8-2013

Solving the Search for Source Code

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

SOLVING THE SEARCH FOR SOURCE CODE

by

Kathryn T. Stolee

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Sebastian Elbaum

Lincoln, Nebraska

August, 2013

SOLVING THE SEARCH FOR SOURCE CODE

Kathryn T. Stolee, Ph.D.

University of Nebraska, 2013

Adviser: Sebastian Elbaum

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

We propose a novel approach to semantic search that addresses some of these limitations and is designed for queries that can be described using an example. In this approach, programmers write lightweight specifications as inputs and expected output examples for the behavior of desired code. Using these specifications, an SMT solver identifies source code from a repository that matches the specifications. The repository is composed of program snippets encoded as constraints that approximate the semantics of the code.

This research contributes the first work toward using SMT solvers to search for existing source code. In this dissertation, we motivate the study of code search and the utility of a more semantic approach to code search. We introduce and illustrate the generality of our approach using subsets of three languages, Java, Yahoo! Pipes, and SQL. Our approach is implemented in a tool, Satsy, for Yahoo! Pipes and Java. The evaluation covers various aspects of the approach, and the results indicate that this

approach is effective at finding relevant code. Even with a small repository, our search is competitive with state-of-the-practice syntactic searches when searching for Java code. Further, this approach is flexible and can be used on its own, or in conjunction with a syntactic search. Finally, we show that this approach is adaptable to finding approximate matches when exact matches do not exist, and that programmers are capable of composing input/output queries with reasonable speed and accuracy. These results are promising and lead to several open research questions that we are only beginning to explore.

DEDICATION

To my loving spouse.

ACKNOWLEDGMENTS

A tremendous amount of time and effort goes into the work behind and writing of a dissertation. I have been extremely fortunate to have a team of people behind me, providing support and guidance along the way. It would be impossible to list everyone and express exactly how much their support means to me. To my family, friends, and colleagues, thank you.

To my husband, Derrick, for his unfailing love and support.

To my advisor, Sebastian, for his support, guidance, and friendship.

To my committee, Matt, Myra, Gregg, and Kent, for their time, effort, and interest.

To my family, Mom, Dad, Colin, Dan, Ann, Lillie, Vivian, and Ethan.

To my best friends, Aimee, Stephanie, and Elena.

To my UNL colleagues, Brady, Zhihong, Josh, Eric, Javier, Tingting, and Pingyu.

To the UNL CSE faculty and staff.

To the Lincoln, NE community. Thank you for the past 9 years of hospitality.

GRANT INFORMATION

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consider a novice Java programmer who is trying to find a snippet of code that extracts an alias from an e-mail address (also known as a username and appears before the @ symbol). The programmer turns to Google, which our survey (Section 3.1) and others [59] show is the most common approach, and issues a search query with the following keywords: *extract alias from e-mail address in Java*. As expected, the query returns millions of results (over 46 million, in fact). None of the top ten results (P@10, a typical IR measure to assess the precision of search engine results [10]), even provide a method for decomposing an e-mail address into parts, which is the first step towards extracting the alias. Now, if the programmer is knowledgable enough about the domain to refine the query with the term *substring*, then the top ten results include two relevant solutions.

Despite the simplicity of the programming task, this illustrates a common situation for programmers. Following a search for code reuse or use as an example, programmers must sift through many irrelevant results, especially when the desired behavior cannot be tied to source code syntax or documentation. As repositories of source code

grow in size and diversity, and as programmers continue to turn to search during development [57], finding relevant code becomes increasingly important.

We have designed an approach to code search that addresses many of the shortcomings of existing search techniques, most notably by allowing programmers to describe what they want their code to do rather than how it is implemented, returning only relevant results in the search, and finding *close enough* solutions when no exact solutions exist. The general idea is that programmers provide concrete behavioral specifications as inputs and outputs and an SMT solver identifies available source code, encoded as constraints, that matches the specifications. This approach is designed to be effective for a subset of code search tasks, that is, those that can be expressed by an input and output example. The underlying assumption is that a non-trivial number of code search queries have this property, and we provide some evidence of this in Section 3.3.

With the previous example of searching for a program that extracts the alias from an e-mail address, the input could be the string "susie@mail.com" and the output the string "susie". While this form of query changes the search model from the common keyword query, it lets the programmer specify the desired behavior, without the need to know *how* to achieve a certain outcome, just *what* that outcome is.

This example-based specification model is inspired by two lines of work, *programming by example* (or, *programming by demonstration*), and *program synthesis*. Some programming by example approaches aim to generate programs for tasks that are demonstrated by example [12], such as providing a string before and after a transformation. Yet, the programs that can be generated through those approaches are limited and must follow well-defined templates or sequences. For example, in the TELS text editing by example system [76], programs are generated by recording and generalizing editing actions on strings in a text editor. For generalizations that involve

string constants, they use a rigid hierarchy based on common subsequences. Other properties of strings, such as length or case-insensitive equality, are not included in the hierarchy and thus would not be part of a generated program. In our work, we find *existing* code that performs a relevant transformation, which promotes reuse.

The other line of work, program synthesis, aims to generate programs that match a provided input/output example, and program generation is guided by a constraint solver. Similar to programming by demonstration, this approach also relies on predefined functions and templates to guide the solver in finding a solution. The solver will try every possible combination of functions and templates to achieve the desired behavior, which can be quite time consuming, even for small programs (e.g., in toy problems, insertion and deletion on graphs can take several minutes to resolve [60]). Our approach is not restricted to predefined functions and templates, allowing us to return code that may be too complex for a code synthesizer to generate efficiently, and also promotes reuse (we discuss more related work in Section 10.2).

Just like most other search engines, our search approach involves three phases: *specifying* a query, *indexing* entities (i.e., source code), and *matching* the query to relevant entities. We discuss each phase in the context of our approach to code search.

Specifying a query is performed by the programmer, the one who wants to find code for a particular task. In our approach, a programmer writes a concrete behavioral specification in the form of input and output, such as "susie@mail.com" and "susie" from the previous example.

Indexing source code is an automated process performed offline. This process involves no input from the programmer. Our indexing is unique in that it requires a transformation that involves symbolic analysis [7, 8, 36] to map a program's semantics onto constraints that summarize the program behavior. For example, the indexing process would map the Java snippet:

```
1 int upper = input.indexOf('@'});
2 String output = input.substring(0, upper);
```

onto the following constraints (roughly):

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

Constraints `c1` and `c2` represent line 1 of the source code. The first constraint, `c1`, defines `upper` as the location of '@' in `input` or -1, and `c2` asserts `upper` is the first index of '@' in `input`, per the semantics of the `indexOf` method in `java.lang.String`.[1] Constraint `c3` represent the second line of source code. It asserts that the `output` matches `input` within bounds of `0` and `upper`, per the semantics of the `substring` method in `java.lang.String`. This is the basic process by which our approach indexes programs: mapping program semantics to constraints by evaluating each program statement. The constraints are generated automatically, a process described in Section 5.2. The constraints are never shown to the programmer, but rather are consumed by the solver during the search process to identify viable matches.

Matching is now possible, given the input/output query provided by a user and an encoded repository of programs generated by symbolic analysis. The first step in this phase consists of transforming the provided input/output into additional constraints. For the previous example, that would be:

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

The second step consists of pairing the input/output constraints with each of the programs indexed in the repository, and using an SMT solver to identify which pairs are

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

satisfiable and hence constitute a match. For our email alias example, an SMT solver would return *sat* for the conjunction of the snippet encoded through constraints `(c1 ∧ c2 ∧ c3)` and the input/output encoded through constraints `(c4 ∧ c5)`, indicating that the code indeed matches the specification. Contrastingly, if the specified output was instead "mail.com" (the programmer meant to identify the e-mail domain instead of the alias), the SMT solver would return *unsat* indicating that the code does not match the specification.

This illustrates the essence and novelty of the approach, but it does not address some critical issues such as the broader applicability of the approach to other domains, handling richer specification models required by diverse domains, and refining the set of potential matches. In this work, we address these issues.

We formally define our approach to semantic search. For two domains, the Java String library and Yahoo! Pipes mashup programs, we instantiate and assess the approach. An additional domain is included for motivation and approach illustration, SQL select statements (Chapter 4). These domains were selected in part to illustrate the generality of the approach by utilizing three diverse forms of input/output specification and in part because of their relatively simple and common underlying semantics and the availability of repositories that could be searched for evaluation (Chapter 7, Chapter 8).

1.1 Contributions

The contributions of this work are:

- Characterization of how developers use search to find code based on a survey of 100 participants (Section 3.1)

- Definition of a novel approach to semantic code search that uses an SMT solver to identify matches given lightweight specifications and programs encoded as constraints using symbolic analysis (Section 5)
- Illustration of the applicability of this approach and a range of potential specification models covering three languages: Java, Yahoo! Pipes, and SQL (Section 4)
- Implementation of the approach in two domains: Java string library and Yahoo! Pipes (Section 6)
- Empirical evaluation of how our search approach compares to, and can improve on, the state-of-the-practice code searches in Java (Section 7)
- Assessment of the effectiveness and efficiency of the approach given various levels of abstraction and measuring precision and recall in Yahoo! Pipes (Section 8.1)
- Assessment of the cost and accuracy of formulating output from an input in Yahoo! Pipes and SQL through a controlled quasi-experiment (Section 8.2)

1.2 Outline of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides background information on the domains used in the evaluation and the program analysis techniques used in the transformation process from source code to constraints. To show the importance of research in code search and motivate the use of examples as a query model, we survey programmers and look for evidence of input/output examples in the wild in Chapter 3. To illustrate various aspects of our approach, from the benefits of using multiple specifications to the power of abstraction, Chapter 4 provides several examples in three targeted domains, Java, Yahoo! Pipes, and SQL. Our approach to

semantic search is formally defined in Chapter 5, and we describe the implementation details for Java and Yahoo! Pipes in Chapter 6. The evaluation and results are split into two chapters by domain, where we evaluate this approach in the Java domain in Chapter 7 and in Yahoo! Pipes in Chapter 8. Threats to validity are presented in Chapter 9, followed by related word in Chapter 10. A discussion of the implications of this work, future work, and conclusion are in Chapter 11.

Chapter 2

Background

Our approach to code search combines two areas of software engineering, program analysis and code search. In this section, we introduce the concepts from program analysis required to understand the approach and the domains used for illustration and evaluation throughout the rest of this work.

Our approach to code search has been applied in three domains, Java (Section 2.1), Yahoo! Pipes (Section 2.2), and SQL (Section 2.3). We provide brief backgrounds on each of these domains, focusing on how programs are composed in the state-of-the-practice, as code search is intended to be most useful during program composition.

The indexing phase of the proposed search approach relies on *symbolic execution*, a program analysis technique that uses symbols, rather than concrete values, to execute programs (Section 2.5). The solving phase relies on constraint solvers, which we describe in Section 2.4.

2.1 Java

Java [28] is an object-oriented programming language used for a variety of purposes and primarily by experienced programmers. Millions of lines of Java code from open source projects are publicly available to analyze [14], and Java is ranked as the second-most-popular programming language today based on the number of skilled engineers world-wide, courses offered, and third-party vendors [73].

In a survey we conducted of 109 programmers about their search behavior (Section 3.1), Java was the most popular language among the participants with 63 (58%) identifying Java as one of their most commonly used programming languages. Figure 2.1 provides a simple Java method that takes a string as input and returns that same string minus the last character (i.e., `trimmed`).

Listing 2.1: Java Example Method

```
1 static String trimEnd(String input) {
2     String trimmed;
3     if (input.length() > 1) {
4         trimmed = input.substring(0, input.length() - 1);
5     } else {
6         trimmed = input;
7     }
8     return trimmed;
9 }
```

Java programmers use a variety of tools to support their development practices (e.g., IDEs, Javadoc documentation, tutorials, automated analysis tools, advanced APIs). Programs are composed by typing (or copy-and-pasting) into a text editor, which makes it easy to leverage resources written by others.

2.2 Yahoo! Pipes

Yahoo! Pipes is a visual mashup language that boasts over 90,000 users [32], and a public repository of over 100,000 artifacts [49]. Motivated by the popularity of the Yahoo! Pipes mashup environment and the vast number of artifacts available in a public repository, we chose to target Yahoo! Pipes as part of this work.

A mashup is an application that manipulates and composes existing data sources or functionality to create a new piece of data or service. One common type of mashup, for example, consists of grabbing data from some data sources (e.g., house sales, vote records, bike trails, map data), joining those data sets, filtering them according to a criterion, and plotting them on a map published at a site [77]. This type of behavior is naturally expressed in Yahoo! Pipes.

To write a program in Yahoo! Pipes, programmers use the Pipes Editor, a visual development environment accessible through the browser. Pipes are composed by dragging and dropping predefined modules and connecting them with wires to define the data and control flow. Figure 2.1 shows a sample pipe mashup that joins and filters the content of two RSS feeds based on the word "tennis".

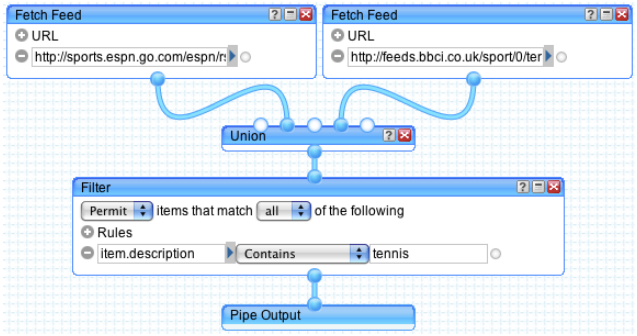


Figure 2.1: Yahoo! Pipes Example Method

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

Figure 2.2: Example Record/Item from an RSS Feed (author anonymized)

The structure of a pipe resembles a graph, where the nodes are referred to as modules (boxes in the figure), and the edges are referred to as wires (connections between the modules). Visually, mashup modules appear as the boxes in Figure 2.1 and wires appear as the lines between the modules. The behavior of the pipe can be best understood from top to bottom, as the data flows in a directional manner from the top of the pipe through the output at the bottom. In the example, there are two URLs, one in each *Fetch Feed* module, which act as inputs to the pipe. The output from those modules is concatenated by the *Union* module, and then a filter operation is performed to retain items with the word "tennis" in the description (the *Filter* module). Every pipe has exactly one output, shown at the bottom of the pipe and labeled *Pipe Output*. The output of a module is a combined and modified list of items from the RSS feeds provided as input.

The URLs access RSS feeds, and a pipe can have one or more URLs as input. An RSS feed produces a list of items. Each item is a map data structure with key-value pairs. For illustration an example item, also called a record, gathered from an RSS feed is shown in Figure 2.2. The keys are `Title`, `Description`, `Link`, `Author`, and `Date`. The first four keys map to values of type string, and the last key maps to an integer (i.e., the date is converted to an integer).

Yahoo! Pipes programs are executed on Yahoo's servers and the output is provided in the browser. The output of a Pipes program can be integrated into a webpage or a RSS feed aggregator.

2.3 SQL

SQL [65] statements are programs that support database management operations (e.g., insertion, deletion, retrieval). It is ranked 17th in popularity among programming languages according to a recent study based on the number of skilled engineers world-wide, courses offered, and third-party vendors [73].

SQL select statements have been used for decades to support data retrieval, operating on their own or being embedded into other programming languages. For example, consider the following database table containing contact information for club members and their join date:

```
 Name              | Date       | email
+------------------+------------+--------------------------+
 Alice Bower       | 2008-08-19 | abower@unl.edu
 Carl Dent         | 2008-10-10 | cdent@smsd.org
 Eleanor Fae       | 2008-08-19 | efae@cse.unl.edu
 Greg Happ         | 2008-08-19 | ghapp@unl.edu
```

The following SQL select statement will extract only those participants who have e-mail domains `@unl.edu`:

```
SELECT * FROM members WHERE email LIKE '%@unl.edu';
```

yielding the following result:

```
 Name              | Date       | email
 +-----------------+------------+--------------------------+
 Alice Bower       | 2008-08-19 | abower@unl.edu
 Greg Happ         | 2008-08-19 | ghapp@unl.edu
```

2.4 Constraint Solvers

In our search approach, matching is performed by a Satisfiability Modulo Theory (SMT) solver, which is a specific type of constraint solver. The purpose of an SMT solver is to determine if a logical formula is satisfiable. That is, the solver looks for an assignment of values to variables such that a formula is true.

To illustrate, consider the following example with four predicates including linear arithmetic and inequalities. Let the variables be integers, $a, b, c \in \mathbb{Z}$, subject to the following constraints:

$(a \geq 0) \wedge (b = 2) \wedge (c = 3) \wedge (a + b = c)$

An SMT solver would determine that this formula is satisfiable when $a \mapsto 1$ (this reads, "a maps to 1"). The output of a solver has three potential values, *sat*, *unsat*, and *unknown*. In this case, *sat* is returned since the formula is satisfiable. One comment about this formula is that it is rather tightly constrained in that there is only one value for a such that the formula is satisfiable. However, a more flexible formula can be created and given to the solver, such as the following (the second predicate is removed):

$(a \geq 0) \wedge (c = 3) \wedge (a + b = c)$

Here, the domain of a is constrained to be positive ($a \in \mathbb{Z} \wedge a \geq 0$), but the domain of b is unconstrained within the integers ($b \in \mathbb{Z}$). There are many possible solutions, for example, $a \mapsto 1 \wedge b \mapsto 2$, and $a \mapsto 2 \wedge b \mapsto 1$. Thus, the solver would return *sat*. However, the solver needs to find only one solution to return *sat*.

In the event that no solution exists, such as the following formula, the solver returns *unsat*. This means there is a conflict among the constraints and there does not exist an assignment of values to the variables such that the formula is satisfiable:

$(a = 0) \wedge (b = 2) \wedge (c = 3) \wedge (a + b = c)$

The change from the first formula is in the first predicate, where $a = 0$, and now there is a conflict since $0 + 2 \neq 3$. Thus, *unsat* is returned.

A third output of a solver is *unknown*, which can occur when a constraint system is too complex for the solver, or when finding a solution takes too long and the solver is stopped preemptively.

SMT solvers are similar to SAT solvers, but instead of boolean variables, SMT formulas can contain predicates from other theories, such as linear arithmetic just shown, and additionally uninterpreted functions. Many off-the-shelf solvers are available for use (e.g., Z3 [79], Yices [78], CVC3 [11]), and most accept a common format for the constraint definition, SMTLIB2 format [61]. An example of this format is shown in Listing 6.1.

In this work, we translate source code semantics and concrete input/output examples into predicates that can be consumed and checked by an SMT solver. That is, given an input/output example and an encoded program, the solver will return *sat* if the program meets the specification.

2.5 Symbolic Execution

Symbolic execution is the process of executing a program with *symbolic*, rather than *concrete* values. It traverses the paths of a program while accumulating path conditions to describe the potential behavior of a program. This technique was first proposed in the 1970's [7, 8, 36], but has come to see greater popularity and application with recent advances in computing technology [6]. In our approach, we use symbolic execution to collect path constraints to assist with the encoding process (Section 5.2).

To give an idea of how symbolic execution works, consider the program in Listing 2.1. There is exactly one predicate, leading to two path conditions. The predicate comes

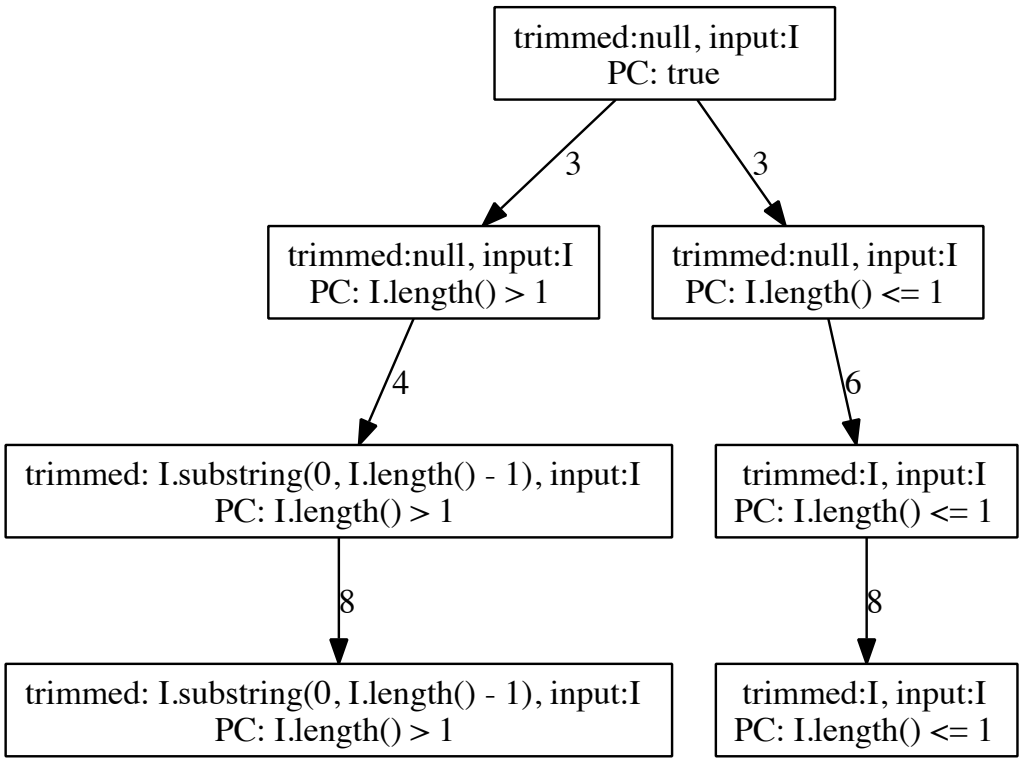


Figure 2.3: Symbolic Execution of Program in Listing 2.1

from the $if - statement$ on line 3. There are two string variables in the program, *input* and *trimmed*. One, *input*, has a symbolic value after line 2 since its value is never defined concretely. We show the symbolic execution tree in Figure 2.3 where the transitions are labeled with program control points.

At the start of the program (lines 1-2), variable *input* is symbolic in the domain of Strings and *trimmed* is null. This is shown at the top of the tree where $trimmed : null, input : I$ appear, indicating these *input* holds a free, or symbolic, value. For program arguments (i.e., *input*), the developer can choose which are symbolic or concrete; in this analysis, it is left symbolic. The path condition (PC) is *true* since all executions of the program will get to line 2. Line 3 holds a predicate, on which the tree branches. On the left side is the true branch when $I.length() > 1$ (I, holding

the symbolic value for *input*), and on the right is the false branch where $I.length \leq 1$. The path conditions are updated as a conjunction of *true* and the new branching condition. This branching stage also checks for path infeasibility with a decision procedure so that not all paths need to be explored and unreachable code can be ignored. A constraint solver, like the ones introduced in Section 2.4, is often used for this stage.

Lines 4 and 6 in the program restrict the value of *trimmed*. This is reflected after the 4 and 6 transitions in the tree, where $trimmed : I.substring(0,\ I.length() - 1)$ after line 4 or $trimmed : I$ after line 6. Both branches of the tree end at the return statement on line 8; nothing changes in the path condition or the variable domains. The path condition at the end holds only symbolic values as the value of *input* is free throughout the analysis.

