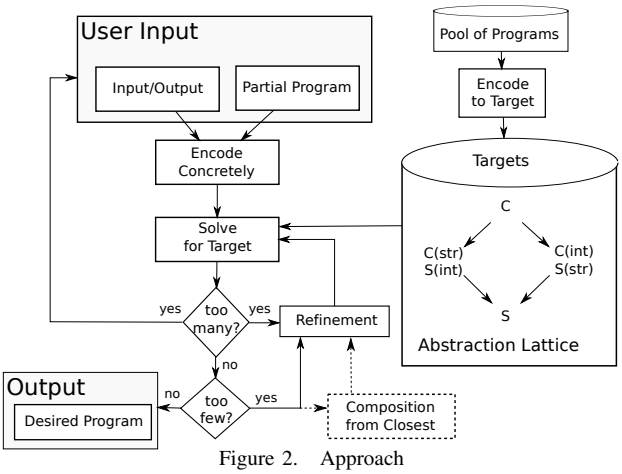


Figure 2. Approach

*2) **Encoding a Repository of Programs**:* Offline, a repository of programs (*Pool of Programs* in Figure 2) is encoded into constraints (*Encode to Target*). 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 efficiently) with the precision of matches (a level too coarse could return too many matches). To permit exact or *close enough* matches to be identified, the constraints are encoded at various levels of abstraction (Section II-B4).

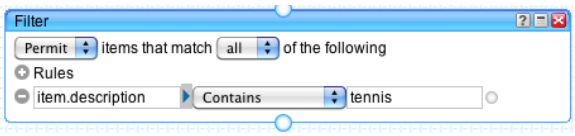
For Yahoo! Pipes, we encode programs at the component level, mapping each components onto constraints. Since it is a dataflow language, constraints are classified in terms of inclusion, exclusion, and order. *Inclusion* ensures completeness; all relevant records from the input exist in the output. *Exclusion* ensures precision; all records in the output are relevant. *Order* ensures that the records are ordered properly, as is necessary when asserting constraints over lists.

*3) **Identifying Matching Programs**:* After encoding the specifications (*Encode Concretely* in Figure 2), an SMT solver determines which, if any, programs match the specifications (*Solve for Target*). On the first iteration of this process we search for an exact match, which corresponds to a concrete encoding (*C* in the *Abstraction Lattice*).

*4) **Refinement**:* 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. To address these scenarios, we support refinement on the specifications and encodings (*Refinement* in Figure 2).

Tuning Lightweight Specifications: If there are too many matches, the developer can refine or extend the specifications. For example, if the input lists provided for the first scenario of Section I-B are already sorted, some programs without a sort component might match the specification.

If there are too few matches, the targets used by the solver might be too strict, so we might need an alternate matching criteria (Section II-B4) or a composition of programs (Section II-B5). Alternately, considering a subset of the input/output pairs may identify a *close enough* match.



(a) Filtering Component in Yahoo! Pipes

Concrete: $(contains(in, r) \wedge substr(field(r), \text{“}tennis'')) \rightarrow contains(out, r)$

Symbolic: $\exists s \mid (contains(in, r) \wedge substr(field(r), s)) \rightarrow contains(out, r)$

(b) Abstraction Specifications for Filtering Component

Figure 3. Abstraction Lattice Example

Changing Program Encodings: Stronger constraints utilize concrete values and identify exact matches (*C* in the *Abstraction Lattice*), while weaker constraints utilize symbolic values (*S* in the lattice). We currently define constraints over two datatypes that can hold concrete or symbolic values: integers and strings. In this lattice, we can relax either the integers (i.e., *C(str) S(int)*), the strings (i.e., *C(int) S(str)*), or both (i.e., *S*). To illustrate, consider a filter component in the Yahoo! Pipes language, shown in Figure 3(a), that accepts records with “tennis” in the description (recall the second scenario of Section I-B). The concrete and symbolic constraints are shown in Figure 3(b). In the symbolic constraint, the string is relaxed, requiring that there exists some string s such that all records in the output list contain s.

Another lattice we explore systematically relaxes the inclusion, exclusion, and order constraints. We recognize that using lattices to relax matching criteria is similar in spirit to the pre/postcondition lattices used in previous work [4] [5]. The key differences are that we do not rely on the availability of component specifications but rather infer them directly from the programs, and our use of symbolic matches. The lattices we define 1) exploit the fact that most languages contain constraints over multiple data types (e.g., strings, floats, integers, booleans, lists) and relax matching by treating variables as symbolic, and 2) leverage domain-specific language properties, such as order constraints, that apply to data-flow languages like Yahoo! Pipes or query languages like SQL that operate on tables of data.