In the encoding phase of our approach to semantic search, these paths and domain restrictions on the variable values are leveraged when translating program source code into constraints for a solver. For example, given the execution tree, Listing 2.2 shows the variables used, path condition, and executed statements corresponding to the left branch of the tree in Figure 2.3, and Listing 2.3 corresponds to the right branch.

Listing 2.2: Path #1 for Listing 2.1

```
1 String input;
2 String trimmed;
3 pc1: input.length > 1
4 trimmed = input.substring(0, input.length() - 1);
5 return trimmed;
```

Listing 2.3: Path #2 for Listing 2.1

```
1 String input;
2 String trimmed;
3 pc2: input.length <= 1
4 trimmed = input;
5 return trimmed;
```

Since *input* is never defined in Listing 2.2 and Listing 2.3, this preserves the fact that it is symbolic from the perspective of the solver. Thus, *input* and I are synonymous in the path encoding. Each of these paths would be encoded as conjunctions of constraints, and the disjunction of the encoded paths represents the behavior of the original program. In our approach, we are able to use path conditions computed through symbolic execution to define the relationships among variables in a program and the potential values variables can hold under different executions of the program.

Chapter 3

Motivation

Searching for source code is a common task among programmers, but the tools, approaches, and uses for found code vary widely and are often tightly coupled with the programming language and developer context. In this section, we motivate the need for further research in code search and provide evidence for the utility of an input/output query model.

To motivate research in code search, we present the results of a survey that aimed to gain a better understanding of how and why programmers search for source code (Section 3.1). Next, we illustrate how code search manifests itself in the state-of-the-practice for our targeted domains, Java, Yahoo! Pipes, and SQL (Section 3.2). Last, we motivate the use of an input/output query model for code search. In our proposed approach, programmers write queries as examples, specifying the input and output of their desired code (Section 3.3). This addresses some of the shortcomings of state-of-the-practice textual query model. To motivate this shift, we explore questions asked in online help forums, observing the frequency of input/output examples within the questions.

3.1 User Survey

Previous work has studied the question of how and why programmers search for source code [59], with a survey that focused on graduate student behavior. To confirm the findings and understand more about the the tools used in code search, we conducted a survey with similar goals. Additionally, we targeted different populations, undergraduate students and people with programming knowledge on Amazon's Mechanical Turk marketplace [44].

To characterize how and why programmers search for source code, we designed a survey with the goal of addressing the following research question *RQ1*, which we broke down further into three sub-questions:

RQ1: How and why do programmers search for source code?

RQ1(a): How frequently do programmers search for code?

RQ1(b): Why do programmers search for source code?

RQ1(c): Which tools to programmers use to search for code?

The survey contained ten questions related to the search habits of the participant (details are found in Table B.1 in Appendix B). The first five questions aim to profile the survey participants. Survey question 6 addresses *RQ1(a)*, questions 7, 8, and 9 pertain to *RQ1(c)*, and question 10 addresses *RQ1(b)*.

3.1.1 Participants

We targeted two populations with this survey, students in two undergraduate classes at the University of Nebraska-Lincoln and workers on Mechanical Turk. It was administered prior to a quasi-experiment discussed in Section 8.2.

Mechanical Turk [44] is a service hosted by Amazon that allows people to reach and compensate others to complete tasks that require human input, such as tagging images or answering survey questions. It hosts the tasks, manages payment, and makes the tasks accessible to a large and existing workforce. There are two roles in Mechanical Turk, a requester and a worker. The requester is the creator of a human intelligence task, or HIT, which is intended to be a small, goal-oriented task. The worker is the one who completes the HIT. A HIT may or may not have prerequisites for the user, which are referred to as qualifications. Completed tasks are returned to the requester for evaluation. If a requester is dissatisfied with submitted work, they hold the right not to pay the worker.

With the Mechanical Turk population, we used a qualification test to help control the quality of responses in an unknown population (see Appendix B). This contained programming questions that required correct responses in order for respondents to participate in the study. These questions, shown in Appendix B.3, ensured that participants had sufficient knowledge of the Yahoo! Pipes and SQL programming languages.

In total, we received valid responses from 109 participants.[1] Of those, 43 (40%) came from junior/senior undergraduate classes at UNL while the remaining 60% came from Mechanical Turk. The breakdown across participant group and gender is shown in Table 3.1.

In a question about programming experience, eight (7%) participants self-reported to have no programming experience,[2] 19 (17%) had less than two years of experience,

[1]Three participants were removed from the pool on account of inconsistent responses. These participants claimed to program weekly but search for source code daily, and this seemed suspicious.

[2]The participants with no programming experience were retained since the subsequent quasi-experiment performed by the participants dealt with end-user programming languages. Even so, these participants passed the qualification test questions.

Table 3.1: Summary of Participants (Question 2)

	Male	**Female**	**Total**
Embedded Systems class at UNL	25	2	27
Software Engineering class at UNL	14	2	16
Mechanical Turk	44	22	66
Total	83	26	109

Table 3.2: Programming and Search Frequency (Question 4, 6)

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

53 (49%) had between two and five years of experience, and 29 (27%) had more than five years of programming experience.

The most common languages used by participants were object-oriented languages like Java and C/C++. Database or spreadsheet languages are also popular, with SQL and Excel being the 4th and 5th most common languages listed (a complete list of languages can be found in Appendix B.2).

3.1.2 Search Frequency

To address *RQ1(a)*, we asked how frequently programmers write source code and how frequently programmers search for code. Table 3.2 summarizes our findings. Among all participants, 43 (39%) reported they program daily and 49 (45%) program on a weekly basis. Among the participants who program daily, over half (25 of 43) also search for code daily. As a summary, *among those participants who program daily or weekly, 85% search for code at least weekly.* This finding is consistent with a recent (and independent) survey that looked at the search habits of 36 graduate students [59].

Table 3.3: Uses of Matched Code (Question 10)

Code Use	**Count**	**Percent**
Copy/paste as is	14	13%
Copy/paste and modify	52	48%
Get ideas for implementation	73	67%
Link to found code	8	7%
Other	6	6%

It was reported that 50% of the participants search for code "frequently" while 39% did it "occasionally."

3.1.3 Why do Programmers Search for Source Code?

To understand a little more about the motivation for code search and address *RQ1(b)*, we asked participants what they did with the source code they were looking for. Once useful code is found, Table 3.3 summarizes what the participants did with it (using a multi-select question). *Half of the participants reported that they would copy/paste and modify found code, which is akin to ad hoc reuse.* Two-thirds would use it to get ideas for implementation, and 13% would copy/paste as is. This is consistent with the previous findings that reuse and implementation examples are the most common purposes for a code search [59]. For the 6% who selected *other*, the participants indicated that the use depends on the time and resources they have available.

3.1.4 Tools Used in Code Search Activities

To address *RQ1(c)*, we asked participants about the tools they use for code search and the types of information they use for their search queries. In a free response question about where the participants search for code, nearly two-thirds (66%) mentioned using *the web*, *the internet*, or specifically *Google*. Table 3.4 presents the full results. One-

Table 3.4: Where do Programmers Search for Sample Code? (Question 7)

Location	**Count**	**Percent**
Google	47	43.1%
the web	24	22.0%
Stackoverflow	22	20.2%
Google Code Search	19	17.4%
Company code base	5	4.6%
MSDN	4	3.7%
API documentation	3	2.8%
www3schools	3	2.8%
14 other sources	1	0.9%

fifth (20%) mentioned searching `stackoverflow.com` specifically, a free question and answer site for programming and technology-related questions. Only 17% mentioned using a code-specific search engine like *Google Code Search*. Despite the availability of code-specific search engines, *information search engines are the most common tools used for code search*, echoing the findings in a previous survey [59].

The most popular methods used by participants for code search (in a multi-select question) were keyword (72%), function name (45%), and libraries used (35%). The 25% who selected *other* generally indicated that they search using a description of their goal (e.g., *reading file input*), and the language they wish to use (See Table B.3). One participant wrote that they search for code *"as a question worded how I think other people would have asked it on stackoverflow or forums"*. Another said they use, *"the specifics of a feature's functionality I'm trying to mimic"*, and another searches based on the *"end result"*. These comments indicate that the programmers are trying to use the textual query to describe the behavior of the code they need.

Finding relevant source code however, is not always easy with the current tools. The *participants reported that they must explore an average of 3.4 snippets of code before something useful is found.* A previous study found that approximately 3 out of

the first 10 matches were useful, which aligns with our finding [59]. It is important to mention that neither survey accounted for the process of *query reformulation*, which is the process of re-stating a query after viewing search results that are irrelevant, and is quite common in searching [27] and adds to the overhead. Evaluating the cost of a syntactic search for finding source code to reuse is an open question left for future work (Section 11.1.1).

3.1.5 Summary

From this survey, we have observed that code search is common and the overhead, which comes from examining and determining whether or not a match is useful, is non-trivial. These results are inline with recent findings from a study investigating the effectiveness of different code search approaches [59].

Still unclear is what it means for a participant to *explore* a search result, the cost of that exploration, and whether the self-reported search behaviors reflect actual search behavior. Gaining a more thorough understanding of the search and exploration processes and durations is critical to understanding the real cost of a code search, and this is left for future work. Regardless, the current overhead does suggest the need for better search tools.

Also unclear is *when* programmers search for code in their development processes, and what triggers the code search. Understanding this question is necessary to integrating a code search tool intuitively into the development process. As this research is still in the formative stages, this, too, is left for future work.

3.2 State-of-the-Practice Code Search

Searching for source code in different languages presents different challenges. Some languages have plentiful and well-documented online resources while others are proprietary or have other features that make searching via textual queries difficult (e.g., visual language, no public repository). In this section, we focus on three languages that expose different contexts, Java, Yahoo! Pipes, and SQL. We introduce each language and explore the current search capabilities in the state-of-the-practice.

3.2.1 Java

Given the popularity of the Java programming language, it is not surprising that Java resources abound on the web and can be found easily using textual search. The Java programmers we surveyed also frequently search for source code online. Of the 63 Java programmers, 56 (89%) program at least weekly, and 49 (78%) search for code at least weekly, which follows from the results of RQ1(a). On average, however, Java programmers look at 3.5 snippets before finding something useful, which is slightly higher than the average.

To illustrate the challenges programmers face with current search support, we draw on the results of three studies. In the first (Section 7.1), we performed 17 searches for source code on the web using titles from `stackoverflow.com` as textual queries. Stackoverflow is a popular question and answer website where programmers can ask programming and technology-related questions and members of the community can suggest answers. The most relevant answer can be marked as 'accepted'. Over 2.5 million questions have been asked on this forum.

Selection criteria is detailed in Section 7.1.1, but these questions were all tagged with `[java]`, `[string]`, and `[substring]` and contained input/output examples as part

of the question text. By evaluating the behavior of the code snippets on the first 10 results (Section 7.1), we found that on average, 1.5 of the snippets were relevant in that they behaved according to the input/output examples provided in the question text. However, we must point out that this performance is nearly twice as bad as the performance of syntactic search engines reported in our survey, where something relevant was found after exploring 3.4 snippets, indicating that approximately 3 snippets among the top 10 ought to be relevant (Section 3.1). A follow-up study that used humans to judge the relevance of snippets, rather than an analysis of the behavior, revealed that approximately 6.75 of the top 10 results are relevant (Section 7.2). This is twice as good as the self-reported surveys, but it may be artificially high due to our experimental procedures (Section 7.2.2.4). Based on our findings, it is unclear the exact cost of evaluating queries in Java, but if we take the surveys as an average, programmers must sift through a fair amount of irrelevant code (3-4 snippets) before finding what they want.

3.2.2 Yahoo! Pipes

Often, existing search capabilities of domain-specific languages are more limited than those for more mainstream languages. This may be due to a lack of a central repository, non-searchable code representations (e.g., for visual languages), or a lack of existing resources to search over (e.g., domain-specific languages). For Yahoo! Pipes, programmers can search only within the Pipes environment using URLs accessed, tags, keywords, or program components as queries. Using common search engines like Google is not viable as the pipes reside in a repository that is public, but has a proprietary format. Even if the authors make the pipe available outside of the

Table 3.5: State-of-the-Practice Search by URL Query on Global Repository

Pipe	URLs	Matches	P@10
1	`rss.weather.com`	71	2
2	`feeds.feedburner.com`	16,990	0

3	`anunturi-gratis.ro`
	`anunturi-utile.ro`
	`feedproxy.google.com`
		1,281	0

4	`ocregister.com`	38	0
5	`feeds.gawker.com` `lifehacker.com.au`	4	1

repository, a syntactic query can match only pipe metadata, such as the title and description created by the author.

To illustrate the challenges programmers face with current search support, we performed searches for five mashups that are intended to be typical pipes in the repository based on popularity and structural uniqueness (a full description of the rationale for the choice of these mashups is justified in Section 8.1.1), querying for the URLs used in each mashup. These searches were executed on the large, Yahoo! Pipes public repository, and the results are shown in Table 3.5. The number of matches can be in the thousands which is not surprising as many mashups include common URL domains.

For each search, the number of matches and *P@10* are reported.[3] For two of the searches, *Pipe 2* and *Pipe 3*, thousands of matches were returned in the search. The search with *Pipe 5*, on the other hand, returned only four results and one was relevant; two pipes were relevant among the top 10 for *Pipe 1*. The average number of relevant matches among the top ten results (P@10), determined by executing each pipe, is 0.06 (the metric, *P*@10, is a percentage of relevant items among the top 10). What this

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

illustrates is that for the more common URLs, programmers must sift through a lot of irrelevant results, and a pipe that behaves as they want might not be easy to find.

Using other built-in search capabilities does not fare much better. Searching by components retrieves even noisier results and requires the programmer to know how the pipe could have been implemented. The effectiveness of searching with tags is dependent on the community's ability and decision to systematically categorize their artifacts. Furthermore, given the redundancy among artifacts in the pipes repository, it has been hypothesized that programmers simply cannot find what they are looking for and are forced to re-invent the wheel [71].

3.2.3 SQL

In the survey presented in Section 3.1, 20 (18%) of the participants identified SQL as one of their most commonly used programming languages. All of the programmers perform programming tasks at least weekly and 70% search for code at least weekly. In a multi-select question, it was revealed that the most common tool used for search is Google (65%), and most programmers search using keywords (75%) and function names (55%). After performing a search, an average of 3.75 snippets of code must be explored before finding something useful.

Programmers can find many resources related to SQL using information search engines. However, given the simplicity of the SQL syntax and its popularity, even well conceived syntactic queries will return many irrelevant results. For example, searching Google for the term, "select rows from table based on substring in SQL" produces over 500k responses, and among the top 10, only two results provide queries that contain the `like` operator, which is a critical piece of the select statement. The process of evaluating code snippets for relevance often requires trial and error, and is costly to

the programmer. Given the frequency of keyword code searches that are performed by SQL programmers, it seems reasonable to target this domain with an alternative search approach.

3.3 Input/Output Examples in the Wild

In this section, we motivate the use of the input/output query model by investigating the extent to which programmers already employ that query model in online help forums. We observe that Java, Yahoo! Pipes, and SQL programmers, often turn to peer communities when they are looking for help. One popular community is `stackoverflow.com`, so this is the forum we use to identify input/output examples in the wild. [4]

3.3.1 Sampling

We collected questions related to Yahoo! Pipes using the tag `[yahoo-pipes]`, questions related to SQL using the tags `[mysql]` and `[select]`, and questions related to Java using the `[java]` and `[string]` tags. The second tag in SQL was used to restrict the questions to those dealing with select statements, as that is the scope of our SQL implementation [70]. The second tag in Java is meant to restrict the question to those dealing with strings, as this has been the primary focus of our implementation (Section 6.3). In Yahoo! Pipes, 248 questions were returned on March 27, 2013, in SQL, over 1,500 questions were returned on February 19, 2012, and in Java 7,420 questions were returned on June 17, 2013. In each domain, we sorted the results according to popularity (i.e., votes) and retained the top 100.

[4] In Section 8.2, we present the results of a quasi-experiment illustrating that programmers can accurately specify desired code using this model.

Table 3.6: Question type categories in StackOverflow. The numbers in parentheses represent the questions with textual or input/output examples.

Question Type	**SQL**	**Y Pipes**	**Java**
How do I do X with {SQL, YP, Java}?	74 (53)	58 (40)	43 (29)
Can I do X with/without Y?	8 (4)	18 (14)	9 (4)
What's wrong with ..., or Why does ... work?	7 (2)	9 (8)	15(14)
How does Y work?	6 (1)	4 (1)	19 (8)
What is an alternative to {SQL, YP, Java} for X?	0 (0)	10 (0)	0 (0)
Best way to do X?	0 (0)	0 (0)	5 (4)
Y versus Z?	4 (1)	0 (0)	9 (4)
unrelated	1 (0)	1 (0)	0 (0)

3.3.2 Analysis

An initial analysis was performed in the SQL domain to observe common question themes. This involved two passes over the questions. The first pass was for content analysis to collect common question themes and the second pass categorized the questions. This same process was repeated for the Yahoo! Pipes domain, and then on Java. When a question fit two or more categories, we selected the category that most closely fit based on the accepted or highest-voted answer from the community. Eight categories emerged from this analysis.

The next step was to check if the questions also had examples to illustrate the context of the question, and if those examples were in the form of an input and output. Examples contain parts of the context for the question asker as a means to illustrate their question.

3.3.3 Results

In each domain, there were a handful of dominant question types, shown in Table 3.6. In a question containing X, it represents a specific task, such as *remove a field from an RSS item* in Yahoo! Pipes, *combine two tables* in SQL, or *capitalize the first letter*

in a string in Java. *Y* refers to a language construct, such as the `Regex` module in Yahoo! Pipes, the `Inner Join` construct in SQL, or the `hashCode()` function in Java. For each question, we provide the frequency of occurrence in each language, as well as the number of those questions with which a descriptive or input/output example was provided (in parentheses). As an example, *How does Y work?* questions describe six questions from SQL, four questions from Yahoo! Pipes, and 19 questions from Java; one question in each of Yahoo! Pipes and SQL provided an example to more clearly illustrate the question being asked. In Java, examples were more common with eight of those questions providing examples.

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

"I've been looking for a simple java algorithm to generate a pseudo-random alpha-numeric string. ... Ideally I would be able to specify a length depending on my uniqueness needs. For example, a generated string of length 12 might look something like "AEYGF7K0DM1X"."[5]

To provide another example, consider the following question from SQL: *"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*

```
fruit  | color  | weight
-----------------------
apple  | red    | 34
kiwi   | red    | 23
banana | yellow | 100
kiwi   | black  | 3567
apple  | yellow | 23
```

[5]http://stackoverflow.com/questions/41107/how-to-generate-a-random-alpha-numeric-string

```
banana | green  | 2345
pear   | green  | 1900
```

"And now I want output - something like distinct(apple) and distinct(color) together and order by weight desc:

```
kiwi   | black  | 3567
banana | green  | 2345
apple  | red    | 34
```

"– pear | green // is not ok, becouse green is already used

– banana | yellow // is not ok, becouse banana is already used

So I need not only group by fruit, but also color (all unique). Any advice or solution?" [6]

Note that 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 knows only the behavior that is desired.

We also observed that these questions usually come with examples that help developers better specify the required behavior. Of the 74 "How do I do X" questions in SQL, 53 contained examples. Further, 36 examples were actual snippets of tables that serve as inputs and records that they expect as outputs. Of the 58 questions in Yahoo! Pipes of this type, 40 contained examples and 8 of those examples had example inputs and outputs for the desired behavior. In Java, 29 of the 43 questions contained examples, and 13 of those were inputs and outputs for their desired code. In some questions, the outputs were difficult to specify. One Java question asks how to convert a string to a Date() object. [7] While the input is easy to specify, the output is not, and thus this was not counted. To work around that difficulty, some questions

[6] `http://stackoverflow.com/questions/7639830`

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

used test cases to illustrate input/output examples in a more standard format. [8] All of this provides evidence that programmers already think in terms of examples when trying to accomplish tasks using the SQL and Java, and to a lesser extent, the Yahoo! Pipes language.

Another interesting observation is that "How does Y work" questions are not asked often by programmers to their community, particularly for SQL and Yahoo! Pipes. This may indicate that this type of question is well handled by existing resources like existing tutorials or other syntactic search engine findings.

3.4 Summary

In this chapter, we have presented the results of a survey that asked 109 participants about their programming experience and search habits. It was clear that programmers frequently search for code, and that state-of-the-practice code search techniques have some limitations that may be addressed with a more semantic approach. Then, we introduced the three languages used for illustration and evaluation throughout the rest of this work, showing some specific examples of when state-of-the-practice code search falls short. Since the state-of-the-practice code search requires the use of a textual query and our approach uses an input/output query model, we also explored the frequency of input/output specifications in the wild, observing that examples are commonly used in questions asked on `stackoverflow.com`.

The next chapter provide some illustrative examples of our search approach in the three targeted language, Java, Yahoo! Pipes, and SQL.

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

Chapter 4

Illustrative Examples

Based on the evidence presented in Chapter 3, we see an opportunity to support programmers searching for source code by using an example-based query model that might be more natural to how they currently conceptualize their needs during search. An additional opportunity comes from some of the limitations of existing search approaches, particularly for domain-specific languages, where source code examples and documentation can be hard to find. Our approach to code search takes advantage of both of these opportunities, first by using input/output examples as queries, and second by generating a repository to search over from existing code.

4.1 Semantic Search Approach Overview

Figure 4.1 provides a big picture of our approach. The approach relies on an input/output query model, which may be useful when a programmer can provide an example of what they want code to do, but perhaps finds the task difficult to describe (*I/O Specification*). A *source code repository* contains the potential *results* for the search, and is built by analyzing source code, rather than documentation or other resources

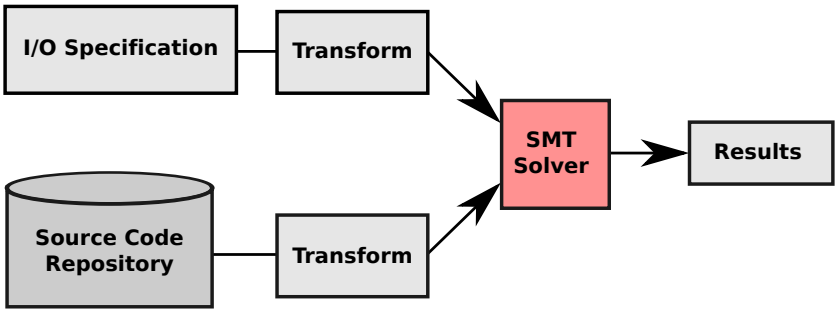


Figure 4.1: High-Level View of Semantic Search with Input/Output

that may be scarce for domain-specific languages. Behind the scenes, the *source code repository* is *transform*ed into constraints, a process called encoding. When a programmer issues a query, the *I/O specification* is *transform*ed into constraints and sent to the *SMT Solver*, along with the encoded repository. The *SMT Solver* identifies source code from the repository that meets the specification; this relevant source code is returned to the programmer as *results*.

We started exploring this approach to semantic search in the context of the end-user programming language, Yahoo! Pipes. Programs written in this language perform operations on lists of items. To generalize the approach to a more common language with similar semantics, we targeted SQL select statements. These programs, or queries, perform filtering operations over tables of data, which added a dimension of complexity to the first implementation. Supporting the Yahoo! Pipes language fragment also requires operations on strings, specifically identifying equality and substring relationships. To build on that support and explore our search approach in a broader context, we have also targeted Java methods and snippets that contain calls to the `java.lang.String` library.

In this chapter, we describe how our proposed approach could be instantiated in the domains targeted by our implementation for Java, Yahoo! Pipes and SQL, and when it would be valuable to do so.

4.2 Java Source Code

We begin illustrating the approach with Java, as this is the most popular of the supported languages and thus has the greatest opportunity for impact. Our approach requires a query consisting of pairs of example inputs and expected output. In the context of the Java String library, those inputs and outputs could be one of several datatypes; integers, characters, strings, and booleans are supported by our current implementation (Chapter 6). A repository of programs is also required, which has been encoded as constraints and from which source code matches are found during search. One of the main challenges of this approach, and one that involves a lot of complexity, is mapping source code onto constraints. In this section, we give a flavor of that complexity with the example in Section 4.2.3, but more details are provided in the approach definition (Section 5.2) and with the implementation details (Section 6.3).

Next, we provide five examples in the Java domain to illustrate key aspects of our search. First, we show how refinement on the specification can impact search results, second, we show how to map input/output specifications onto snippets of code, third, we show how to handle complex programs with multiple paths, fourth, we illustrate how the solver can find values for symbolic variables, and fifth we show what happens when no good matches can be found.

4.2.1 Refining the Search with Multiple I/O Pairs

In Section 1, we presented a specification that could be used as a query to find code that extracts an alias from an email address. The input, "susie@mail.com" and the output, "susie", form the specification. When using only one input/output pair, the specifications may be too weak, so many results may be irrelevant. For the alias extraction example, consider the following two results, `r1` and `r2`, each consisting of a single line: The first result, `r1`, is found by mapping the output to `scheme` and the input to `uri`.

```
r1: String scheme = uri.substring(0, 5);
```

The second result is found by mapping the output to `username` and the input to `to`.

```
r2: username = to.substring(0, to.indexOf('@'));
```

Deciding which results are actually relevant, rather than coincidental, is up to the developer and may not be straightforward. To help with this process, the developer can *provide additional input/output pairs to prune coincidental matches*. For example Adding the input/output pair, input_2 = "alex@univ.edu" and output_2 = "alex", will remove `r1` from the result set (i.e., it matches only the first input/output because the string "susie" has five characters), leaving `r2` as a more relevant result (see Section 5.1.1 for more details).

4.2.2 Mapping Input and Output

The previous example had an input and output that were both of the same type, string. However, we support multiple data types and the input and output types can be heterogeneous. For a programmer who wants to find the length of a file extension (including the dot), the input is a string (i.e., the name of file) and the output is an

integer (i.e., the extension length). Consider the example input string "foo.txt" and the integer output, 4. The following snippet is a match that involves four API calls:

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

Here, the input is mapped to the only undefined variable in the code snippet, `s` (inferred to be of type string). The output is mapped to the LHS of the final assignment statement, `len`, which is the only unused variable. These bindings are calculated by computing and exploring the def-use pairs in the snippet [46], where variables are marked as potential inputs while they are used but not defined.

There may be many potential mappings of an input/output specification to a code snippet. These mappings are critical for a piece of source code to be identified as a match. For specifications with multiple input values, swapping their order could be the difference between a piece of code that is a match, and one that is not. Generally, this process is left up the the solver (Section 5.2.2).

4.2.3 Partial Matches

When entire methods rather than blocks of source code are encoded, the input/output mapping is more straightforward than it is with a code snippet. That is, the input maps to the method parameters and the output to the return value.

However, if the method has multiple paths, an input/output pair may not require all parts of the source code. We can have a situation when an input/output pair matches only part of the source code, specifically, the path that would be executed.

Consider the following method, which returns the input string with the first letter capitalized. Encoding this method involves performing a symbolic execution to identify the path conditions.

```
1  String capitalizeFirstLetter(String in) {
2      if (in.length() >= 1) {
3          String t = in.toUpperCase();
4          String t2 = t.substring(0, 1);
5          String t3 = in.substring(1, in.length());
6          return t2.concat(t3);
7      } else {
8          return "";
9      }
10 }
```

Two paths are identified in this method, and each is encoded separately. The full encoding for this method, then, is a disjunction of the two paths. For the true branch of the predicate on line 2 (i.e., $in.length() \geq 1$), the following represents a rough encoding of the path:

```
c1. (assert (in.length() >= 1))
c2. (assert (t.length() = in.length()))
c3. (assert (t = in.toUpperCase()))
c4. (assert (t2.charAt(0) = t.charAt(0)))
c5. (assert (t2.length() = 1))
c6. (assert
       (for (1 <= i < in.length())
           (t3.charAt(i - 1) = in.charAt(i))))
c7. (assert (t3.length() = (in.length() - 1)))
c8. (assert (return = t2 + t3))
c9. (assert (return.length() = (t2.length() + t3.length())))
```

The predicate is represented by $c1$. Constraints $c2$ and $c3$ describe the relationship between variables t and in in terms of length and content (Section 6.2 describes how strings are handled in the encoding). A complete definition of the $toUpperCase()$ function is provided in Table A.5. Line 4 is represented by constraints $c4$ and $c5$, relating the length and content of strings t and $t2$. Representing line 5 is a little more complicated and requires a quantifier in $c6$ to set the content of $t3$ to be all except the

first character of in, and the length relationship is described by $c7$. Finally, the value of the return variable is described in $c8$ and $c9$, where the '+' operator is overloaded to represent concatenation of strings in $c8$.

For the false branch of the predicate on line 2, the following represents the encoding of that path:

```
c10. (assert (in.length() < 1))
c11. (assert (return = ""))
c12. (assert (return.length() = 0))
```

Here, the predicate is represented as $c10$, with constraints $c11$ and $c12$ asserting the length and content of the return value.

To identify this program as a match, a query could have an input of "abc" with an expected output of "Abc". This would satisfy the constraints for the first path; however, for the second path, the query would violate constraint $c10$ since the length of the input is not less than one. In this method, the input/output exercises only one path represented by constraints $c1 - c9$. As demonstrated in Section 4.2.1, the query could provide an additional example, such as the empty string as the input and the output, which would then exercise the second path.

Regardless, the presence of multiple paths in a program brings up some interesting questions. In the presence of multiple input/output pairs, we could have source code that matches some of the pairs, or input/output pairs that exercise only part of a piece of source code. These situations all impact the matching criteria for the approach, which is discussed in more detail in Section 5.3.

4.2.4 Using the Solver for Symbolic Variables

In some snippets, there may be additional variables that are used, but are not defined and not bound to the input. Consider the following snippet, which matches for the input/output used in the example in Section 4.2.2:

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

After mapping the input to `names` and the output to `index`, this code is not executable because we know nothing about the value of `flag`, so state-of-the-art semantic search engines that utilize test cases to identify matching code (e.g., [39, 50, 53]), would fail. In our approach, the uninitialized variables in the snippet remain uninitialized in the encoding, and we make no assumptions about the values they hold.[1] This snippet is identified as a match because the satisfiable model produced by the solver reveals that the specification matches this snippet when `flag` is set to ".txt" (the solver could have identified ".", ".t", or ".tx" as possible values, but it needs to find only one solution to complete the satisfiable model). By encoding the behavior of the snippets as constraints, *we can identify incomplete code as a match and leverage the solver to guide its instantiation.* Applying that guidance yields the following, modified and complete code:

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

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

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

[1] In our approach, the snippet is taken out of context so we must use type inference to reveal that `flag` is either a character or a string. We assume the more expressive case of string.

4.2.5 Poor Matches

The previous example showed a situation when a program could be instantiated to match the intention of the query. However, finding such a program is not always possible. Say a programmer wants to trim off the last character of a string. As an input, they provide the string, "admirer", the integer 1 since we're trimming one character, and the output string, "admire."[2].

In the repository we built for the evaluation in Section 7.2, there are 64 potential programs that take as an input an integer and a string, and return a string. However, none of these are results for the example. For small repositories of programs, or for very specific queries, this situation may not be uncommon.

Our approach to this issue is to use abstraction, and there are two types of abstraction we consider. The first is relaxing the type signature of the specification, and the second is abstracting the encoding of the source code. We leave the latter for future work. To abstract on the type signature, we can try to find matches with type signatures that subsume the specification yet have the required output type. Instead of matching every input to method parameters, this approach leaves some variables free. Performing such a relaxation returns 13 results, including the following:

```
String extractKey(String key, int start, String delegatePrefix, String
    prefix) {
    return delegatePrefix + key.substring(start + prefix.length(),
        key.length() - 1);
  }
```

Here, there are three string inputs and one integer input, with the output being a string. This means that when we map the input to the source code, there will be two free variables. This source code is a match for the input/output example with $key \mapsto$ "admirer" and $start \mapsto 1$. The solver will assign the following values

[2]This corresponds to question 11 used later in the evaluation (Table 7.8)

to variables, $delegatePrefix \mapsto$ "a" and $prefix \mapsto$ "". Now, this code does not necessarily match the intention of the specification. Since the integer input was set to 1, the solver had to set the value of $delegatePrefix$ to be the first character of the input string, since that character was chopped off in the *substring* call. This may not be the ideal solution, but through abstraction, our search was able to come up with some sort of result that could help guide the programmer in the right direction.

4.3 Yahoo! Pipes Mashups

In a Pipes mashup program, RSS feeds are manipulated and aggregated. The external data sources accessed by the *Fetch Feed* modules are the inputs to the pipe, so in our approach, the programmer provides the URLs for RSS feed(s) as input. This mimics how a programmer would build a pipe from scratch. Similar to the Pipes Editor environment, our framework fetches the RSS feed; this produces the input list. Rather than programming the rest of the pipe, with our approach, the programmer modifies this list directly by reordering, removing, or modifying items to form the output list.

An example of this process is shown in Figure 4.2. The programmer provides the URL, such as the *Input* in Figure 4.2, and our framework retrieves the RSS feed, which has n items. For this URL, $n = 35$, which leads to a very long input list. The programmer selects the third item as the desired output, forming an output list of size one.

For a programmer working in the Pipes environment, they have the ability to reuse an entire program by making a copy (i.e., cloning), or by importing an entire Pipe as a subroutine (i.e., a subpipe). The reuse of modules or sequences of modules is difficult since copying and pasting modules is not supported. Since the scope of reuse in the existing framework is at the whole program level, we maintain that level of abstraction

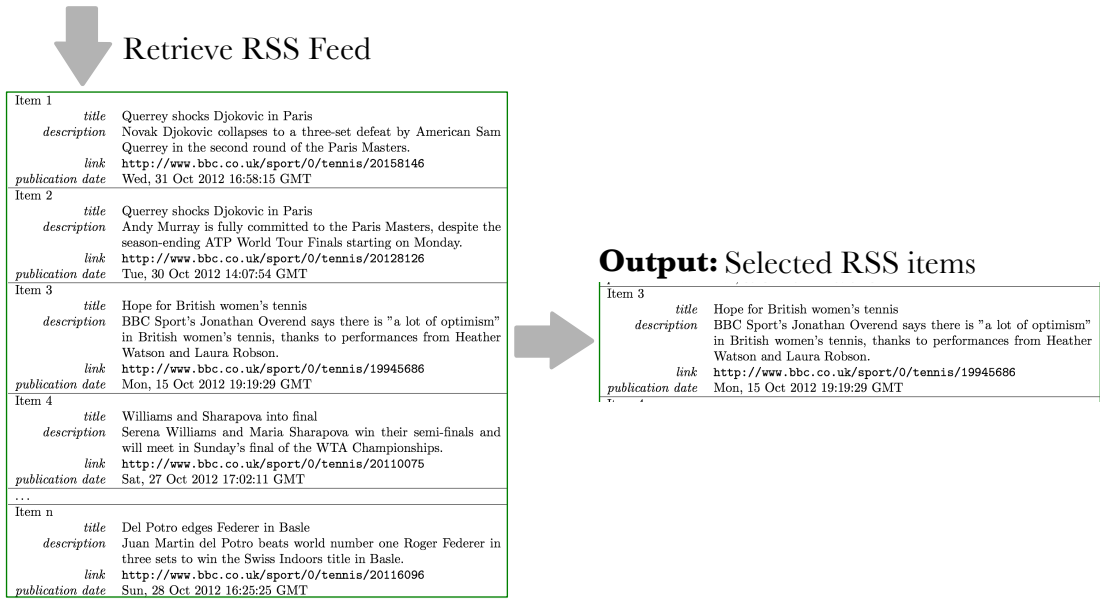
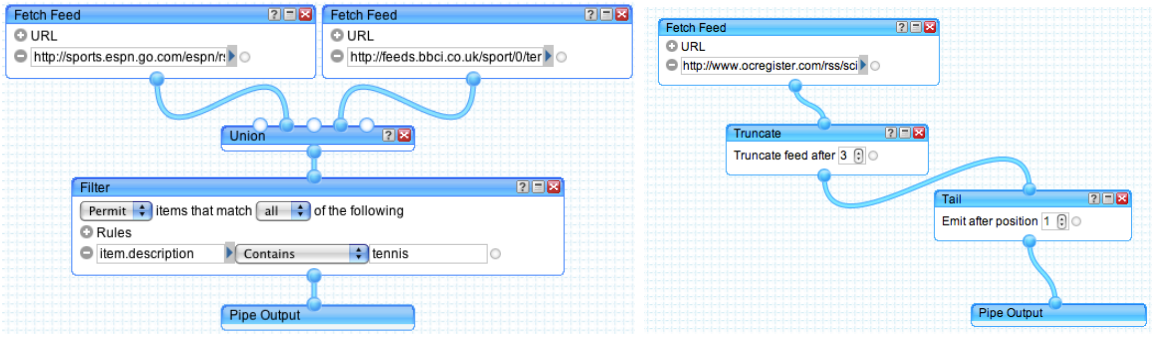


Figure 4.2: Yahoo! Pipes Input/Output Example

for the programmer. For our search in this domain, the programmer specifies the behavior of an entire program, and so entire programs are encoded as constraints and returned by the search (as opposed to snippets or methods, as in Java). Part of the future work is to consider partial programs to promote smaller-scale reuse.

4.3.1 Assigning Inputs to Fetch Modules

Say a programmer wants to collect news about tennis from a website, and created the specification shown in Figure 4.2 (the selected item for the output list contains "tennis" in the description). Using our approach to perform a semantic search over a repository of encoded Pipes programs (Section 8) reveals two possible matches, shown in Figure 4.3(a) and Figure 4.3(b). The solution in Figure 4.3(b) performs head and tail operations on the list to extract the third item, whereas the solution in Figure 4.3(a) joins two RSS feeds and permits items that have "tennis" in the



(a) (b)

Figure 4.3: Possible Solutions to Input/Output Example in Fig 4.2

description. While both match the specification, the solution in Figure 4.3(b) is likely a coincidental match, and could be pruned by adding another input/output example, as demonstrated by the example in Section 4.2.1.

Much like the example in Section 4.2.4 where the variable `flag` was symbolic, since the specification had only one input URL, the pipe solution in Figure 4.3(a) has an uninitialized input, URL_2. In this domain, instead of leaving the RSS feed symbolic, we assume unmapped fetch modules have empty input lists. That is, URL_2 is set to an empty list. This is necessary because the RSS feeds are external resources, and if left symbolic, the solver may identify a program as a match, but require an RSS feed that does not exist. Contrastingly in Java, the symbolic variables we currently consider are not dependent on an external resource and can receive arbitrary values assigned by the solver. As our language support increases to consider complex objects and parameters, this will no longer be the case and we may draw on this concept from the Yahoo! Pipes implementation.

Module	Encoding as Constraints
Fetch_1	c1: (assert (in(Fetch_1) = getRSS(URL_1)))
	c2: (assert (out(Fetch_1) = in(Fetch_1)))
Fetch_2	c3: (assert (in(Fetch_2) = getRSS(URL_2)))
	c4: (assert (out(Fetch_2) = in(Fetch_2)))
wire_1	c5: (assert (in$_1$(Union) = out(Fetch_1)))
wire_2	c6: (assert (in$_2$(Union) = out(Fetch_2)))
Union	c7a: (assert (for (0 $\leq$ i $<$ size(in$_1$(Union))) recOf(out(Union), i) = recOf(in$_1$(Union), i)))
	c7b: (assert (for (size(in$_1$(Union)) $\leq$ i $<$ (size(in$_1$(Union)) + size(in$_2$(Union)))) recOf(out(Union), i) = recOf(in$_2$(Union), (i - size(in$_1$(Union))))))
	c8: (assert (for (0 $\leq$ i $<$ size(out(Union))) (hasRec(in$_1$(Union), recOf(out(Union), i)) = true) $\vee$ (hasRec(in$_2$(Union), recOf(out(Union), i)) = true)))
	c9: (assert (((size(in$_1$(Union)) + size(in$_2$(Union))) = size(out(Union)))))
wire_3	c10: (assert (in(Filter) = out(Union)))
Filter	c11: (assert (for (0 $\leq$ i $<$ size(in(Filter))) ((recOf(in(Filter), i) = r) $\wedge$ contains(field(r "descr"), "tennis")) $\Rightarrow$ (hasRec(out(Filter), r) = true)))
	c12: (assert (for (0 $\leq$ i $<$ size(out(Filter))) (hasRec(in(Filter), recOf(out(Filter), i)) = true)))
	c13: (assert (for (0 $\leq$ i $<$ size(out(Filter))) (for (i $<$ j $<$ size(out(Filter))) ((recOf(out(Filter), i) = r_1) $\wedge$ (recOf(out(Filter), j) = r_2)) $\Rightarrow$ ($\exists$ k, l ((k $<$ l) $\wedge$ (0 $\leq$ k $<$ size(in(Filter))) $\wedge$ (0 $\leq$ l $<$ size(in(Filter))) $\wedge$ (recOf(in(Filter), k) = r_1) $\wedge$ (recOf(in(Filter), l) = r_2))))))
	c14: (assert (size(in(Filter)) >= size(out(Filter))))
wire_4	c15: (assert (in(Output) = out(Filter)))
Output	c16: (assert (out(Output) = in(Output)))

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

Figure 4.4: Mapping the Pipe in Figure 2.1 onto Constraints

4.3.2 Yahoo! Pipes Example: Encoding a Pipe

The encoding process is briefly illustrated in Figure 4.4 for the pipe in Figure 4.3(a). As in the Java domain, we require a mapping of program constructs to constraints representing their behavior. As modules have clear semantics and interfaces, in this domain, we map each module onto constraints and use wires to define the relationships

between modules. A full description of that process is provided in Section 6.4, but we provide the intuition here.

Module constraints are expressed in terms of the input to and output from the module (e.g., `in(Filter)` refers to the list of items that enters the *Filter* module, and `out(Filter)` refers to the list of items that exits the *Filter* module). Constraints $c1$ and $c3$ assign input variables (i.e., URL_1 and URL_2), to each of the *Fetch Feed* (succinctly, *Fetch*) modules. The $getRSS()$ function retrieves a list of RSS items. Constraints $c2$ and $c4$ ensure that the output from the *Fetch* modules are the same as the input. When a pipe is encoded, the URL information is abstracted away so the pipe can be solved for any URL provided as input; this abstraction is imperative so the programmer can find pipes that behave as desired given their defined input and output. This is illustrated in the example with the URL_1 and URL_2 symbolic names in constraints $c2$ and $c4$. This process is unique to Yahoo! Pipes since the inputs to the program are specified concretely, unlike a Java program where the inputs are passed as arguments to a method (for example).

Constraints $c5$ and $c6$ connect the output from the *Fetch* modules to the *Union* module as inputs. The *Union* module concatenates its input lists, which is described by constraints $c7a$, $c7b$, $c8$, and $c9$. The first constraint, $c7a$, ensures that all the items at the front of the output list, `out(Union)` come from the first input list, $\texttt{in}_1\texttt{(Union)}$, and the second constraint, $c7b$, ensures that the next items are from $\texttt{in}_2\texttt{(Union)}$. This is called *inclusion*. The $recOf(L, i)$ function is an accessor to get the record of a list L at index i. The next constraint, $c8$, ensures that all items in the output list from the module exist in one of the two input lists, and in this way no extra items are appended to the end of the list. This constraint enforces *exclusion*. The last constraint for this module, $c9$, ensures that the size of the `out(Union)` is equal to the sum of the sizes of the input lists. This is a *size* constraint. The output from the *Union*

module goes to the *Filter* module per $c10$. Representing the *Filter* module requires four constraints that enforce *inclusion*, *exclusion*, *order*, and *size* properties. The first, $c11$, ensures that all items in `in(Filter)` that contain "tennis" in the description also exist in the `out(Filter)` list. The *exclusion* constraint, $c12$, ensures that all records in the output are also from the input (i.e., none were added and `out(Filter)` $\subseteq$ `in(Filter)`). The next constraint for this module, $c13$, ensures that if two records exist in the output list, their ordering is the same as it was in the input list. In this way, the module is order-preserving. The final constraint for the *Filter* module, $c14$, ensures that the output list is at most as long as the input list. Constraint $c15$ ensures that the output from the *Filter* module goes to the input of the *Output* module, and $c16$ ensures that the output of the pipe, `out(Output)` is the same as `in(Output)`.

4.3.3 Abstracting String Values

When a matching program cannot be found, we can apply abstractions to the encodings to *find code that does not exist as such*, but is an approximate match and can be instantiated to meet the user specifications. For example, say a programmer wants to filter an RSS feed based on "volleyball" rather than "tennis". The inclusion constraint for the *Filter* module in Figure 2.1, $c10$, contains as part of the implication, `contains(field(r` "descr"`),` "tennis"`))`. At a concrete abstraction level, the string "tennis" is encoded as is, which would not satisfy a specification that requires "volleyball". However, with a weaker encoding consisting of constraint `contains(field(r` "descr"`), s))` for some string s, an SMT solver could determine that for $s =$ "volleyball", this program is a match.

This form of abstraction allows the search to identify programs that are *approximate* matches for the desired behavior, and can be modified systematically to satisfy the

input/output specifications, similar to the example in Section 4.2.4 where `flag` was instantiated. We implement and evaluate two abstraction levels within the pipes domain (Section 8).

4.4 SQL Select Statements

When instantiating our approach for SQL, the input and output take the form of database table(s). The indexed SQL select statements are encoded as constraints. Given example tables as input and output, the SMT solver determines, for each encoded SQL select statement, if it matches the specification.

4.4.1 Large Specifications

Consider the programmer who asked the question on stackoverflow, "I have table with records 'user' and 'balance'. How to show 10 usernames with highest balance? ... How to show but only when they have more than 1.000.000$?"[3] The programmer knew the desired behavior and described it through a concrete input example table:

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

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

[3] `http://stackoverflow.com/questions/11599636`

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

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

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

An interesting aspect of this domain is that the *input/output specifications can be large* since they may come from live databases. It then becomes important to understand the impact of large specifications on solver time. As we have explored in previous work, but have not presented as part of this paper, it is not just the specification size that matters but also the complexity of behavior exhibited in the specification [70].

4.4.2 Implicit Join on Inputs

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

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

This query requires an implicit join on the `id` field for the two input tables in order to create the output table. Our approach supports the case when *multiple inputs form a*

single output, a concept we explore further in Section 5.1. Such a query is common and useful in any of the supported domains (e.g., multiple strings in Java, multiple URLs in Yahoo! Pipes, or multiple tables in SQL), and is common in database queries that require merging multiple tables.