*5) **Composing Programs**:* If no single program matches the specifications, there might exist a composition of multiple programs that does (*Composition from Closest* in Figure 2). Here, we begin with the closest program match (defined, for example, in terms of broken constraints), and determine if the output, provided as an input to another program, achieves the specified output. That is, we try to find a conjunction of two programs P and Q that match the specified input/output pair. For example, consider the third scenario of Section I-B. Program P might extract and format population information for each state, but leave the list unsorted. If program Q sorts data, the composition of P and Q would create a pipe PQ with the desired behavior.

C. Scope

We have identified and experimented with one domain of languages, Yahoo! Pipes, but our overall approach is applicable to other domains. Currently, we can encode constraints for:

- List manipulation (i.e., sorting, head and tail, insertion and deletion, size, copy, concatenate, reverse, distinct)
- String processing (i.e., equality, substring, less than comparisons, length, concatenation, reverse)
- Integer arithmetic (i.e., addition, subtraction, equality, less than comparisons)

The constraints represent a limited yet broad range of common programming tasks, which can apply to SQL queries, string processing in languages like Java, C# and C++, Lustre for control systems, or Unix commands that use the pipe operator.

Languages must meet two requirements to be amenable to our approach. First, there must exist a sufficiently large amount of code that solve problems within the domains supported by our encoding; this forms the repository. The second requirement is one to make our approach of practical value; specifying the input/output must be easier than building the program. Over more general domains, the presence of side-effects and the richness of program semantics make specification extraction more difficult, which is a challenge we anticipate as we move forward with this research.

D. Evaluation and Preliminary Work

We hope to address one big question: When is writing our lightweight, incomplete specifications more cost-effective than building a program from scratch, or than other standard search techniques? We have several factors that will be evaluated: difficulty of problem being solved, commonality of the problem, quality of solutions, and background knowledge of the participant.

For preliminary work, we have a repository of Pipes programs from previous work [13] [14], forming *Pool of Programs* in Figure 2. We have prototyped the infrastructure to support experimentation for the Yahoo! Pipes domain and are currently in the evaluation stage of the second iteration of our research methodology (Section II-A). Our current implementation encodes a subset of the Yahoo! Pipes language into constraints in SMT-LIB2 format, considering the problem domains described in Section II-C, and uses the Z3 SMT solver [15] to identify matches. We have automatically encoded over 3600 Pipes programs considering each level of the *Abstraction Lattice* (Section II-B4).

III. Prior Publications

Organized by conference: ICSE [14], FSE [16], ESEM [13] [17], SIGCSE [18], and VL/HCC [19].

Acknowledgment

Special thanks to my Ph.D. advisor Sebastian Elbaum for his continued guidance on this research. This work was supported by the NSF Graduate Research Fellowship under CFDA#47.076 and NSF Award #0915526

References

[1] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *Software, IEEE*, vol. 25, no. 5, pp. 45 –52, sept.-oct. 2008.

[2] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, June 1992.

[3] 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.*, vol. 21, no. 1, pp. 4:1–4:25, Dec. 2011.

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

[5] J. Penix and P. Alexander, "Efficient specification-based component retrieval," *Automated Soft. Eng.*, vol. 6, April 1999.

[6] S. P. Reiss, "Semantics-based code search," in *International Conference on Software Engineering*, 2009, pp. 243–253.

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

[8] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "Exemplar: Executable examples archive," in *Int'l Conf. on Software Engineering*, 2010, pp. 259–262.

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

[10] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *Conference on Programming language design and implementation*, 2011.

[11] "Yahoo! Pipes," http://pipes.yahoo.com/, February 2011.

[12] 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.

[13] K. T. Stolee, S. Elbaum, and A. Sarma, "End-user programmers and their communities: An artifact-based analysis," in *International Symposium on Empirical Software Engineering and Measurement* 2011, pp. 147–156.

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

[15] "Z3: Theorem Prover," http://research.microsoft.com/en-us/um/redmond/projects/z3/, November 2011.

[16] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee, "Using assertions to help end-user programmers create dependable web macros," in *Symposium on Foundations of Software Engineering*, 2008.

[17] K. T. Stolee and S. Elbaum, "Exploring the use of crowdsourcing to support empirical studies in software engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2010.

[18] K. T. Stolee and T. Fristoe, "Expressing computer science concepts through kodu game lab," in *Technical symposium on Computer science education (SIGCSE)*, 2011.

[19] K. T. Stolee, S. Elbaum, and G. Rothermel, "Revealing the copy and paste habits of end users," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2009.