4.5 Summary

In this chapter, we have discussed several interesting aspects of our approach to semantic code search in three domains, illustrating the generality of the approach. Chapter 3 revealed some opportunities to better support code search for programmers, and we specifically focus on the opportunities to allow programmers to specify search queries as examples and to build repositories to search over using existing code. We have also shown how our search is flexible in finding programs that are partial matches and programs that are close to the desired behavior when intended solutions are not available. We have also shown how a programmer can identify relevant code when *there are many coincidental matches* by adding additional input/output examples and how we can return source code that provides a *partial* match for a specification or for a specification that only *partially* matches source code. In the next chapter, we define this approach to semantic code search more formally.

Chapter 5

Approach

In this section, we present the general definitions of each piece of our approach to semantic search via SMT solvers. Illustrated in Figure 5.1, the gray boxes indicate the key components and technical challenges: specifying behavior by defining *lightweight specifications* as input/output pairs, *encoding* programs and specifications as constraints, returning and ranking results, and refining program encodings and specifications when *too few* or *too many* matches are found. The crawling and program encoding processes happen *offline*, meaning those activities do not impact the search time, whereas invoking the solver to identify relevant code happens *online*, meaning this activity does impact the search time. We now describe each component in detail.

5.1 Specifying Behavior

Instead of textual queries to find syntactic matches, our approach takes lightweight specifications that characterize the desired behavior of the code (*Lightweight Specifications* in Figure 5.1). These specifications, LS, are lightweight and incomplete in

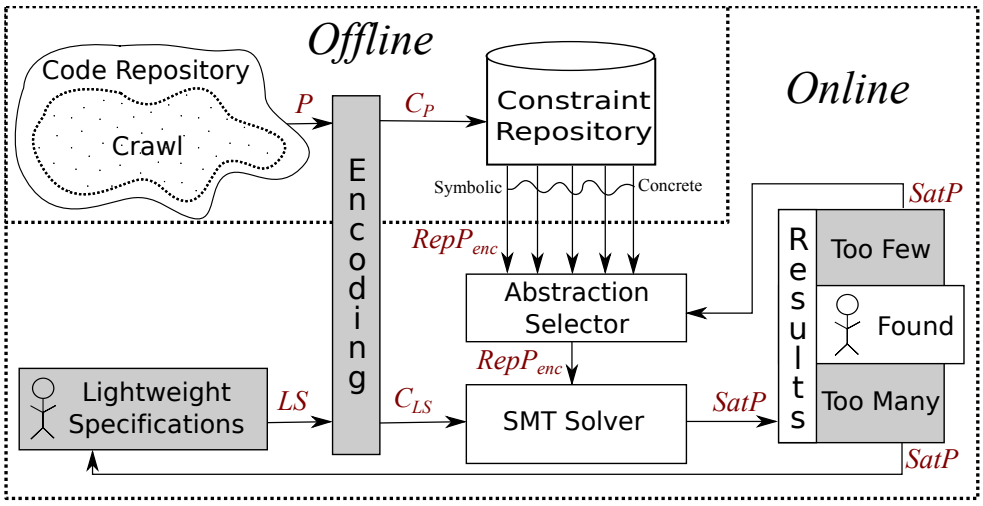


Figure 5.1: Detailed View of Approach

that they consist of concrete input/output pairs that exemplify part of the desired system behavior, like "susie@mail.com" and "susie" from Chapter 1.

To more completely specify the desired behavior, multiple input/output pairs can be defined: $LS = \{ls_1, ls_2, \ldots, ls_k\}$, for k pairs where $ls_j = (I_j, O_j)$. Each input, I_j, is a set of elements and each output, O_j, is a set of elements related to I_j. Each element of the input and the output also has a defined type, which is used in the solving phase to identify potentially matching programs (Section 5.3).

For example, say a programmer is looking for a Java program that determines if a string contains another string, case insensitive. This requires two inputs, both of type string, and one output of type boolean. LS in Listing 5.1 illustrates two input/output pairs.

Listing 5.1: Lightweight Specification Example

```
LS = { ({"Foo", "fO"}, {true}), ({"abc", "def"}, {false}) }
```

Here, ls_1 is a positive example returning true, and ls_2 is an example returning false. The inputs for both pairs, I_1 and I_2, each have two elements, both strings, and the outputs, O_1 and O_2, both contain a single element of type boolean.

The elements that compose the input and output take different forms depending on the domain. The examples used until this point, as well as our current implementation, restrict the output to a single element. Including multiple outputs is part of future work described in Section 11.2.4. For our current implementation of the approach in Java (Chapter 6), the input is a non-empty set of booleans, characters, integers and strings, and the output is either a boolean, character, integer, or string. In Yahoo! Pipes, the input is a non-empty set of URLs (from which the RSS feeds are retrieved) and the output is a list of items. For SQL select statements, the input is a non-empty set of tables and the output is a table. Conceptually, the input and the output can be any element that can be used to illustrate the behavior of desired code. Each instantiation of the approach defines what forms these elements can take. The size of k defines, in part, the strength of the specifications and hence the number of potential matches.

If the specifications or the encoded program constraints are too weak, many matches may be returned (*too many* in Figure 5.1); if they are too strong, the solver may not yield any results (*too few*). Refinement and abstraction are processes that help to address these situations, respectively.

5.1.1 Specification Refinement

As illustrated in Section 4.2.1, a programmer can provide specifications incrementally, starting with a small number of pairs and adding more to further constrain the search.

This incremental process would follow the arrow from *Too Many* results back to *Lightweight Specifications* in Figure 5.1.

To provide an example, given the specification from Listing 5.1, Listing 5.2 is a potential match. It matches both input/output pairs in LS, but simply checks the lengths of the input strings and misses the objective of the query. On the other hand, Listing 5.3 matches LS and the objective of the query in that it converts both input strings to lowercase and then checks for containment. However, this result may be buried below many irrelevant results.

Listing 5.2: Specification Refinement Example Result 1

```
String notSameLength(String a, String b) {
    return a.length() != b.length();
}
```

Listing 5.3: Specification Refinement Example Result 2

```
boolean checkContains(String input1, String input2) {
    String inA = input1.toLowerCase();
    String inB = input2.toLowerCase();
    return inA.contains(inB);
}
```

This is an instance when specification refinement would be helpful as adding another input/output pair can differentiate between these two relevant results. This process is defined as $adding(LS,\ ls_{k+1}) \rightarrow LS'$, where ls_{k+1} is the newly added input/output pair to LS, forming LS'. For the previous example, we suggest two directions to differentiate among Listing 5.2 and Listing 5.3 as results for LS in Listing 5.1. The first, $adding(LS, (\{\text{“abc”},\ \text{“ABC”}\},\ \{true\})) \rightarrow LS'$, would rule out Listing 5.3 because the strings have the same length, and yet the output is $true$. Another option would be $adding(LS, (\{\text{“abc”},\ \text{“d”}\},\ \{false\})) \rightarrow LS'$, which also rules out Listing 5.3 since the lengths are different and the output is $false$.

Alternatively, the programmer can replace an input/output pair with one that captures a more unique or distinguishing aspect of the desired behavior. This process is defined as $replace(LS,\ ls_i,\ ls_j) \rightarrow LS'$, where the specification ls_i is replaced by ls_j. As an example, the following *replace* call would differentiate between Listing 5.2 and Listing 5.3 by replacing the positive example with one in which the two input strings have equal length, as follows:

$replace(LS,\ (\{\text{"Foo"}, \text{"fO"}\}, \{true\}),\ (\{\text{“abc”}, \text{“ABC”}\}, \{true\})) \rightarrow LS'$.

Providing guidance on this process is part of the future work described in Section 11.2.1.

5.1.2 Specification Abstraction

If there are no matches for a query, the programmer has the option to weaken the specifications. This can occur when there truly is no match in the repository, or when the search takes too long to provide a response. An example of this last case occurs when the tables provided as input for SQL have hundreds of rows causing the solving time to take minutes; in this case it may be useful to select the subset of the table that still captures the key desired behavior. A similar phenomenon can be observed with Yahoo! Pipes when there are many items in the input and output lists, or in Java when performing multiple operations over very long strings. This process is defined as, $remove(LS,\ ls_i) \rightarrow LS'$, which is the reverse of *adding* in that an input/output pair, ls_i, is removed. Alternatively, $replace(LS,\ ls_i,\ ls_j) \rightarrow LS'$ can be used to capture a different set of behavior by replacing ls_i with ls_j. Alternative approaches consider negative examples or regular expressions, which we leave for future work (Section 11.1.3)

5.1.3 Specification Encoding

Prior to solving, the specifications LS must be encoded, transforming LS into $C_{LS} = \{C_{ls_1}, C_{ls_2}, \ldots, C_{ls_k}\}$, where each C_{ls_j} represents the constraints for an input/output pair ls_j, and C_{LS} is the set of encoded input/output pairs.

For example, consider the LS in Listing 5.1. Each $ls \in LS$ is encoded separately. Listing 5.4 shows C_{ls_1}, the encoding for ls_1; C_{ls_2}, the encoding for ls_2, is in Listing 5.5.

Listing 5.4: C_{ls_1}: Encoded Lightweight Specification ls_1 from Listing 5.1

```
1 // ls = ({"Foo", "fO"}, {true})
2 (declare-fun input1 () String)
3 (declare-fun input2 () String)
4 (declare-fun output () Boolean)
5
6 (assert (= input1 "Foo"))
7 (assert (= input2 "fO"))
8 (assert (= output true))
```

Listing 5.5: C_{ls_2}: Encoded Lightweight Specification ls_2 from Listing 5.1

```
1 // ls = ({"abc", "def"}, {false})
2 (declare-fun input1 () String)
3 (declare-fun input2 () String)
4 (declare-fun output () Boolean)
5
6 (assert (= input1 "abc"))
7 (assert (= input2 "def"))
8 (assert (= output false))
```

In Listing 5.4, the input and output variables are declared on lines 2-4. Lines 6-8 assert the values of the specifications. The structure is identical for Listing 5.5. Since no variable names are provided as part of the specification definition, arbitrary names are always assigned, (e.g., `input1`). The binding of these variables to program variables, which is necessary for solving, happens as part of the solving phase (Section 5.3).

5.2 Encoding

In our approach, encoding and solving are analogous to crawling and indexing performed by search engines [38]. Offline, a repository (*Code Repository* in Figure 5.1) is crawled to collect programs. These programs are parsed, then encoded as constraints using symbolic analysis [7, 8, 36], and the constraints are stored in a *Constraint Repository*.

The process of encoding takes a collected set of programs $RepP = \{P_1, P_2, \ldots P_n\}$ and transforms it into a collection of encoded programs, $RepP_{enc} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$. Two general steps are involved, program transformation and program input/output identification, as shown by the pseudocode in Listing 5.6. For all programs in a repository, and all paths in those programs, the encoding process transforms each path into a set of constraints. An encoded program is a set of its encoded paths, and an encoded repository is a set of encoded programs.

Listing 5.6: Encoding Process Overview

```
1  encodeRepository(Repository RepP) {
2      RepP_enc ← {} // RepP_enc is a set of programs, each encoded
3
4      for(Program P : RepP) {
5          P_enc ← {} // P_enc is a set of paths in P, each encoded
6
7          for(Path q : P) {
8              C_q ← {} // An encoded path, C_q, is a set of constraints
9              C_q ← transformToConstraints(ssa(q))
10             C_q ← C_q ∪ addInputOutputConstraints(ssa(q))
11
12             P_enc ← P_enc ∨ {C_q} // add encoded path as disjunction to P_enc
13         }
14
15         RepP_enc ← RepP_enc ∪ {P_enc} // add encoded program to RepP_enc
16     }
17     return RepP_enc
18 }
```

Each step of the process is described in the sections that follow, considering the Java method in Listing 5.7 for illustration. Java was selected for illustration due it its complexity and for ease of presentation. This method was formulated to illustrate the challenges and opportunities that arise in the Java language. It takes a string, `foo`, as input and returns a string. If `foo` contains the string "x", then the substring from 0 to the index of "x" is returned. If not, an unmodified `foo` is returned.

Listing 5.7: Approach Encoding Example

```
1 static String bar(String foo) {
2     if(foo.contains("x")) {
3         int i = foo.indexOf("x");
4         foo = foo.substring(0, i);
5     }
6     return foo;
7 }
```

5.2.1 Program Transformation

The transformation process happens at the path level in a program. Since the current specification model requires concrete input and output values, each will exercise a single path, as illustrated with the example in Section 4.2.3.

To facilitate the path-level analysis, we take advantage of symbolic analysis. For each program $P \in RepP$, a symbolic execution engine traverses the paths in P so each can be encoded separately. For convenience, these paths are identified as $Q_P = \{q_1, q_2, \ldots, q_m\}$ for m paths in P. This process is implied as part of the for-loop on line 7 in Listing 5.6.

To illustrate with program P in Listing 5.7, the symbolic execution tree is shown in Figure 5.2. There is one predicate in the program, and so there are two path conditions and two branches. These two branches form the set of paths for P, denoted

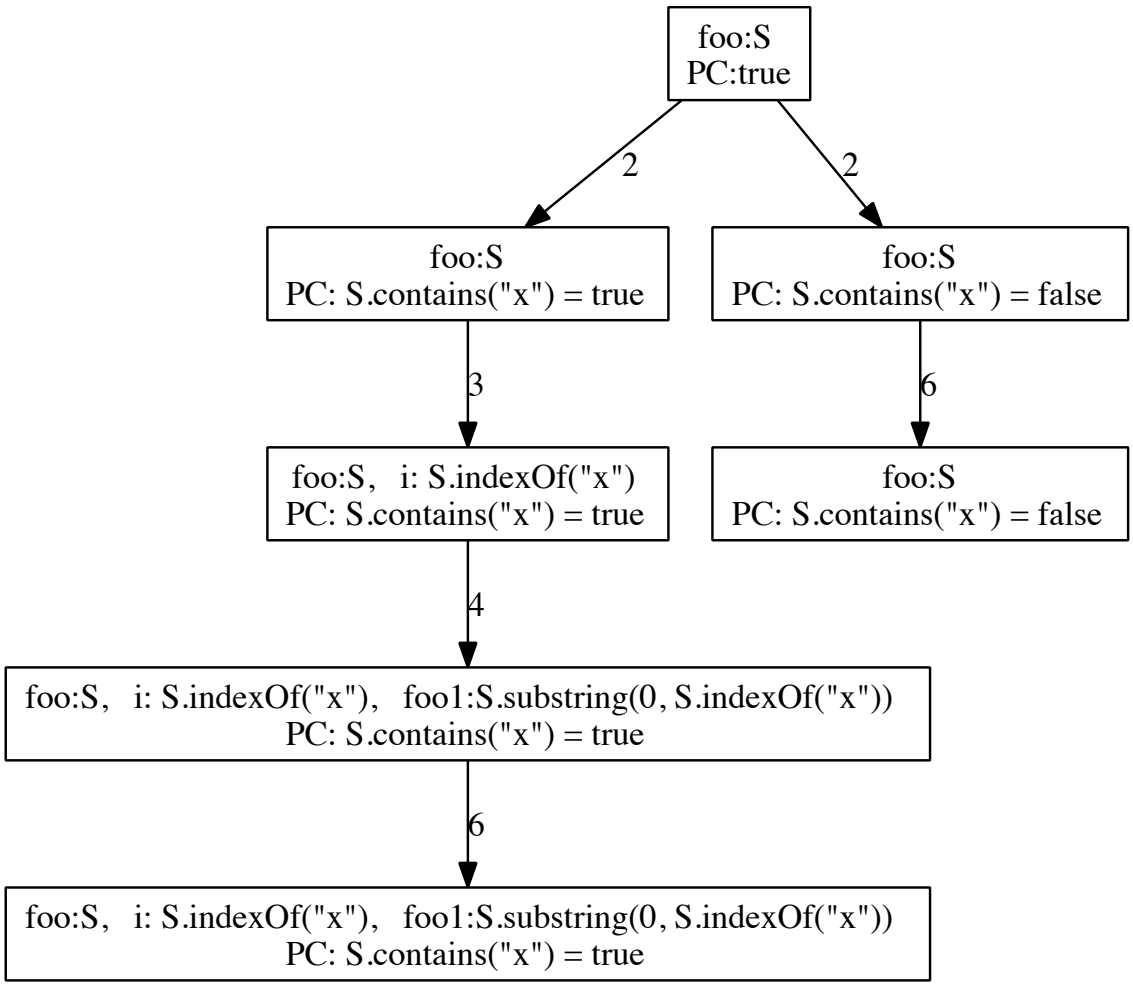


Figure 5.2: Symbolic Execution of Program in Listing 5.7

$Q_P = \{q_1, q_2\}$. From the symbolic execution tree, path encodings are built at each leaf.

The path condition is set to *true* at the beginning of the method and it is refined every time a predicate is reached. In the left leaf in Figure 5.2, three variables are defined, foo, i, and $foo1$. The path condition is set on the transition of line 2, where $PC : S.contains(\text{“x”}) = true$. In the original program, foo is redefined on line 4 in Listing 5.7, thus, a new variable is introduced, $foo1$, to represent the new value,

$foo1 : S.substring(0, S.indexOf(\text{“x”}))$. In essence, this converts the program into static single assignment form (SSA) form. As variables are introduced and manipulated throughout the program, their domains are updated in the symbolic execution tree. For example, line 3 introduces i, and so i is defined after the *3* transition in Figure 5.2. Line 4 re-defines foo, so $foo1$ is introduced after the *4* transition in Figure 5.2.

At the end, the domains on the variables represent the restrictions that have been imposed on their values throughout the symbolic execution of this program path, dictated by the program statements that are executed along the path. The path condition sets the conditions under which this path is executed, and the variables used in the path conditions are symbolic. Let q_1 in Listing 5.8 represent the path for the left branch in Figure 5.2. Lines 1-3 define the variables, the path condition is on line 4, lines 5-6 manipulate the variable domains, and the return statement is on line 7. Since foo is symbolic and unrestricted during the symbolic execution of the method, in the paths, foo is declared in Listing 5.8, but not defined, leaving it symbolic.

Listing 5.8: Path q_1 for Listing 5.7

```
1 String foo;
2 String foo1;
3 int i;
4 pc1: foo.contains("x") == true
5 i = foo.indexOf("x");
6 foo1 = foo.substring(0, i);
7 return foo1;
```

The right leaf of Figure 5.2 corresponds to the path condition where $PC : S.contains(\text{“x”}) = false$. Here, no manipulations are made to the domain of foo, and no additional variables are introduced. Let q_2 in Listing 5.9 represent the path for the right branch in Figure 5.2. Line 1 defines the variable foo, line 2 defines the path condition, and line 3 is the return statement.

Listing 5.9: Path q_2 for Listing 5.7

```
String foo;
pc2: foo.contains("x") == false
return foo;
```

Listing 5.10: C_{q_1}: Encoding of path q_1 from Listing 5.8

```
// declare variables
(declare-fun foo () String)
(declare-fun foo1 () String)
(declare-fun param1 () String)
(declare-fun param2 () String)
(declare-fun i () Int)
(declare-fun lower () Int)

// path condition
(assert (= foo.contains(param1) true))

//variable values
(assert (= foo.indexOf(param2) i))
(assert (= param1 "x"))
(assert (= param2 "x"))
(assert (= foo.substring(lower, i) foo1))
(assert (= lower 0))
```

Listing 5.11: C_{q_2}: Encoding of path q_2 from Listing 5.9

```
// declare variables
(declare-fun foo () String)
(declare-fun param1 () String)

// path condition
(assert (= foo.contains(param1) false))

//variable values
(assert (= param1 "x"))
```

The output of the symbolic execution stage of the analysis are the path conditions and variable domain restrictions based on the executed statements, shown in Listing 5.8 and Listing 5.9 for the method in Listing 5.7.

An encoded program P_{enc} is a set of its encoded paths. So, each path is transformed into constraints representing the path logic. Since our approach is meant to support many languages, a different encoding scheme is required for each language (see Chapter 6). The transformation process for each path begins on line 8 of the algorithm in Listing 5.6.

Each encoded path is represented in conjunctive normal form, $C_q = \{c_1 \wedge c_2 \wedge \cdots \wedge c_o\}$ for the o constraints. Continuing with the example, rough representations of the encodings for the paths from Listing 5.7 are provided in prefix notation (see Chapter 6 for specifics). Listing 5.10 roughly represents the constraints for q_1 and Listing 5.11 roughly represents the constraints for q_2. Java-style comments are included to differentiate between variable declarations and encoded program logic.

This process is repeated for every path q to create an encoded representation of P as a disjunction of encoded paths; $C_P = \{C_{q_1} \vee C_{q_2} \vee \cdots \vee C_{q_m}\}$ for the m paths in P. With the example, $C_P = \{C_{q_1} \vee C_{q_2}\}$. In the end, the encoding process maps every program to a set of constraints such that $RepP_{enc} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$. The aggregation of $RepP_{enc}$ is shown on line 15 of Listing 5.6.

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 $C_P = \{true\}$. These extremes correspond to the least and the greatest number of matches and the worst and the best search speeds respectively, but there is a spectrum of choices in between. Two levels are explored in Chapter 6, specifically encoding at the component level (Yahoo! Pipes) and the library level (Java).

5.2.2 Mapping Input and Output

For each path in $q \in Q_p$, the next step is to identify its input and output, creating an input/output signature and mapping for when the path is paired with some LS during solving. Line 10 in Listing 5.6 represents this process. The input/output is defined and mapped for paths rather than for whole programs since the transformation to SSA may require that the output be mapped to different variables in different paths (i.e., the ϕ function in SSA serves this purpose).

In general, an input could be any memory location, web resource, file, or other entity containing information for a program that is read, but not written. An output is potentially any variable, file, or memory location that is written. For most programs, the input and output are symbolic by default. For example, variables `input1`, `input2` and the return statement in Listing 5.3 do not have concrete values unless the program is executed. In the case of a program that has hard-coded inputs, as for a Yahoo! Pipes program which has concrete URLs embedded in the `Fetch Feed` modules, the input must be relaxed and encoded symbolically so the program can be matched against any arbitrary LS with matching types.

In Java, the inputs are identified as variables that are used but not defined. The output is the result of a return statement, or in the case of a code block, the left-hand side of an assignment statement. For branching structures, the arity of the input/output may vary with paths depending on the variables that are used. In Yahoo! Pipes, the inputs are the lists of items flowing out of `Fetch Feed` modules and the output is the list of items flowing out of the `Pipe Output` module. In SQL, the inputs are the tables accessed in the `FROM` clause of the select statement, and the output is the resulting table. By identifying the inputs and outputs from a program path q, an

input/output signature is created and mapped onto the program variables to prepare for binding with an arbitrary LS.

Continuing with the Java example, for paths q_1 and q_2, the input is identified as `foo`, which is the only variable that is used but not defined. The output is mapped to the return statements in the paths. In this example, the output is different for each path. For q_1, the output is `foo1` where as for q_2, the output is `foo`. In both cases, the output is of type string.

Once the input/output variables have been identified, the next step is to bind the variables to new input/output variables that will be instantiated when the path is bound to an LS in the solving phase. The mapping constraints for q_1 are shown in Listing 5.12 and for q_2 in Listing 5.13. The new variables are `input` and `output`, both of type string like their counterparts in the program encoding.

Listing 5.12: Input/Output Mapping for path q_1 from Listing 5.8

```
// declare input and output variables
(declare-fun input () String)
(declare-fun output () String)

// map input and output to program variables
(assert (= input foo))
(assert (= output foo1))
```

Listing 5.13: Input/Output Mapping for path q_2 from Listing 5.9

```
// declare input and output variables
(declare-fun input () String)
(declare-fun output () String)

// map input and output to program variables
(assert (= input foo))
(assert (= output foo))
```

While not the case with this example, if there are multiple inputs in a path q with the same type, the mapping is a disjunction of all possible bijections of the input in LS

to the input variables in q. This allows the solver to identify bindings of specifications to program variables such that the program can match the specification. For this reason, the ordering of the elements in an input do not matter. An example of this situation is shown in Section 5.2.3 with Listing 5.15.

The last step is to add the mapping constraints to the encoded path constraints, as shown with the union operator on Line 10 in Listing 5.6. This completes the path encoding C_q so it can be added to P_{enc}, as shown in Line 12 of Listing 5.6. Note that every encoded path is a set of constraints, and so P_{enc} is a set of sets of constraints.

In the solving phase when a path q is matched to an LS, there needs to be type matching on the inputs and outputs in q to the types in LS. For this reason, we create a type signature for q, similar to an input/output pair in LS. The difference is that the type signature contains variable names, rather than values, and type annotations. The values for these variables are set when the path is combined with a specification and sent to the solver. For example, with q_1, the type signature $TS_{q_1} = (\{\text{input:String}\}, \{\text{output:String}\})$. The type signature for q_2 is identical.

Type signatures are attached to paths rather than programs as the signatures can change in the presence of abstraction, and can also be different across paths in a program depending on the variables that are used. More discussion on this follows in Section 5.2.3.1.

5.2.3 Abstraction Selection

Encoding occurs at an abstraction level (*Abstraction Selector* in Figure 5.1). The initial encodings are as concrete as possible, as shown in Listing 5.10 and Listing 5.11, but weaker encodings that replace concrete values with symbolic ones, as discussed with the example in Section 4.3.3, are also computed and used. Weaker encodings can

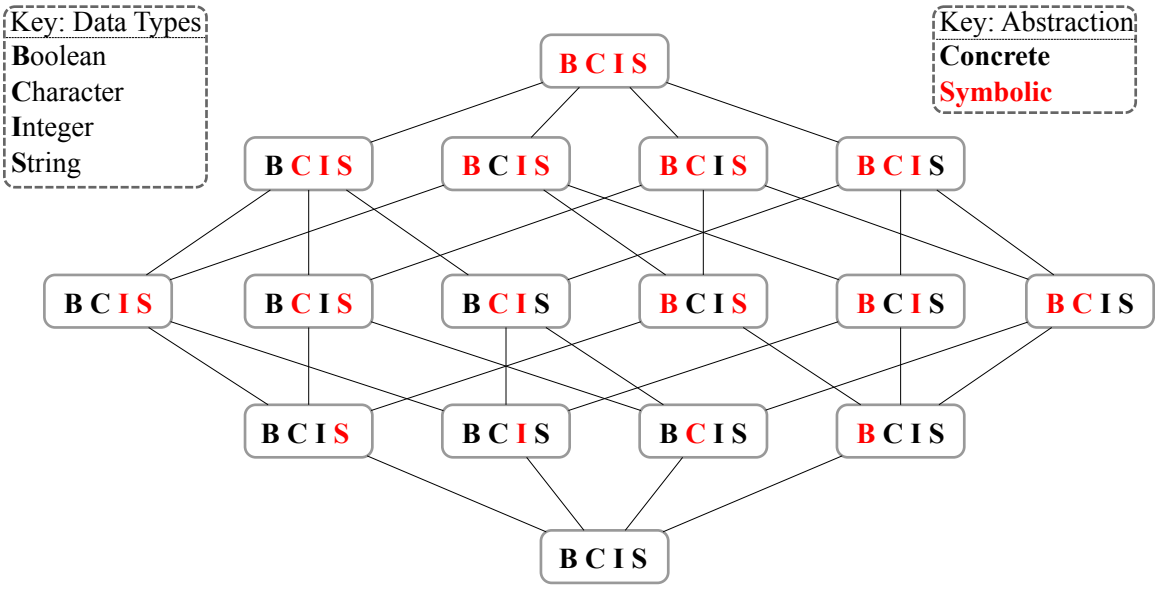


Figure 5.3: Type Lattice Example

approximate program behavior and be useful when searching over small repositories or in the presence of strong specifications. We exploit the fact that most languages contain constraints over multiple data types (e.g., strings, characters, integers, booleans) for which the variable values can be relaxed and encoded as symbolic.

An abstraction lattice, shown in Figure 5.3, guides the abstraction selection. This lattice shows the potential abstraction levels, based on data type, for the types supported in our Java implementation. While this lattice is defined for types, it could also consider abstraction in variable instances; defining such additional lattices is part of future work.A subset of the lattice also applies for Yahoo! Pipes and SQL.[1] The lattice provides a convenient structure to show the relationships among abstraction levels. At the bottom, the programs are encoded as concretely as possible. By moving up the abstraction lattice, the encodings relax and programs have the potential to satisfy a broader set of specifications.

[1]The Yahoo! Pipes and SQL languages take advantage of strings and integers, so the lattice subset includes just those datatypes.

A link between two levels in the lattice indicates that the lower level is weaker than the upper level. That is, the upper level subsumes that lower level in that it has the potential to match a broader set of specifications. Bottom levels represent lower levels of abstraction (i.e., more concrete values, weaker specifications) and the top levels represent higher levels of abstraction (i.e., more symbolic values, stronger specifications).

Moving downward in the lattice is a *weakening* move; in our context, that means that some programs that match a specification at the upper level will not match at the lower level. Conversely, moving upward in the lattice is a *strengthening* move, where there may not exist a program that matches a specification at a lower level, but there may exist a program that will match at a higher level. Drawing on the lattice in Figure 5.3, moving from **BCIS** at the top, down to **BCIS**, which is down and all the way left, is a weakening move that toggles the booleans from symbolic to concrete.

5.2.3.1 Weakening Example in Java

With the encoded path q_1 in Listing 5.10, `foo` is encoded symbolically, but `param1` and `param2` are encoded concretely. A weaker encoding simply defines some, or all, concrete variables to be symbolic. In Listing 5.10, removing the constraints on lines 12 and 13 would turn `param1` and `param2` from concrete to symbolic encodings, obtaining a more general version of that program. This would be equivalent to adding two additional arguments to the program in Listing 5.7, forming the modified program in Listing 5.14, where *param*1 and *param*2 are now symbolic as part of the input.

In the abstraction lattice in Figure 5.3, the initial encoding of q_1 corresponds to the bottom level of the lattice, **BCIS**. Once the strings are abstracted and the constraints removed, the encoding moved up the abstraction lattice and to the left where strings are made symbolic, **BCIS**. However, other data types could be abstracted in q_1, such

as the the integer variable `lower`. From **BCIS**, abstracting the integers would move up the abstraction lattice and to the left, landing at **BCIS**. Since there are no concrete character or boolean variables in q_1, the **BCIS** level is the same as all those that subsume it (i.e., those connected and above).

Listing 5.14: Approach Encoding Example

```
1 static String bar(String foo, String param1, String param2) {
2     if(foo.contains(param1)) {
3         int i = foo.indexOf(param2);
4         foo = foo.substring(0, i);
5     }
6     return foo;
7 }
```

For consistency, variables weakened in q_1 must also be weakened in q_2. Otherwise, the two paths, which came from the same program, would not be representative of that same program at the higher abstraction level. Since `param1` is the only common, weakened variable between q_1 and q_2, only line 7 in Listing 5.11 needs to be removed.

To show the benefits of a weakened encoding, consider the following two specifications, LS_1 and LS_2:

```
LS1 = { ({"cowxboy"}, {"cow"})}
LS2 = { ({"cow-boy"}, {"cow"})}
```

LS_1 will match the concrete encoding of q_1 in Listing 5.10 and the weakened encoding in which `param1` and `param2` are made symbolic. In both encodings, I_1 = "cowxboy" is mapped to `foo`. Considering LS_2, the more concrete encoding of q_1 would not match. However, with the weakened version of q_1, there would be a match since `param1` and `param2` would be treated as symbolic and could match any string.[2] Abstraction can be very powerful in this approach to code search, finding code that is *close enough* to a match in that it can be changed slightly to match the specifications.

[2]In the solving phase, the satisfiable model would reveal that `param1 =` "-" and `param2 =` "-". See Section 5.3 for details.

One consequence of weakening the string variables is that `param1` and `param2` become potential inputs, which changes the type signature of q_1, forming TS'_{q_1}. That is, TS'_{q_1} = ({input1:String, input2:String, input3:String}, {output:String}). Additionally, the input/output mapping changes as a result of the new set of inputs. The new mappings are shown in Listing 5.15.

Listing 5.15: More Abstract Input/Output Mapping for path q_1

```
// declare input and output variables
(declare-fun input1 () String)
(declare-fun input2 () String)
(declare-fun input3 () String)
(declare-fun output () String)

// map input and output to program variables
(assert (or
    (and (= input1 foo) (= input2 param1) (= input3 param2))
    (and (= input1 foo) (= input3 param1) (= input2 param2))
    (and (= input2 foo) (= input1 param1) (= input3 param2))
    (and (= input2 foo) (= input3 param1) (= input1 param2))
    (and (= input3 foo) (= input1 param1) (= input2 param2))
    (and (= input3 foo) (= input2 param1) (= input1 param2))))
(assert (= output foo1))
```

With the addition of symbolic string variables, `param1` and `param2`, assigning inputs to program variables becomes more complex. Now that there are three inputs of type String in q_1, all possible assignments of input string variables to the symbolic program variables in q_1 must be considered, so all possible bijections of the inputs from the type signature to the input variables in q_1 are encoded. For this example, there are $3! = 6$ possible mappings. The bijections are computed *within type* to prevent type mismatch. That is, if q_1 also had two integer inputs, then the integer bijection and mapping would be computed independently of the string mapping.

Similary, the type signature of q_2 also changes to TS'_{q_2} = ({input1:String, input2:String}, output:String), and the revised mapping is shown in Listing 5.16.

Listing 5.16: More Abstract Input/Output Mapping for path q_2

```
// declare input and output variables
(declare-fun input1 () String)
(declare-fun input2 () String)
(declare-fun output () String)

// map input and output to program variables
(assert (or
    (and (= input1 foo) (= input2 param1))
    (and (= input2 foo) (= input1 param1))))
(assert (= output foo))
```

5.2.3.2 Weakening Example in Yahoo! Pipes

An example of how abstraction impacts the encoding of a `filter` module in Yahoo! Pipes (Figure 5.4) is shown in Figure 5.5. The behavior of the module is to accept items with *tennis* in the description. In the **BCIS** encoding, the string *tennis* is concrete; in the **BCIS** encoding, the string has been replaced with a symbolic string s, so the constraint is satisfied if the solver can identify a value for s. This allows for a weaker encoding of the `filter` module.

One thing to point out with the lattice in Figure 5.3 is that it toggles *all* instances of a data type from symbolic to concrete. There may be stages in-between two levels where some, but not all, of the variables of a given type should be relaxed. Another consideration is that types might not be the only property that should be considered in an abstraction lattice. Other properties, such as bounds on integers or string lengths, could be added to the lattice. Identifying useful abstractions and levels of abstraction requires thorough empirical experimentation, and is part of the future work described in Section 11.2.3.

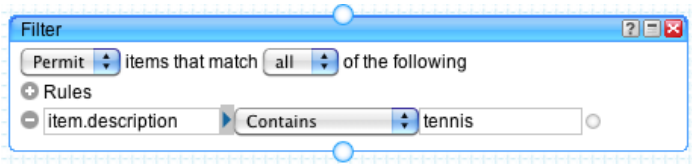


Figure 5.4: Example of Yahoo! Pipes Filter Module

BCIS:
$(\text{recordOf}(in, r) \wedge \text{contains}(\text{field}(r, \text{“}description''), \text{“}tennis'')) \rightarrow \text{recordOf}(out, r)$
BCIS:
$\exists s \mid (\text{recordOf}(in, r) \wedge \text{contains}(\text{field}(r, \text{“}description''), s)) \rightarrow \text{recordOf}(out, r)$

Figure 5.5: Lattice Encoding Example of Yahoo! Pipes Filter Module using Figure 5.3

5.3 Solving

The constraint repository, $RepP_{enc}$, is used by the solver, in conjunction with the encoded specifications, C_{LS}, to determine matches (*SMT Solver* in Figure 5.1). Given C_{LS}, for each $C_P \in RepP_{enc}$, the approach invokes $Solve(C_q \wedge C_{LS}) = (sat, unsat, unknown)$ for every $C_q \in C_P$ such that $TS_q \supseteq TS_{LS}$. The final condition ensures that the type signature of q contains as many elements as the type signature for LS. A more constrained search requires equality, whereas a more relaxed search allows for the superset relationship (the result of a superset relationship is that some input variables in q remain symbolic and the solver, then, assigns values).

Solve returns *sat* when a satisfiable model is found or *unsat* when no model is possible. When the solver is stopped before it reaches a conclusion or it cannot handle a set of constraints, *unknown* might be returned. The set of matches, or *SatP*, consists of all programs P for which at least one path $q \in Q_P$ returns *sat*.

In practice, to invoke the SMT solver for a given specification and encoded path, some additional information is needed, which we call *search parameters*. The first

parameter is the abstraction level of the encoded programs, which is set using the *Abstraction Selector*, shown in Figure 5.1. The selector starts with the strictest (most concrete) level, but this may be relaxed as the search process iterates in the presence of tight or complex constraints, as described in Section 5.2.3. The second parameter is the solver time, which defines how long the solver is allowed to run on a particular constraint system. In some cases, as shown in Section 8.1.4, it can take several minutes for the solver to return *sat* or *unsat*, so setting a maximum solver time can lead to an efficient search, though it can miss some matches.

5.3.1 Matching with Multiple I/O Pairs

When a specification contains multiple input/output pairs (i.e., $k > 1$), each of the pairs is checked independently.[3] That is, $SatP$ is computed for every $ls \in LS$. Strong matches are those that appear in the intersection for all specifications, specifically, $\bigcap_{ls \in LS} SatP_{ls}$. Similarly, those paths that match more ls are also strong matches.

Some programs will *over-approximate* the behavior of the specifications when the following holds: $\exists q \in Q_P \mid \forall ls \in LS \wedge Solve(C_q \wedge C_{ls}) = unsat$. That is, there exists a path in P that LS does not cover. As an example, consider the following LS:

```
LS = {({"cowxboy"}, {"cow"}), ({"abcx"}, {"abc"})}
```

The program in Listing 5.7 over-approximates this behavior since the specification exercises only one of the two paths. Thus, there is behavior, namely path q_2 in Listing 5.11, that returns *unsat* for both ls_1 and ls_2 above. Since q_2 does not contribute to the satisfiability of the specification, it is possibly irrelevant.

[3]Refining the approach to consider the input/output pairs simultaneously is left for future work and discussed in more detail in Section 11.2.6

At the same time, since both ls_1 and ls_2 return *sat* for q_1, then the specification may contain some redundancy. Only part of the specification is necessary to identify P as a result.

Other programs will *under-approximate* the behavior of the specifications when the following holds: $\exists ls \in LS \mid \forall q \in Q_p \wedge Solve(C_q \wedge ls) = unsat$. That is, there exists an $ls \in LS$ that does not have a matching path in P. As an example, consider the following LS:

```
LS = {({"cowxboy"}, {"cow"}), ({"abcx"}, {"abc"}), ({"foo"}, {"bar"})}
```

Considering again the program in Listing 5.7, both paths q_1 and q_2 return *unsat* for ls_3, so P under-approximates the desired behavior and only partially satisfies the specification.

We hypothesize that the number of specifications or paths matched could be considered in the ranking of the results (Section 5.3.2), and explore this notion in Section 7.2.5.3. Beside ranking, the unmatched specifications could be used to guide modifications to a program. One approach to this leverages counter-example guided inductive synthesis [40] in which the concrete input/output pairs that are *unmatched* could be used to guide modification to the program P. This discussion is left for future work in Chapter 11.2.2. For unmatched paths, test-case generation techniques could generate input/output pairs that would match those paths, and ask the user if that behavior is consistent with the code they desire.

5.3.2 Ranking

For the search in Java, the results are returned to the programmer ordered according to various criteria. We discuss three directions in this section, yet their detailed analysis, implementation, and study are left for future work.

5.3.2.1 Path Matching

For each LS and each program P, there are a variety of levels of matching by considering subsets of LS and subsets of the paths in P. The strength of a match is measured as a function of how well the program P matches a provided LS.

For a program P, Q_P is the set of paths in P. For Q_P and a specification LS, let Q_{sat} be the set of paths q such that $\exists ls \in LS \mid Solve(C_{ls} \wedge C_q) = sat$. That is, Q_{sat} is the set of paths in P such that at least one $ls \in LS$ returns sat. Further, let LS_{sat} be the set of ls such that $\exists q \in Q_P \mid Solve(C_{ls} \wedge C_q) = sat$. That is, LS_{sat} is the set of input/output pairs such that at least one path $q \in Q_P$ returns sat.

When $LS = LS_{sat}$, all $ls \in LS$ have a match in P, and this program should be highly ranked. When $Q_P = Q_{sat}$, all paths in P have been covered by LS, and perhaps these programs should be highly ranked. Table 5.1 presents a matrix with potential rankings considering the relationships among these sets. For example, the highest rank could be assigned to a program when both of the aforementioned conditions hold. If either LS_{sat} or Q_{sat} is empty, this means either no paths in P match, or no $ls \in LS$ match, so P is not a result. Handling *middle rank* programs requires further study (Section 7.2.5.3). When $Q_{sat} = Q_P$ and $LS_{sat} \subset LS$, then all paths in P are matched, but only part of LS contributed to the match. This could describe a situation when P has one path, but $|LS| \geq 2$, meaning that at most half of LS is satisfied by P, and in that case P under-approximates the specification. When $LS_{sat} = LS$ and $Q_{sat} \subset Q_P$, this means all of LS was matched by some path(s) in P, but not all of P was matched. That is, P over-approximates the behavior described by LS, which may or may not be reasonable. This would be an opportunity to then generate additional I/O pairs that exercise the unmatched paths in P and ask the programmer if that behavior matches their intention (see Section 5.3.1). Again, further study is needed.

Table 5.1: Path Matching Ranking

	$\|LS_{sat}\| = \varnothing$	$LS_{sat} \subset LS$	$LS_{sat} = LS$
$\|Q_{sat}\| = \varnothing$	–	–	–
$Q_{sat} \subset Q_P$	–	Lowest Rank	Middle Rank
$Q_{sat} = Q_P$	–	Middle Rank	Highest Rank

Within each of the quadrants, there is another consideration that relates to how saturated a program is by LS. If $|LS_{sat}| > |Q_{sat}|$, then at least one of the matched paths is covered by two or more $ls \in LS_{sat}$. On the other hand, if $|LS_{sat}| \leq |Q_{sat}|$, then every ls exercises a different behavior in P. Understanding how to consider this information in a ranking algorithm is left for future study.

5.3.2.2 Concrete Variable Density

Another direction is to consider the density of concrete variables in the program (calculated by the percentage of variables that are concrete), as these are more likely to fit the programmer's query *as is* and without modification (such as instantiating symbolic variables with values from the satisfiable model).

As an example, consider the LS in Listing 5.17, which provides an input/output pair with an integer input and a boolean output. Two potential solutions are shown in Listing 5.18 and Listing 5.19.

Listing 5.17: LS for Concrete Variable Density Ranking

```
LS = {({3}, {true})}
```

Assigning LS to the first solution in Listing 5.18 means all variables are assigned since $a \mapsto input$. The density of concrete variables in this first solution is $1/1 = 1.00$.

Listing 5.18: Solution #1 for Listing 5.17

```
boolean op_ifle(int a) {
    return a > 0;
}
```

On the other hand, there are three input variables in the second solution in Listing 5.19. Assigning LS leaves two free so the solver assigns values depending on the desired output from LS (e.g., for ls_1, if $low \mapsto input$, then a satisfying assignment of the others could be $x \mapsto 4$ and $pup \mapsto 5$). The density of concrete variables in this solution is $1/3 = 0.33$. With this ranking algorithm, the solution in Listing 5.18 would rank above the solution in Listing 5.19.

Listing 5.19: Solution #2 for Listing 5.17

```
boolean isBetween(int x, int low, int pup) {
        return low <= x && x <= pup;
    }
```

In the event of a tie, however, this ranking approach falls short. Consider the solution in Listing 5.20, where the variables could be mapped as follows: $col \mapsto input$, $PROFILE\_DATA\_COLUMN \mapsto 3$, and $row \mapsto 0$.

Listing 5.20: Solution #3 for Listing 5.17

```
int PROFILE_DATA_COLUMN;

boolean isCellEditable(int row, int col) {
           return col == PROFILE_DATA_COLUMN;
}
```

Here, the density of concrete variables is also $1/3 = 0.33$, yet, unlike Listing 5.19, one of the variables is never used (i.e., row). Another criterion must be used to break the tie. One criterion could be the number of reads and writes on the variables. For example, instantiating a symbolic variable that is used only once seems to require less change to the program intention than to instantiate a symbolic variable that is read

five times, so our initial response would be to rank the former program higher. Another approach could use program paths (e.g., Listing 5.18 has one path but Listing 5.19 has two paths). Understanding the best ranking requires thorough investigation and user input, and is left for future work.

5.3.2.3 User Context

A third ranking approach would exploit the user context, and is likely more appropriately applied within a constrained environment like a company repository. Considering user context could, for example, rank higher those snippets of code that have been written by the programmer who specified LS, or that come from a file recently touched by that same programmer. In search results, user context can be a powerful aid since programmers are likely most familiar with code they have written or worked with in the past in the past. As with the other ranking algorithms, further study is needed here.

5.3.3 Ambiguity

Under the hood, the quantifier instantiation heuristics in most SMT solvers are sound but may be incomplete [13]. Most SMT solvers, and specifically the Z3 [79] solver used in our implementation, use the Model-Based Quantifier Instantiation (MBQI) technique for building models for satisfying queries with quantifiers, which most of Java encodings make use of (see Section 6.3). The MBQI creates a candidate interpretation of the quantifier formula based on an interpretation that satisfies the quantifier free assertions, but sometimes the candidate interpretation cannot be found. The result is that *unknown* may be returned when the logic is actually satisfiable. Treating a response of *unknown* as unsatisfiable means our search is also incomplete, but treating

those responses as results means our search is potentially unsound. Understanding what should be done with the *unknown* results requires further investigation and is left for future work.

5.4 Summary

At this point, we have presented a general definition of the approach including all phases of the search process, specifying a query, indexing source code, and finding matches. Each phase has been illustrated using Java examples for clarity.

Still left to be studied is *when* to apply abstractions during the search process and how to *rank* matches. Potential solutions to these questions will need to be thoroughly evaluated empirically, and that is left for future work. In the next chapter, we describe in greater detail how this approach has been instantiated in two of the targeted languages, Java and Yahoo! Pipes.

Chapter 6

Implementation

In this chapter, we introduce Satsy, an input/output search engine that has implemented our approach to semantic search (Chapter 5). Satsy supports two languages, Java and Yahoo! Pipes, for the purpose of finding programs given an encoded repository and a specification.[1] The encoding tools transform programs into constraints in the SMT-LIB2 [61] format, are different for each language supported, and are described separately from Satsy. Constraints in SMT-LIB2 format are stored as text and are consumed by the Z3 SMT solver v.4.1 [79]. The implementation utilizes the UFNIA: *Non-linear integer arithmetic with uninterpreted sort and function symbols* theory in the encoding of all programs.

An overview of how Satsy works is presented next, followed by the data types supported in our implementation and the implementation details for Java in Section 6.3 and for Yahoo! Pipes in Section 6.4.

[1]The SQL implementation is not within the scope of this thesis, but is included in this motivation and illustration because it has been part of previous work [70].

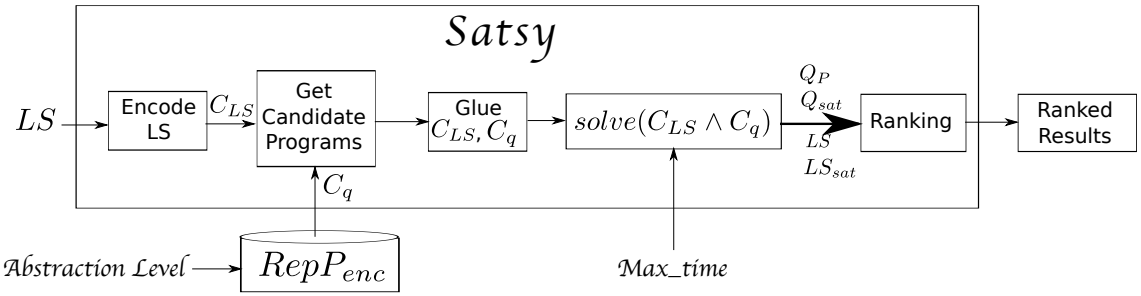


Figure 6.1: Satsy Solving Overview

6.1 Solving with Satsy

Satsy is the tool that takes as input a lightweight specification and a repository of constraints, and returns a set of ranked results as output. Satsy implements the online portion of the approach in Figure 5.1, and is shown in Figure 6.1. Given an LS from a programmer, Satsy first encodes the specification (Section 5.1.3), forming C_{LS}. Next, a set of candidate paths is gathered from a repository of encoded programs, $RepP_{enc}$, subject to a specified abstraction level of encodings. The set of candidate paths represents those paths for which the type signature of the path are subsumed the type signature of the specification, $TS_q \supseteq TS_{LS}$. Next, for each path q, and for each $ls \in LS$, the encoded specification is "glued" to C_q using additional constraints that map TS_{ls} onto TS_q (Section 5.2.2). Next, the solver is invoked for each $ls \in LS$, and each candidate path q, $solve(C_{LS} \wedge C_q)$, with a specified maximum solver time. For each program (where paths are organized back into their original program), four pieces of information are returned for consideration in the ranking algorithm: encoded paths Q_P, matched paths Q_{sat}, specification LS, and matched specification LS_{sat} (see Section 7.2.5.3 for details).

Building the repository of encoded programs, $RepP_{enc}$, is a process that is highly dependent on the language and available tools. The Java implementation is described in Section 6.3, followed by the Yahoo! Pipes implementation in Section 6.4.

6.2 Data Types

Encoding the language subsets for Yahoo! Pipes and Java requires similar data types. Three primitive types are supported (characters, integers, booleans) and one composite type, list. These basic types are sufficient to represent all the constructs the encoding supports across these domains. For example, a string is a shorthand for a list of characters and a Yahoo! Pipes record is a map of strings to objects with names modeled as strings.

Booleans and integers are built-in types for the UFNIA theory, but characters (`Char`), lists (`List`), records (`Record`), and strings (`String`) are declared as uninterpreted sorts. As strings are used by both targeted languages, the string implementation is described here (lists and records are discussed in Section 6.4).

Each character supported in the encoding is declared as a distinct function with type `Char`. A string is defined by its length and by explicitly defining every ith character of the string for $0 \leq i < length$. An axiom determines equality between strings when both have the same length and contain the same characters.

Listing 6.1 shows how strings are encoded in SMT_LIB2 format (lines that start with the ';' character are comments). The declarations for the sorts are on lines 2 and 3. The character alphabet in this example is set to `a, b, c` on lines 6 - 9, though in practice it contains the entire roman alphabet and symbols found on a standard English keyboard, as shown in the grammar in Figure 6.3. The functions required for string definition are on lines 12 and 13, and the axiom for string equality is on lines

16 - 22. It reads, for all strings `s1` and `s2`, if they have the same length *and* if for all integers i where $0 \leq i < length(s1)$ the strings have matching characters at index i, then this implies their equality (line 22).

Listing 6.1: String Encoding Constraints

```
1  ; custom sorts for string and char definition
2  (declare-sort Char 0)
3  (declare-sort String 0)
4
5  ; functions for all Chars
6  (declare-fun _a_ () Char)
7  (declare-fun _b_ () Char)
8  (declare-fun _c_ () Char)
9  (assert (distinct _a_ _b_ _c_))
10
11 ; functions for string definition
12 (declare-fun length ( String ) Int )
13 (declare-fun charAt (String Int) Char)
14
15 ;axiom for equality
16 (assert (forall ((s1 String) (s2 String))
17               (=> (and
18                    (= (length s1) (length s2))
19                    (forall ((i Int))
20                           (=> (and (<= 0 i) (< i (length s1)))
21                               (= (charAt s1 i) (charAt s2 i)))))
22                   (= s1 s2))))
```

An alternative choice to the implementation in Listing 6.1 would have been to use the theory of arrays, which is supported by many SMT solvers (e.g., Z3 [79]). The decision to use uninterpreted sorts and axioms instead was based on performance; at the time of implementation, we were using Z3 as the solver in Satsy, and array theory in Z3 was slower than the alternative.

To illustrate the string implementation, consider the assignment statement, *String* s = "abc"; The implementation of string s is shown in Listing 6.2. The string is defined on line 2, after which lines 3 - 5 set the value of s and line 6 asserts the length.

Listing 6.2: A String as Constraints

```
1 ; String s = "abc";
2 (declare-fun s () String)
3 (assert (= (charAt s 0) _a_))
4 (assert (= (charAt s 1) _b_))
5 (assert (= (charAt s 2) _c_))
6 (assert (= (length s) 3))
```

To efficiently support operations on strings, the lengths are bounded where the bound is configurable (in line with recent work on solving string constraints [5][35]). Moreover, this is reasonable given that humans will create input/output pairs that will tend to be small. The same length restrictions are placed on list lengths. For four of the supported datatypes, integers, characters, booleans, and strings, our encoding supports concrete or symbolic values.

For integers and booleans, as these are built-in types, removing their values, as was shown in Section 5.2.3.1, is sufficient for encoding symbolic variables with these types. For characters, however, an additional axiom is required to restrict the solver to select from only the characters defined in the implementation when instantiating a symbolic character within the solver. Listing 6.3 shows an axiom for the alphabet presented in Listing 6.1.

Listing 6.3: Symbolic Character Encoding

```
1 (assert (forall ((c char))
2     (or
3         (= c _a_)
4         (= c _b_)
5         (= c _c_))))
```

For strings, declaring a symbolic string simply requires a constraint that the length of the string be greater than or equal to zero. For example, weakening the string s in Listing 6.2 would look like the encoding in Listing 6.4, where the string is declared on line 2 and the length constraint is added on line 3.

Listing 6.4: A SymbolicString as Constraints

```
1 ; String s;
2 (declare-fun s () String)
3 (assert (>= (length s) 0))
```

With the current implementation, there is an opportunity for the solver to assign characters at indices in a string that are outside of the bounds of the string length. That is, a string of length 3 could have a value at index 5. Presently, equality between strings is defined by the values of the strings within the bounds of zero and the string length. Further, for any operation on a string the string bounds are checked so no operation occurs outside of the bounds of the string. Thus, this should have a limited impact, if any, on the results of the study. This threat to internal validity is discussed further in Section 9.1.2.

6.3 Java Implementation Detail

This section describes how the encoding section of the approach (Section 5.2) has been implemented in the Java language. This process is depicted in Figure 6.2. For each program in a repository that matches our supported language, the first step (1) is to check if there are multiple paths. If there are, the program is sent to JPF, where the paths are traversed and logged. All paths, either from the original single-path program or from JPF, are encoded as constraints (3), which are stored in a repository (4). Each of these steps is now described in detail.

6.3.1 Language Support

All programs $P \in RepP$ (Figure 6.2) contain only constructs that are supported by our implementation. This includes a subset of the Java language covering predicates,

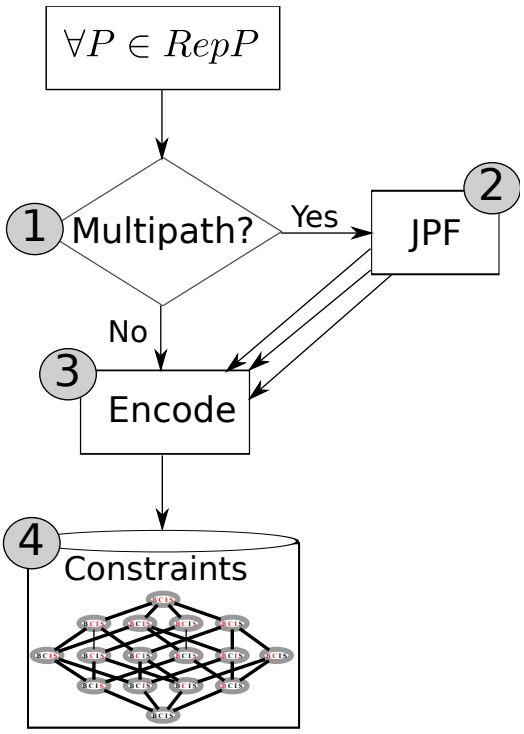


Figure 6.2: Java Encoding Overview

assignment statements, and return statements, subject to the grammar shown in Figure 6.3. We support arithmetic operations and most of the `java.lang.String` API, including the following string manipulation functions: `charAt`, `concat`, `contains`, `endsWith`, `equals`, `equalsIgnoreCase`, `indexOf`, `isEmpty`, `lastIndexOf`, `length`, `replace`, `startsWith`, `substring`, `toUpperCase`, `toLowerCase`, and `trim`. This API was selected based on its popularity, the rich set of behaviors that can be achieved using combinations of the methods, and the potential to extend some of the principles to other APIs, such as list and array processing (see future work in Section 11.2.4).

The grammar in Figure 6.3 was implemented in ANTLR. It is used to recognize when we can support a whole method or part of a method, and in the transformation process. The scope of supported Java *snippets* covers methods and blocks that fit the supported grammar. Some snippets have a single path while others have multiple

Figure 6.3: Java Supported Grammar (intra-procedural)

⟨*statements*⟩ ::= (⟨*statement*⟩)*

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

⟨*variableDeclaration*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ ‘`;`’

⟨*ifStatement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘`{`’ ⟨*statements*⟩ ‘`}`’
(‘`else`’ ‘`if`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘`{`’ ⟨*statements*⟩ ‘`}`’)*
(‘`else`’ ‘`{`’ ⟨*statements*⟩ ‘`}`’)?

⟨*assignment*⟩ ::= ⟨*type*⟩? ⟨*ident*⟩ ‘`=`’ ⟨*expr*⟩ ‘`;`’

⟨*return*⟩ ::= ‘`return`’ ⟨*expr*⟩ ‘`;`’

⟨*expr*⟩ ::= ⟨*logicalOrExpression*⟩

⟨*logicalOrExpression*⟩ ::= ⟨*logicalAndExpression*⟩ (‘`||`’ ⟨*logicalAndExpression*⟩)*

⟨*logicalAndExpression*⟩ ::= ⟨*equalityExpression*⟩ (‘`&&`’ ⟨*equalityExpression*⟩)?

⟨*equalityExpression*⟩ ::= ⟨*relationalExpression*⟩ ((‘`!=`’ | ‘`==`’) ⟨*relationalExpression*⟩)?

⟨*relationalExpression*⟩ ::= ⟨*addExpression*⟩ ((‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’) ⟨*addExpression*⟩)?

⟨*addExpression*⟩ ::= ⟨*multExpression*⟩ ((‘`+`’ | ‘`-`’) ⟨*multExpression*⟩)?

⟨*multExpression*⟩ ::= ⟨*primaryExpression*⟩ ((‘`*`’ | ‘`/`’ | ‘`%`’) ⟨*primaryExpression*⟩)?

⟨*primaryExpression*⟩ ::= ⟨*invokeExpression*⟩
| ⟨*stringLiteral*⟩
| ⟨*charLiteral*⟩
| ⟨*integer*⟩
| ⟨*booleanLiteral*⟩
| ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ⟨*ident*⟩

⟨*invokeExpression*⟩ ::= (⟨*ident*⟩ | ⟨*stringLiteral*⟩) ((‘`.`’ ⟨*noparam*⟩ ‘`(`’ ‘`)`’)
| (⟨*oneparam*⟩ ‘`(`’ ⟨*expr*⟩ ‘`)`’)
| (⟨*twoparam*⟩ ‘`(`’ ⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)? ‘`)`’))+

⟨*noparam*⟩ ::= ‘`length`’ | ‘`toLowerCase`’ | ‘`toUpperCase`’ | ‘`trim`’

⟨*oneparam*⟩ ::= ‘`charAt`’ | ‘`concat`’ | ‘`contains`’ | ‘`endsWith`’ | ‘`equals`’ | ‘`equalsIgnoreCase`’ | ‘`indexOf`’ | ‘`lastIndexOf`’ | ‘`startsWith`’ | ‘`substring`’

⟨*twoparam*⟩ ::= ‘`indexOf`’ | ‘`lastIndexOf`’ | ‘`replace`’

⟨*type*⟩ ::= ‘`char`’ | ‘`String`’ | ‘`int`’ | ‘`boolean`’

⟨*booleanLiteral*⟩ ::= ‘`true`’ | ‘`false`’

⟨*integer*⟩ ::= (‘`0`’ | (‘`1`’..‘`9`’) (‘`0`’..‘`9`’)*);

⟨*stringLiteral*⟩ ::= ‘`"`’ (LETTER | DIGIT | ‘`:`’ | ‘`/`’ | ‘`_`’ | ‘`,`’ | ‘`.`’ | ‘`\`’ | ‘`\"`’ | ‘`'`’ | ‘`-`’ | ‘ ’ | ‘`(`’ | ‘`)`’ | ‘`;`’ | ‘`{`’ | ‘`?`’ | ‘`}`’ | ‘`=`’ | ‘`@`’ | ‘`[`’ | ‘`]`’ | ‘`&`’ | ‘`!`’ | ‘`#`’ | ‘`%`’ | ‘`+`’ | ‘`*`’ | ‘`^`’ | ‘`~`’ | ‘`‘`’ | ‘`<`’ | ‘`>`’ | ‘`|`’)* ‘`"`’

⟨*charLiteral*⟩ ::= ‘`‘`’ (⟨*escape*⟩ | ∼(‘`'`’|‘`\`’)) ‘`'`’

⟨*escape*⟩ ::= ‘`\`’ (‘`b`’ | ‘`t`’ | ‘`n`’ | ‘`f`’ | ‘`r`’ | ‘`"`’ | ‘`'`’ | ‘`\`’)

⟨*ident*⟩ ::= (LETTER | ‘`_`’ | ‘`$`’)(LETTER | ‘`_`’ | DIGIT | ‘`$`’)*

paths. Snippets are removed from context and considered independently of their original implementation. That is, if a class has only one method supported by the grammar, we keep that method and throw away the rest. For this reason, the behavior of the code may differ from the behavior in the original implementation (e.g., if a field value is accessed, we declare it symbolically within the scope of the snippet). This threat is discussed in Section 9.1.2.

6.3.2 Program Transformation

The transformation process covers steps (1) - (3) in Figure 6.2. Methods that contain a predicate, conjunction, or disjunction need to be processed as multi-path programs using symbolic execution, as described in Section 5.2.1 ((2) in Figure 6.2). This is done using a listener attached to the symbolic execution extension [34, 33] to the Java PathFinder model checker [75]. The basic process involves two steps. First, a JPF driver is written to invoke the method under evaluation, and second, the listener is attached. The listener traverses the paths in the method, recording path conditions and the executed statements along those paths. Next, each path is encoded as constraints ((3) in Figure 6.2). This is done using an ANTLR grammar and a set of transformation rules that map program constructs onto constraints.

6.3.2.1 JPF Driver Creation

For JPF to be invoked on a method, that method needs to be called from a main method so JPF can execute it symbolically by replacing the concrete inputs with symbolic ones. This process is necessary because the listener (which collects path condition information) needs to be attached to executable Java code in order to traverse the paths.

Since the methods we encode are taken out of context, a driver needs to be created that will invoke each method from a main method. The driver wraps the method in a class with a main method, and call the method from the main method using arbitrary, concrete values as arguments. In this way, a path is formed to the method that JPF can trace. The implementation can create a driver and invoke JPF automatically when given a method, without human intervention.

An example driver for the method in Listing 5.7 is shown in Listing 6.5. The method under evaluation, `bar`, appears at lines 9-15. The main method, on lines 5-7, calls `bar` on line 6 with an arbitrary input string, "abc". When the listener is invoked on *bar*, "abc" is replaced by a symbolic string.

Listing 6.5: JPF Driver for Listing 5.7

```
1  package jpfDrivers;
2
3  public class DynamicDriver1 {
4
5      public static void main(String[] args){
6          bar("abc");
7      }
8
9      static String bar(String foo) {
10         if(foo.contains("x")) {
11             int i = foo.indexOf("x");
12             foo = foo.substring(0, i);
13         }
14         return foo;
15     }
16 }
```

6.3.2.2 JPF Listener

We have implemented a listener in JPF that records path conditions and executed statements for Java methods with bodies that are recognized by the grammar in Figure 6.3.

Within JPF, the symbolic execution of a method is a depth-first traversal of the search space [33]. As a method is symbolically executed, the listener logs the statements that have been executed by processing the bytecodes. When a branching bytecode is reached, JPF makes a decision on whether to take the true or false branch. The listener records this decision and continues aggregating executed statements. When the end of the method is reached (i.e., a return statement), the listener logs this path. Then, the search backtracks to a branching condition, negates it, and the process continues. The end result is a set of paths containing branching conditions and executed statements. These form the set of paths Q_P for a program P.

As an example, the listener was used to break down the multi-path method in Listing 5.7 in Section 5.2.1 into the paths in Listing 5.9 and Listing 5.8 using the driver in Listing 6.5. Embedded within the listener is the logic to transform each path into SSA form. For single-path programs, each program is pre-processed into SSA form outside JPF using an ANTLR grammar (part of (3) in Figure 6.2).

One by-product of using JPF as the symbolic execution engine to collect paths is that it removes from consideration any infeasible paths (i.e., those paths are no longer traversed once a conflict is identified, and so the paths are never recorded by the listener). This is advantageous for the solving phase because paths that will never return true for any input not returned by the listener, and thus are never encoded and stored in our constraint repository.

6.3.2.3 Program Encoding

Encoding happens on the path level. Once the paths have been identified, the transformation step can begin ((3) in Figure 6.2).

Using the grammar in Figure 6.3, each piece of each program path is transformed into constraints using our transformation tool. The non-terminals in the Java grammar

are represented as program objects in the transformation code, and each object has different properties. For example, `<invokeExpression>` contains the receiving object (`<ident>`), the method name, and between zero and two expressions (`<expr>`). Within the transformation code is a transformation table, which acts like a template for each object. Given the object, the properties fill in the holes of the template and the constraints are generated systematically. The collection of all constraints generated for all non-terminals in the Java grammar form the encoded program path.

A tricky part of the encoding is translating method invocations on string objects. The encoding logic for each of the support string library methods is shown in Table A.1, Table A.2, Table A.3, Table A.4, Table A.5, and Table A.6. We provide a sample here in Table 6.1. For each method call, the constraint purpose and definition are presented. When types for the variables are not stated, we used the following scheme: i and j are integers, s and sub are strings, c is a character.

Table 6.1: Sample Java Implementation Details

$s.contains(sub) = true$

ensure size constraints of strings	$(\text{length}(s) \geq \text{length}(sub))$
ensures sub is a subset of s	$(\exists j$ $((j \geq 0)$ $\wedge(j < (\text{length}(s) - \text{length}(sub)))$ $\wedge(\forall i$ $(((i < \text{length}(sub)) \wedge (i \geq 0))$ $\rightarrow (\text{charAt}(s, (i + j)) = \text{charAt}(sub, i))))))$

$s.\,\text{indexOf}(c) = i$

ensures $s[i] = c$	$(s.\,\text{charAt}(i) = c)$ $\vee(i = -1)$
ensures i is the first occurrence of c	$(\forall j$ $(((j \geq 0) \wedge (j < i))$ $\rightarrow (\neg(\text{charAt}(s, j) = c)))$
sets bounds on i	$((i \geq 0) \wedge (i < \text{length}(s)))$ $\vee(i = -1)$

For example, consider the encoding for $s.contains(sub) = true$ in Table 6.1. To encode this as constraints, we consider the length and content of strings s and sub. For length, it is required that the length of the receiving object s be at least the length of the argument sub. This is the first constraint defined in Table 6.1 for the *contains* method invocation. For content, we need to identify an offset in s such that between the bounds of the offset and the offset plus the length of sub, the two strings match. This is the criteria for containment, and is expressed in the second constraint in Table 6.1 where j is the offset.

One oddity in the implementation comes with the encodings for $indexOf$ and $lastIndexOf$ in Table A.2 and Table A.3, with an example in Table 6.1. The index i is set to either be the location of the character or string in the reference string (i.e., s), or -1 in the event that the character or string cannot be found. In the SMTLIB2 conversion, the first and third constraints on the $indexOf$ are joined to ensure no interference (e.g., it would be invalid if $(s.\,\text{indexOf}(i) = c)$ in the first constraint and $(i = -1)$ in the third both evaluate to true, so these are combined to protect against that situation). More details on this follow in Appendix A.1.

The encodings over-constrain the program behavior, and in this way, err on the side of returning *unsat* to avoid false positives. The consequence is that some potential matches may be missed. For example, the alphabet considered for string values is pre-determined. If, for example, a programer provides a specification that requires the solver to use an accented character, such as ó, then *unsat* would be returned since that character is not recognized by our implementation. An unintended benefit is that this actually improves the search time since the solver is able to more quickly return *unsat* by providing a conflict than to return *sat* and provide a satisfiable assignment.

6.3.2.4 Abstraction

One process not described for Java is that of applying abstractions. In the encoding process for source code and with the current implementation, weakening concrete values with symbolic values is not supported. Adding support for weakening in Java would involve a small engineering effort, and is part of our planned future work. For the specifications provided by the programmer, the current implementation supports the *adding*, *removing*, and *replace* functions (Section 5.1.2).

6.3.2.5 Limitations and Nuances

Some restrictions and extensions are made to the grammar in Figure 6.3 with respect to the encodings, and are worth pointing out. These are:

- As shown in Table A.4, calls to $s.replace(c, d) = t$ restrict the types of the arguments to *replace* to characters. The grammar does not impose such a restriction, which means the encoding must reject snippets that contain $CharSequence$ objects as arguments to *replace*.

- In the Java VM, characters are represented as integers, allowing for relational comparisons using the `'<'`, `'<='`, `'>'`, and `'>='` operators. However, in our encoding, each character is defined as a function returning an uninterpreted sort (Section 6.2), meaning that relational operators cannot be encoded. Thus, the encoding restricts relational operators to integers; addressing this limitation is part of the future work.

- The `null` value for all data types is not currently supported. One challenge of this is defining equality among the `null` value across datatypes, and this is left for future work.

- The '+' operator can be overloaded for string concatenation
- As described in Section 5.2.2, the mapping involves all possible bisections of the inputs in LS to the input variables in q. However, we do not keep track of the bindings for each $ls \in LS$ to ensure consistency in binding among all input/output pairs in a specification. This is left for future work.

6.3.3 Constraint Storage

Step (4) in Figure 6.2 involves storing the constraints for each program. As each path in a program is processed separately, the paths for each program are stored separately. In this step, we use a MySQL database with the following structure:

```
mysql> describe snip;
+---------+---------------+------+-----+---------+----------------+
| Field   | Type          | Null | Key | Default | Extra          |
+---------+---------------+------+-----+---------+----------------+
| id      | int(11)       | NO   | PRI | NULL    | auto_increment |
| file    | varchar(1024) | NO   |     | NULL    |                |
| method  | varchar(128)  | NO   |     | NULL    |                |
| source  | text          | NO   |     | NULL    |                |
| path    | text          | NO   |     | NULL    |                |
| srcHash | varchar(16)   | YES  | MUL | NULL    |                |
+---------+---------------+------+-----+---------+----------------+
```

The *id* field is a unique identifier for the record. The *file* and *method* fields provide provenance information for where the path and its parent method came from. The original implementation of the method is stored in *source*, and the *path* field is the SSA representation of the path, right before it goes through the encoding step. The *srcHash* field is a foreign key linking each record to related records in

the other database tables. One table contains type signature information used to quickly identify potential matches and the other contains the actual constraints as text. When different levels of abstraction in the program encodings are supported by the Java implementation table, the table containing the constraints will need to have an additional field indicating the abstraction level of that particular encoding. Using this information is part of the Yahoo! Pipes implementation, which considers four levels of abstraction in the encoding (Section 6.4.3).

We chose a MySQL database for storage out of convenience, though in practice, the efficiency of retrieving the constraint representation of potential matches to provide to Satsy (Figure 6.1) is very important, so this may not be the best option long-term.

6.4 Yahoo! Pipes Implementation Detail

This section describes how the encoding section of the approach (Section 5.2) has been implemented in the Yahoo! Pipes language. This process is depicted in Figure 6.4. For all programs in a repository that match our supported language, the first step is to (1) refactor the program into a more simplified version to ease some of the burden on the solver. Next, the refactored program is encoded into constraints, which (3) are stored in a repository. Each of these steps is described here in detail.

6.4.1 Language Support

All programs $P \in RepP$ (Figure 6.4) contain only constructs that are supported by our implementation. This includes a subset of the Yahoo! Pipes mashup language covering the following modules: `fetch, filter, output, sort, split, tail, truncate`, and `union`, covering six of the top 10 most commonly used constructs.

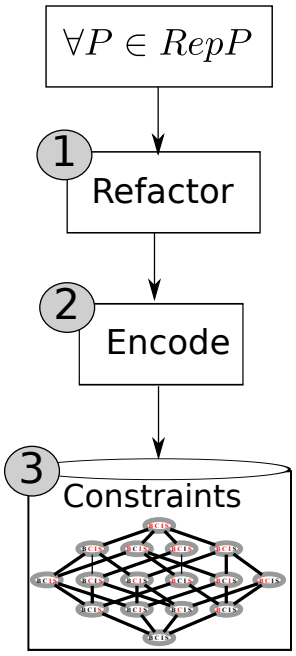


Figure 6.4: Yahoo! Pipes Encoding Overview

We formally describe the supported language fragment in the grammar in Figure 6.5. To map a pipe onto the grammar, we transform each program into a parallel-serial graph, as defined in our previous work [71]. To illustrate, the program in Figure 4.3(b) would be represented as: `output tail truncate fetch`, and the program in Figure 4.3(a) would be represented as: `output filter union (fetch) (fetch)`.

This language fragment covers common list manipulation operators, but our implementation does not cover constructs like `rename`, through which items within a list are modified. Future work includes extending this language support to cover constructs that manipulate the list items themselves.

Figure 6.5: Yahoo! Pipes Supported Grammar

⟨*pipe*⟩ ::= '`output`' ⟨*composition*⟩

⟨*composition*⟩ ::= ⟨*operator*⟩? ⟨*grouping*⟩
| ⟨*init*⟩

⟨*grouping*⟩ ::= '`union`' ⟨*segment*⟩+

⟨*segment*⟩ ::= '`(`' ⟨*init*⟩ '`)`'
| '`(`' ⟨*operator*⟩? ⟨*grouping*⟩ '`)`'
| '`(`' ('`(`' ⟨*operator*⟩ '`)`')* '`)`' '`split`' ⟨*composition*⟩

⟨*init*⟩ ::= ⟨*interior*⟩? '`fetch`'

⟨*operator*⟩ ::= ('`filter`' | '`sort`' | '`truncate`' | '`tail`')+

6.4.2 Program Transformation

The transformation process covers steps (1) and (2) in Figure 6.4. Since there are no predicates in the Yahoo! Pipes language, every program has only one program path. For this reason, symbolic execution is not necessary.

6.4.2.1 List Implementation

Lists in Yahoo! Pipes are implemented similarly to strings (Listing 6.1), with a list being analogous to a string and a record being analogous to a character. Lists are defined by their size (i.e., length) and by defining the records at each index i in the list. Listing 6.6 describes the List implementation for Yahoo! Pipes. Comparing this to the string implementation, the main differences come in the custom sort definitions on lines 2 and 3 and in the list definition function names, which are on lines 12 and 13. The function $recordOf$ for lists behaves like $charAt$ for strings, and $size$ for lists behaves like $length$ for strings.

Listing 6.6: A Yahoo! Pipes list as Constraints

```
; custom sorts for list and record definition
(declare-sort Record 0)
(declare-sort List 0)

; functions for all Records
(declare-fun r1 () Record)
(declare-fun r2 () Record)
(declare-fun r3 () Record)
(assert (distinct r1 r2 r3))

; functions for list definition
(declare-fun size ( List ) Int )
(declare-fun recordOf (List Int) Record)

;axiom for equality
(assert (forall ((l1 List) (l2 List))
               (=> (and
                    (= (size l1) (size l2))
                    (forall ((j Int))
                           (=> (and (<= 0 j) (< j (size l1)))
                               (= (recordOf l1 j) (recordOf l2 j)))))
                   (= l1 l2))))

; List in = [r1, r2, r3];
(declare-fun in () List)
(assert (= (recordOf in 0) r1))
(assert (= (recordOf in 1) r2))
(assert (= (recordOf in 2) r3))
(assert (= (size in) 3))
```

Listing 6.7: Yahoo! Pipes Record Implementation

```
; uninterpreted functions for list and record definition
(declare-sort Record 0)
(declare-sort String 0)

; getter functions for the records
(declare-fun getTitle (Record) String)
(declare-fun getDescr (Record) String)
(declare-fun getAuthor (Record) String)
(declare-fun getLink (Record) String)
(declare-fun getPubdate (Record) Int)
```

A list is composed of records, and as shown in Figure 2.2, a record is a map of key-value pairs. In the constraint representation, each of the keys requires a getter function used to set the values for each key. The record data structure, as implemented in SMT-LIB2 format, is shown in Listing 6.7. The getter functions are on lines 6-10 for each key in the map.

6.4.2.2 Program Encoding

In the transformation process, we take advantage of a previous infrastructure [68] to refactor the Pipes programs to a more simple format ((1) in Figure 6.4). These refactorings focus on decreasing the size of the pipes and standardizing them according to the community standards [68]. This is not a necessary step for the encoding process, though it may lead to modest performance gains. The biggest advantage, though we have not evaluated it fully, is that it relieves some burden Satsy when solving for Pipes programs (Figure 6.1) since the sizes of the programs (and thus the number of constraints) are smaller.

In a Pipes program, the encoder ((2) in Figure 6.4), maps each module and wire onto constraints using inclusion, exclusion, order, and size properties. *Inclusion* constraints ensure completeness; all relevant records from the input exist in the output. *Exclusion* constraints ensure precision; all records in the output are relevant. *Order* constraints ensure that the records are ordered properly in the list. *Size* constraints ensure that the lengths of the input and output lists for each module make sense (e.g., the lengths are equal for a *sort* module, but $size(out) \leq size(in)$ for a $filter$ module). The *equality* constraints ensure that the output of the source module is equivalent to the input of the destination module.

The specifics of the module encodings are shown in Table A.7, Table A.8, and Table A.9 for the supported constructs, and a sample is provided in this section in

Table 6.2. For each module, the constraint type (inclusion, exclusion, order, size) is shown followed by the constraint definition. When types for the variables are not stated, we used the following scheme: i and j are integers, s is a string, r, r_1, and r_2 are records, *in* and *out* are lists.

Table 6.2: Sample Yahoo! Pipes Encoding

filter(in, title, equals, s) = out

inclusion	$(\forall i($ $((i \geq 0) \wedge (i < size(in)))$ $((getTitle(recordOf(in, i)) = s)$ $\rightarrow hasRec(out, r))$ $\wedge((\neg(getTitle(recordOf(in, i)) = s))$ $\rightarrow \neg(hasRec(out, r)))))$
exclusion	$(\forall r$ $(hasRec(out, r) \rightarrow hasRec(in, r)))$
order	$(\forall r_1, r_2$ $(hasRec(out, r_1) \wedge hasRec(out, r_2)$ $\wedge(\exists i, j$ $((i < j)$ $\wedge(recordOf(out, i) = r_1) \wedge (recordOf(out, j) = r_2))))$ $\rightarrow (\exists k, l$ $((k < l)$ $\wedge(recordOf(in, k) = r_1) \wedge (recordOf(in, l) = r_2))))$
size	$(size(in) \geq size(out))$

To illustrate using the `filter` module, the inclusion constraint ensures that all records in the input with some property are in the output. In Table 6.2, that property is that the *title* of every record is *equal* to some string s. The exclusion constraint ensures that all records in the output come from the input (i.e., no new records were introduced). The order constraint makes sure the ordering of the items is preserved from input to output, and the length constraint ensures that the output list is the same size or shorter than the input list.

There is an additional restriction to the grammar in Figure 6.5. Due to the implementation of strings in the encoding, the *sort* module can be encoded only when the sort operation is performed on the publication date of the items. This is similar to the restriction in the Java encoding where relational operators are valid only on integers.

6.4.2.3 Abstraction

In the Yahoo! Pipes language, we support abstraction on datatypes as described in Section 5.2.3, which is used by Satsy (Figure 6.1). Specifically, we support four levels based on the supported datatypes, integers and strings. These levels are shown in the lattice in Figure 6.6, which is a subset of the lattice presented in Figure 5.3.

During encoding weakening (Section 5.2.3.2), abstraction can be applied for strings and integers within the *filter* module, and on integers within the *truncate* module. That is, in the *filter* module encoding shown in Table A.7, a symbolic representation would relax the value of s, allowing the solver to identify a value for s to match the input/output specification. For the truncate module encoding shown in Table A.8, a symbolic encoding would relax the value of n, asserting that $n \geq 0$.

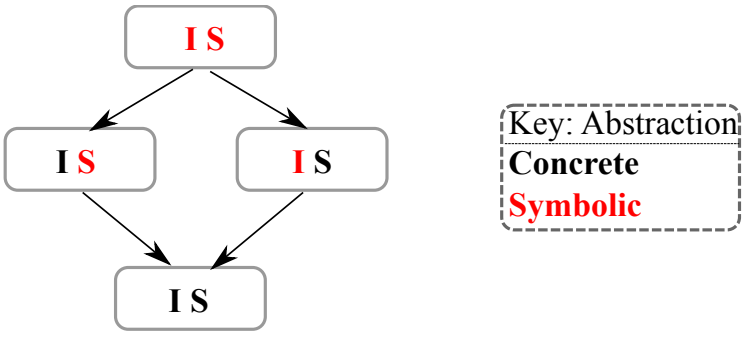


Figure 6.6: Type Lattice for Yahoo! Pipes

6.4.3 Constraint Storage

Step (3) in Figure 6.4 involves storing the constraints for each program. In this implementation, we support four levels of abstraction. Thus, for each program, we have four different sets of constraints representing the abstraction levels (corresponding to the levels of the lattice in Figure 6.6).

Unlike the Java implementation, all inputs and outputs in the Yahoo! Pipes language have the same types, so type matching is not a necessary step in the process. However, the input can contain multiple URLs and thus multiple lists, so matching input lists from LS to the number of URLs in a program is one of the criteria for a program to be a candidate match in Satsy (Figure 6.1). Thus, as part of the constraint storage, the number of potential inputs, corresponding to the number of *Fetch Feed* modules, is recorded.

While the Java implementation uses a database storage structure, for Yahoo! Pipes, the storage is in flat files on a server. Abstraction levels are identified by the file extension, and the number of potential inputs is indicates as a comment at the top of the each file.

6.5 Summary

In this section, we have presented the implementation details for Satsy, the semantic search engine, and two languages, Java, and Yahoo! Pipes. By now, it has probably become clear that the adaptation effort of this approach to a new domain is quite high and requires experts who are intimately familiar with a targeted domains' semantics and SMT theories. This overhead could be mitigated, in part, by reusing some of the transformations that are common among libraries and languages (e.g., string manipulation between Yahoo! Pipes and Java), or by approaches that automatically

extract invariants from program code. In Chapter 10, we review the state-of-the-art research in this area.

Chapter 7

Java Evaluation

In the evaluation of this approach, we have designed a series of studies to assess and highlight some key aspects of the approach through the two implemented domains: Java and Yahoo! Pipes. This chapter focuses on the Java implementation in Satsy, and we aim to address one general research question, which is broken into three sub-questions:

RQ2: How effective is Satsy at returning relevant source code?

- **RQ2(a)** How do search results from Satsy compare to those found using Google when searching over the same repository of programs?
- **RQ2(b)** How much can existing syntactic search approaches be improved by augmenting results with Satsy?
- **RQ2(c)** How do search results from Satsy compare to those found using Google and those found with the code-specific search engine, Merobase, from the perspective of a programmer?

Toward *RQ2*[1] in Java, we conduct two separate studies. The first study, presented in Section 7.1, addresses *RQ2(a)* and *RQ2(b)*. Effectiveness is measured using the semantics of the first 10 source code snippets returned by a search, computing P@10 based on behavior. That is, we analyze each source code snippet and verify that it behaves as specified by an input/output example. In the second Java study, which addresses *RQ2(c)* and is presented in Section 7.2, we measure effectiveness by asking human participants if source code returned by a search is relevant to a programming task, computing P@10 based on their feedback.

Comparing our search to state-of-the-art and state-of-the-practice searches, as is the goal of *RQ2(a)* and *RQ2(c)*, is tricky since the query models are heterogeneous across approaches (e.g., keyword, formal specification, input/output example, etc.) and the content of the repositories may vary significantly. To evaluate the effectiveness of the search in Java, we performed a staged assessment with these studies. For each study, we require four parts to assess search effectiveness: a repository, queries to issue, search engines, and oracles to judge correctness of the search results. Table 7.1 presents a broad overview of these parts for each of the Java assessments, and their associated research questions.

Table 7.1: Java Evaluations for *RQ2*

RQ	**Repository**	**Queries**	**Searches**	**Oracle**
RQ2(a)	Koders	Stackoverflow	Constrained Google, Satsy	Stackoverflow
RQ2(b)	Web	Stackoverflow	Google, Satsy	Analysis
RQ2(c)	GitHub	Humans	Google, Merobase, Satsy	Humans

In the first assessment toward *RQ2(a)* (Section 7.1.1), the *repository* was gathered by scraping `koders.com`. The *queries* issued against the repository were gathered from questions asked on `stackoverflow.com`, and two search engines were compared: the Google search engine pointed at our scraped repository, and Satsy pointed at our

[1]RQ1 was introduced and discussed in Section 3.1

scraped repository. Correctness of the search results was judged based on analyzing the search results and determining 1) if each result matches the input/output example from the `stackoverflow.com` questions, and 2) if a result matches the accepted answer from `stackoverflow.com` (i.e., the *oracle*).

In the second assessment toward *RQ2(b)* (Section 7.1.2), we reuse the queries from *RQ2(a)*. The goal of this study is to evaluate the benefits of using Satsy *after* a Google search. Thus, the repository is the Web as indexed by Google. Matches were identified as those that were returned by Google and *not* rejected by Satsy, and therefore were based on the source code behavior.

To address some of the limitations of the assessment regarding *RQ2(a)* (e.g., larger repository, human assessors of relevance), we performed another assessment toward *RQ3(c)* (Section 7.2). Here, the *repository* was built by scraping projects from `GitHub`, and we used humans to generate *queries* for three search engines: Google, a code-specific search engine called Merobase, and our search pointed at the newly-scraped repository. Since ultimately this approach is directed to help programmers, correctness of the search results was judged by humans.

7.1 Java Study 1

Comparing two search approaches requires the same query and the same space of potential matches (i.e., repository), so that the results can be compared. In this first Java study, we compare Satsy to a state-of-the-practice syntactic search, Google. Since our encodings are limited, we could not simply index all programs on the web as Google does, but we are able to control the space of potential matches using a local code repository. We then search for code using Satsy and Google, where both search approaches pull matches from the same local repository (*RQ2(a)*). To complement

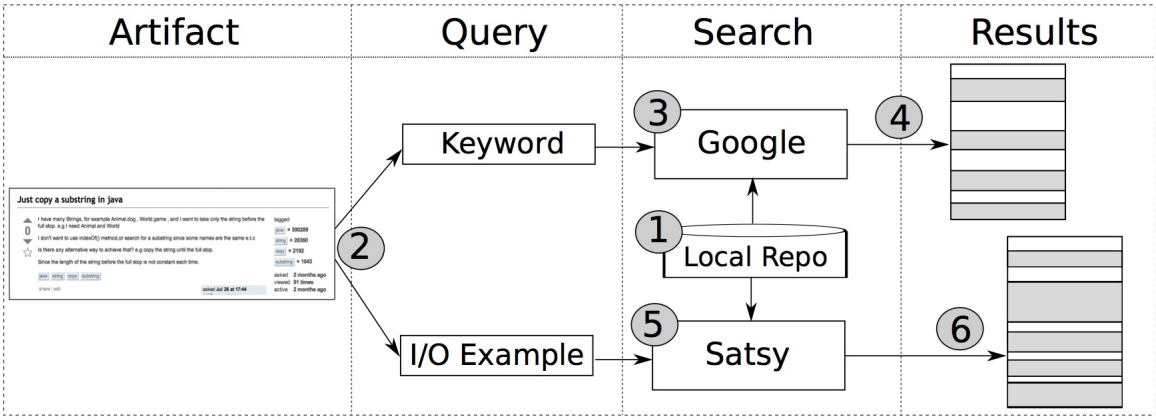


Figure 7.1: RQ2(a) Workflow

this study, and address in part the threat that constraining Google is not realistic in practice, we then evaluate how using Satsy after a syntactic search performed using unconstrained Google can reduce the code snippets that must be evaluated by the programmer (*RQ2(b)*) [70].

7.1.1 RQ2(a): Comparison to State-of-the-Practice

The goal of this experiment is to analyze the search results from Google and from Satsy for the purpose of evaluation with respect to the behavior of the code in the search results. Code behavior is analyzed from the point of view of the researcher in the context of questions asked on `stackoverflow.com` that can be answered with results from a local repository, where the queries for the searches were collected directly from the question text. This experiment aims to address *RQ2(a): How do search results from Satsy compare to those found using Google when searching over the same repository of programs?*

We use a local repository that we created and control to compare the relevance of results obtained by Satsy on the local repository against the relevance of the results obtained by a Google search engine pointed to our local repository. This process is

depicted in Figure 7.1. The first step is to collect artifacts for a local repository; all search results returned by Satsy and Google will come from this repository (see (1) in Figure 7.1). Next, we collect artifacts from `stackoverflow.com` with queries in two formats, keywords for Google and input/output for Satsy (2). The third step is to search Google given the keyword, and obtain the results (4). Next (5), we search Satsy with the input/output example and obtain results (6). The last step (not pictured) is to compute metrics and compare the results.

7.1.1.1 Artifacts

There are several artifacts required for this study, as indicated in the design. We discuss the two phases of artifact collection. First, we need to build a repository of programs to search over. Second, we need queries for the searches, which come from questions asked on `stackoverflow.com`.

Repository. We built a local repository on July 29, 2012 by issuing syntactic searches on Koders.com [37], a code search engine that indexes open source code repositories containing more than 15 million lines of code. The searches contained each of the `java.lang.String` functions supported by our encoding[2]. We then scraped all lines of Java source code that contained a call to at least one of the supported functions, totaling 5192 lines. We pruned out duplicates, lines that contained functions we do not support, and those that are not assignment or return statements. This left 713 unique snippets of code that form the Java code repository used in this evaluation. These lines of source code were not simply assignment statements, rather, they could

[2]When these artifacts were scraped, the following Java methods were not supported: `equalsIgnoreCase, isEmpty, replace, toUpperCase, toLowerCase`, and `trim`. Additionally, if-statements were not supported.

have multiple, nested method invocations. Table 7.2 summarizes the output types for each snippet in the repository.

Table 7.2: Output Types for Java Repository

Output Type	**Snippets**
String	255
Int	340
Boolean	42
Char	76
Total	713

Among the 713 snippets, there were a total of 793 calls to Java string methods, with an average of 1.11 method invocations per snippet. Table 7.3 shows the API calls and their frequency of appearance, in order of popularity. For example, *indexOf* was used most frequently with 206 instances, and *endsWith* was among the least frequently used with 18 instances. This illustrates the level of complexity and diversity in the snippets.

Table 7.3: Instances of API Calls in Java Repository for Study 1

indexOf	**substring**	**lastIndexOf**	**concat**	**charAt**
206	176	170	88	76

length	**endsWith**	**contains**	**startsWith**	**equals**
35	18	10	10	4

Queries. Satsy and Google use different query models, keyword for Google and input/output for Satsy, so we found a source that would allow us to extract both input/output examples and keywords to perform the searches. Using questions posted on `stackoverflow.com`, we use the posting title as the keyword query and the input/output example(s) as the query for Satsy.

Table 7.4: Java Artifacts Specifications for *RQ2(a)*

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

Of the 67 questions tagged in `stackoverflow.com` with `[java]`, `[string]`, and `[substring]`, 40 (60%) contain some form of explicit example. For 17 of those questions, our Java implementation supported encoding the input/output example(s). The remaining 23 involved constructs we do not support, such as regular expressions or arrays. The titles and input/output for the 17 questions are shown in Table 7.4. The *Q* column identifies the question number, and *Title* is the title as it appears in `stackoverflow.com`. Each of these questions has an input and output example, where the input and the output are each of type string. These are shown in the *LS from Q* column. For some questions, specifically, *Q 1*, *9*, and *13*, multiple examples were provided, and we consider all of those in the search.

7.1.1.2 Metrics

For *RQ2(a)*, to compare the results across the search techniques on the local repository, we use the number of results and *P@10* (the number of relevant results among the top 10), a common IR metric [10]. A relevant result is one that will behave as specified given an input/output example. In the case of uninitialized variables, the solver was free to find values that would make the snippet work, and in this way our measure of relevance was generous, though not biased toward any particular search approach. We simply allowed incomplete Java code as results.

To illustrate, consider for *Q 1*, ls_1 = ({"Animal.dog"}, "Animal"). The following Java source code matches where $filename \mapsto$ "Animal.dog" and $classname \mapsto$ "Animal", and thus would be considered a result.[3]

```
string classname = filename.substring(0, filename.indexof('.'));
```

[3]Binding of all variable names is required, as illustrated in Section 2.4, which is why the LHS of the assignment statement is mapped to the output from ls_1.

This solution has no free variables, and the solver returns *sat* given the mapping, so it is clear that this is a result. However, consider the following Java source code:

```
repository = location.substring(0, colon_index);
```

The mapping of *location* $\mapsto$ "Animal.dog" and *repository* $\mapsto$ "Animal" would result in a match. In this case, however, there is an additional free variable, *colon_index*. Since the solver can assign this variable to the string, ".", this is also considered a result. Yet, with mappings only of the components in ls_1, this code would not be executable. This is meant by *generous* matching; incomplete code can be identified as a result because the solver is free to assign values as needed. Contrastingly, the following Java source code would not be a result for ls_1:

```
s = s.concat(d);
```

No matter what variable bindings are used, the source code behavior does not match the example.

7.1.1.3 Implementation

This evaluation was an artifact-only evaluation and required no human input, outside of artifact collection. We ran this experiment on a generic laptop computer. Timing data in Section 7.1.1.4 for encoding was collected while running the encoder in Eclipse; timing data for solving was collected using the `-st` flag on the command line with Z3.

7.1.1.4 Results

The results for *RQ2(a)* are shown in Table 7.5. The *Q* column matches the specifications shown in Table 7.4. The next sets of columns, *Satsy* and *Google Local*, show the number of results and P@10 metrics for the two search approaches. Each row represents one of the 17 questions used for the queries.

Table 7.5: Java Results for *RQ2(a)*

	Satsy		Google Local	
Q	**#**	**P@10**	**#**	**P@10**
1	4	0.4	99	0.1
2	24	1.0	34	0.3
3	48	1.0	37	0.2
4	13	*1.0	100	*0.2
5	48	1.0	5	0.0
6	0	0.0	99	0.0
7	21	1.0	42	0.3
8	20	1.0	99	0.0
9	49	*1.0	41	*0.3
10	0	0.0	40	0.0
11	23	*1.0	70	0.3
12	13	*1.0	38	0.2
13	24	1.0	2	0.0
14	22	*1.0	38	0.3
15	13	1.0	42	0.2
16	14	1.0	2	0.0
17	13	1.0	34	0.2
Average	**20.5**	**0.85**	**48.4**	**0.15**

Key:
#: The number of results from the search
P@10: Relevant results from the search
(***** results match Stackoverflow)

Using Satsy, an average of 20.5 matches were found for each query, ranging from zero (in two cases, questions 6 and 10) to 49 results.[4] As an example, for *Q 2* in Table 7.4, given the input "23 14 this is random" and output "this is random", Satsy finds 24 matches, with $P@10 = 1.0$. To provide an example, the following source code matched the specification:

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

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

By the design of our semantics search approach implemented in Satsy, all the results are relevant in that they match the input/output specifications, so the P@10 metric will always be 1.0, unless there are fewer than 10 results, in which case it is still computed as the number of relevant results divided by 10. However, as we pointed out in Section 4.2, some matches may be coincidental. Without programmer input, differentiating the coincidental matches from the more relevant matches is not feasible.

When using a Google search engine on the local repository, on average, 48.5 matches were found for each query, ranging from two to 100. These results are under the *Google Local* column in Table 7.5. In *Q 2*, for example, we see that Google returns 34 results. Checking the top ten results against the input/output specifications reveals that only three of the top ten results were relevant. For example, the following snippet is *irrelevant* because it grabs the first part of a string, rather than the last part as required in the input/output example:

```
String string = string.substring(0, end);
```

On average, P@10 = 0.15, so 1.5 of the results are relevant, with a range from zero to three.

Comparing the retrieved results to the solutions proposed and positively voted on by the `stackoverflow.com` community shows that Satsy returns results that match the community solutions for five of the 17 searches (Q 4, 9, 11, 12, and 14, each marked with the *), and the Google results matched `stackoverflow.com` for two searches (Q 4 and 9). A match was determined if all API calls were the same between two snippets.[5]

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

To illustrate, Satsy returns a total of 49 snippets for *Q 9*, including snippets `s1` and `s2`:

```
s1. repository = location.substring(0, colon_index);
```

```
s2. String fieldname = line.substring(0, idx);
```

Google returned 41 results, including snippets `s3` and `s4`:

```
s3. String = string.substring(0, end);
```

```
s4. String axispart = mdxquery.substring(mdxquery.indexOf(select),
    mdxquery.indexOf(from));
```

Stackoverflow suggests snippet `s5` as a result:

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

While `s4` can be instantiated to fit the specification in *Q 9*, snippets `s1`, `s2`, and `s3` match the API calls used in `s5`.

In terms of performance, encoding all 713 snippets takes 2.991 seconds (averaged over ten runs), which is approximately 4ms per snippet. Among all the input/output examples in Table 7.4 and all searches, the average solver time to determine *sat* is 0.0483 seconds and to determine *unsat* is 0.0051 seconds. However, given that the snippets and the specifications are small, this may represent a best-case scenario. In the presence of larger and more complex programs, and larger and more complex specifications, these performance measures will drop, as we observe with the Yahoo! Pipes assessment (Section 8.1).

What we observed is that *using the same repository, Google returns over twice as many results as Satsy, but among the top ten, Satsy is over five times more effective at returning relevant results.* For four of the 17 searches (5, 8, 13, 16), Satsy provides matches when Google does not find any as the syntactic query was not rich enough

to identify results. Yet, this evaluation comes with many limitations, which we aim to address with a follow-up study for*RQ2(c)* (Section 7.2). Threats to validity are discussed in Section 9.

7.1.2 RQ2(b): Augmenting the State-of-the-Practice

The goal of this experiment is to analyze the search results from Google combined with Satsy. We aim to evaluate the the reduced space of source code that must be analyzed from the point of view of the developer in the context of running Satsy over the results from a Google query. The queries were drawn from those asked on `stackoverflow.com` that can be answered with results from a local repository (same queries as *RQ2(a)*). The goal of this experiment is to address *RQ2(b): How much can existing syntactic search approaches be improved by augmenting results with Satsy?*.

Under the hypothesis that semantic and syntactic approaches to code search are complementary, toward this goal and for *RQ2(b)*, we perform a Google search on the web (i.e., a syntactic approach) and explore how the results could be filtered and improved by also using Satsy. The results from Google form a repository of preliminary matches; with an input/output query, Satsy identifies which results are relevant, which are not, and which *might* be relevant (i.e., those that Satsy cannot encode).

The workflow for this study is presented in Figure 7.2, where the numbers correspond to steps in the workflow. First, we take a question from `stackoverflow.com` (*Artifact*) and extract the keyword and input/output queries, precisely as was done for *RQ2(a)*. Second, Google is searched using the keyword query; the output is a set of results (3). Source code snippets are extracted from the top 10 search results, and this forms a small repository of preliminary matches (4). Using the input/output query

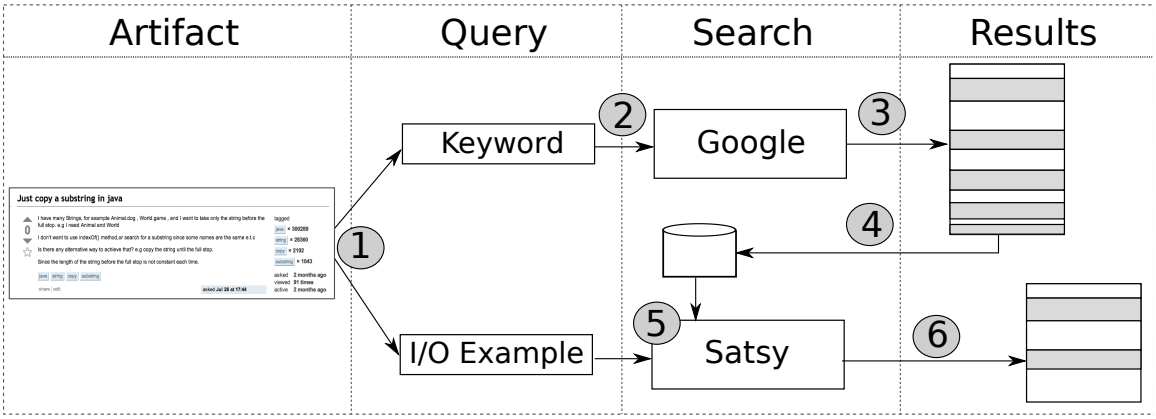


Figure 7.2: RQ2(b) Workflow

and the repository of preliminary matches built from the results in (3), Satsy searches for programs that match in (5). The final set of result in (6) represents either the source code that fits the input/output query, or that Satsy could not encode because, for example, it contained constructs not supported in our encoding, like arrays. The snippets Satsy does support *and* that return *unsat* for the input/output example are pruned from the search results. The result set in (6) represents the reduced set of results for the programmer to evaluate.

7.1.2.1 Artifacts

In this study, the repository of programs is built on-the-fly from the results of a Google query on the Web. The questions used for queries are the same as for *RQ2(a)*, shown in Table 7.4.

7.1.2.2 Metrics

For *RQ2(b)*, we define new metrics, *S@10* and *S'@10*. The metric *S@10* represents the number of code snippets returned in the top ten results from a syntactic Google search. We measure the number of snippets since the search results on the web may return pages

with multiple snippets of code. To capture *S@10*, we issue Google queries, then scrape and count the Java code snippets from the top 10 page results. This corresponds to the results in step (3) of Figure 7.2. In all searches, the `stackoverflow.com` question from which the keyword query was generated was always the top result; this was ignored as including it biased the results toward the question.[6] So, results 2-11 from Google were used to generate S@10.

The metric *S'@10* represents the number of snippets the programmer must evaluate after applying Satsy on top of the Google results. To capture *S'@10*, we first must encode all the snippets in *S@10*. Next, we run Satsy using the encoded *S@10* snippets as a repository and the example input/output as a specification, and discard snippets that are irrelevant, forming the result set in step 6 of Figure 7.2. This set of *Discarded* snippets represents those that the programmer *does not need to evaluate* by hand. *S'@10* is calculated as the difference (*S@10* − *Discarded*), representing the reduced space of snippets for the user to evaluate.

7.1.2.3 Implementation

As in *RQ2(a)*, this evaluation was artifact-only and required no human input, outside of artifact collection. We ran this experiment on a generic laptop computer.

7.1.2.4 Results

The results for *RQ2(b)* are shown in Table 7.6. The Q column matches the specifications shown in Table 7.4. The columns under the *Google + Satsy* label show results for *RQ2(b)*, including metrics *S@10*, *S'@10*, and *Discarded*. The *% Reduction* column quantifies the difference between S@10 and S'@10 as a percentage decrease.

[6] This restricted is lifted for the study toward *RQ2(c)* in Section 7.2.

Table 7.6: Java Results for *RQ2(b)*

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

Key:
S@10: One-line Java snippets from top 10 Google pages
Discarded: From S@10, Satsy returns *unsat*
S'@10: The reduced pool of snippets to evaluate
(* a *sat* snippet was found)
Reduction: The reduction in snippets to be evaluated

On average, $S@10 = 15$ snippets were gathered from the top 10 search results per Google search, with a range from zero to 38 (zero occurred when none of the retrieved pages were in the Java language). The number of *Discarded* snippets captures the space of results that the programmer would not have to examine if using Satsy to refine the Google results. The programmer must then look only at *S'@10* snippets. *Overall, the number of snippets returned could be reduced by 34% just by using our*

semantic search on top of the Google results. In two cases, for questions 4 and 12 (marked with *), at least one snippet returned *sat*, indicating that the snippet matches the specification and would be a solution. Since Satsy does not support the entire Java language, matches were not as common; for those snippets that we do support, most could be quickly discarded.

While Satsy can assist syntactic searches by removing irrelevant results, it should be noted that if a syntactic query misses a possible solution (i.e., a snippet of code that would provide a solution is not in the Google result set), then Satsy would not have the opportunity to evaluate that solution. Here again, the effectiveness of the search is dependent on the programmer's ability to write a query tied to documentation or syntax, a limitation that is addressed when Satsy is used in isolation.

7.2 Java Study 2

Similar to the study presented in Section 7.1.1 (differences *italicized*), the goal of this experiment is to analyze the search results from Google, *a code-specific search engine Merobase,* and Satsy for the purpose of evaluation with respect to *how relevant source code is to a programming task* from the point of view of *developers* in the context of questions asked on `stackoverflow.com`, where the queries for the searches were *created by developers.* The goal of this experiment is to address *RQ2c: How do search results from Satsy compare to those found using Google and those found with the code-specific search engine, Merobase, from the perspective of a programmer?*

This second Java study was designed to address four threats to validity of the first Java study in Section 7.1.1. First, the queries used for the first study were gathered from the titles of questions on `stackoverflow.com`. While an acceptable first step, these may not be representative of what programmers would actually search

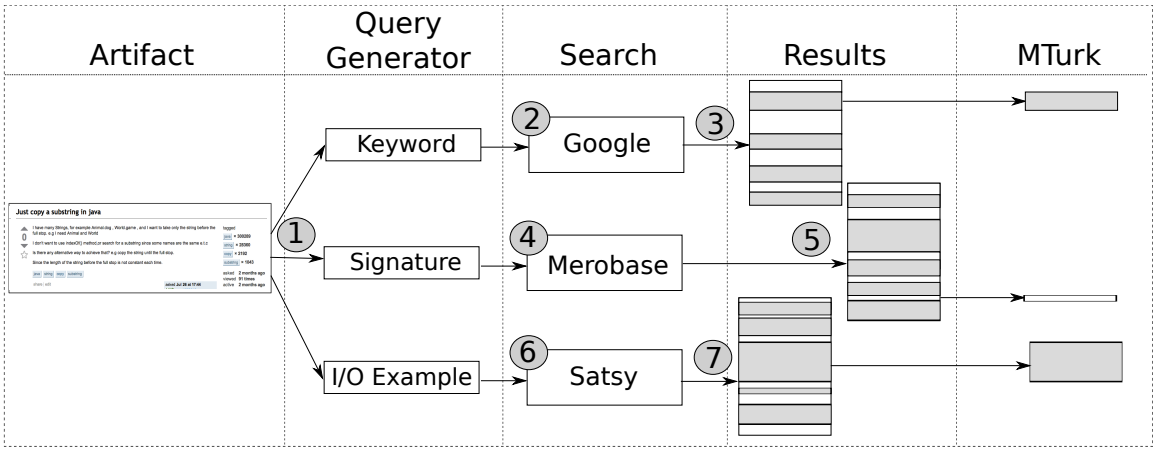


Figure 7.3: RQ2(c) Workflow

for online. In this study, we use human generators to write queries that are used for searching. This also allows us to get multiple queries for the same problem to increase the reliability of $P@10$ measures. Second, in the first study, by constraining Google to a local repository, it could not take advantage of the sophisticated ranking algorithms that have made it so effective. In this study, we lifted this restriction and perform searches on unconstrained Google. Third, as Google is not specifically designed to search for source code, we added an additional search, Merobase,[7] which has indexed over eight million open source Java components. Fourth, the ultimate judge of the relevance of a piece of source code is a human. While the accepted answers on `stackoverflow.com` give an indication of what a human finds to be the best answer, there may be a broader idea of relevance that we are not capturing. In this study, we use human participants to assess source code found in search results as relevant.

[7] `Merobase.com` is a source code search engine that has indexed over eight million components. It allows programmers to search using type signatures of their desired code, which is the feature we explored in this evaluation.

We describe the workflow for this evaluation in Figure 7.3. We begin by obtaining questions from `stackoverflow.com`. These questions are given to human generators (1), who create queries for the programming problem in three formats: keyword for Google, method signature for Merobase, and input/output for Satsy. Next, given the keyword query, a Google search is performed (2) and source code snippets are extracted from the results (3). This process is repeated for Merobase given the method signature (4), yielding a set of results (5). The same process is performed with Satsy given the input/output query (6), and results are obtained (7). Once results are obtained for all search approaches given each of several artifacts for each of several query generators, the code snippets are posted to Mechanical Turk where human participants evaluate the relevance of the code snippets to the programming task from the `stackoverflow.com` artifact.

7.2.1 Design

Search queries are generated with respect to an information need (we call this a *problem*) with a format dictated by the *search approach* that will be used. These form the two treatments that we manipulate in this experiment, and consider a full factorial design. To generate queries used on the searches, we use human *query generators*. For the purpose of analysis, the query generators are treated as the subjects, as they are assigned to combinations of the treatments. Each query is issued against a search engine to get a set of results. Each of the top 10 results is judged for relevance by a human; the average relevance among the top 10 is the response variable (P@10).

7.2.1.1 Treatment Structure

The treatment structure is full factorial because every level of every factor is combined with every level of every other factor. The factors and their levels are:

A Search approach (Google, Merobase, Satsy)
B Problem / Programming Task (22 problems)

Ultimately we want to compare the effectiveness of the search approaches (factor A) considering a syntactic search engine (i.e., Google), a code-specific search engine (i.e., Merobase), and a semantic search engine (i.e., Satsy). We carefully selected a variety of problems (factor B) because we want to compare the approaches' effectiveness under different contexts, under the hypothesis that certain search approaches are better suited for certain types of problems. Section 7.2.2.2 provides full details on the levels of factor B.

7.2.1.2 Experimental Design

We set up the study as a 3x22 factorial design with 12 subjects per group, where factor A is the search approach with three levels and factor B is the programming problem with 22 levels. To control costs we chose to reduce the size and treat this as a smaller pilot study; a full study is intended as future work. We therefore reduced the size of the deployed study to 3x8 with three subjects per group, which was the final size used for the analysis.

In this design, the subjects are the query generators, and we had 12 total subjects. In the reduced study, query generators were randomly assigned to each group until coverage of three subjects per group was reached (72 queries are needed to cover a 3x8 study with three subjects per group, so we used approximately six queries from

each of the query generators[8]). Relevance of the search results was determined by 30 human participants on Mechanical Turk (different from the query generators). For each query, 10 snippets of source code were gathered. Each snippet was evaluated by a different participant on Mechanical Turk, and each participant evaluated the relevance of 24 total snippets (eight from each level of Factor A, crossed with three from each level of factor B).

7.2.2 Artifacts

There are several artifacts required for this study, as indicated in the design and Figure 7.3. In this section, we discuss four phases of artifact collection. First, we need to build a repository of programs to search over (*Search* for Satsy, (6) in Figure 7.3). Second, we need problems for which solutions will be searched (*Artifact* in Figure 7.3). Third, we collect queries for the problems from the query generators ((1) in Figure 7.3). Fourth, we collect code snippets from the results of searches (*Results*, (3), (5), and (7) in Figure 7.3).

7.2.2.1 Repository

Two of the search approaches, Google and Merobase, already have repositories to search over. For Google, this is all the webpages it has indexed. For Merobase, this is eight million Java components from open source projects. For Satsy, we need to build the repository.

We collected a repository of programs by first scraping 2,952 projects tagged as `[java]` from `GitHub.com`, a project hosting website. These projects were scraped on February 3, 2013, and at that time, represented about 10% of the total Java projects accessible through the website. We explored all the **.java* files, accounting for 197,473

[8]Table 7.10 provides details on this.

files with over 700,000 methods. Of the files, 5,506 contained at least one method supported by the Java grammar in Figure 6.3. While we had the opportunity to reuse the repository from the the Java study toward *RQ2(a)*, we wanted to capture a more diverse set of behaviors. Unlike the study in Section 7.1.1, here, we considered entire methods rather than single-line programs.

We encoded 8,430 methods in total with 9,909 paths among the methods. The average paths per method was only 1.17, indicating that most of the supported methods had only one path. Of the methods 534 had multiple paths, with an average of 3.8 paths per multi-path method.

These numbers do seem low, and we identify two reasons for this. The first is that our encoding is missing some commonly-used constructs, like loops and the value, *null*. We will be able to support more methods as the language coverage grows (Section 11.2.4). The second reason follows from some limitations of the symbolic execution extension to JPF. The string processing within JPF is incomplete, meaning errors are thrown on some multi-path methods that our Java grammar (Figure 6.3) supports, and thus those methods are not encoded and stored. A small repository size does hurt the effectiveness of Satsy, but scraping more code seemed unlikely to increase the size and diversity of the existing repository without first addressing some of the language and implementation limitations (as we explore in Section 7.2.5.4).

7.2.2.2 Programming Tasks

Comparing the three search approaches requires queries in three different formats. To facilitate comparison across the search approaches, we gathered programming tasks from which query generators would write queries in three formats. One requirement is that each question needed to have input/output examples supported by our imple-

Table 7.7: Dimension 1: Input/output data types

Input Type \ Output Type	Char	Boolean	Integer	String
Char		20, 28		
Boolean		1	12	30
Integer		7, 22	23	18, 24, 25, 27, 29
String	13	5, 21, 26		2, 6, 8, 11, 31

mentation. We had an additional goal of type diversity to illustrate the applicability of this approach beyond string processing.

To obtain these programming problems, we first explored 3,500 questions on `stackoverflow.com` tagged with **[java]**, ordered by popularity, on April 16, 2013. The frequency of questions with input/output examples and solutions that could be supported by our implementation was scarce, and we obtained only 18 questions. Second, to increase the diversity of questions with respect to data types, we also looked at the first three pages with tags **[java][char]**, **[java][math]**, **[java][boolean]**, and **[java][integer][string]** on April 20, 2013. This yielded 13 more questions, for a total of 31. We then re-examined the questions to remove duplicates or very similar questions. The final count was 22 questions (i.e., nine questions were duplicates or very similar, such as "remove the first character of a string" and "remove the last character of a string", and we do not gain much diversity by having both). Mapping the input and output types of the examples puts the 22 questions into the categories described by the matrix in Table 7.7, which relates the *input type* of an example to its *output type*.

For example, question 5, "check if one string contains another, case insensitive" takes as input two strings and returns a boolean (output). It is worth mentioning that many of the categories in Table 7.7 are blank. This is mainly a function of implementation limitations. For example we are unable to perform relative character

Table 7.8: Programming Tasks for Query Generators

Q	**Description**	**Sample** LS
1	Check if two out of three booleans are true.	$(\{true, true, false\}, \{true\})$, $(\{false, true, false\}, \{false\})$
2	Left-pad integers in Java with zeros, so each resulting string has length 4	({“23”}, {“0023”}), ({“123”}, {“0123″})
5	Check if one string contains another, case insensitive	({“aBCd”, “cd”}, $\{true\}$)
6	Capitalize the first letter of a string	({‘‘foo”}, {“Foo”})
7	Determine if a number is positive	$(\{2\}, \{true\})$, $(\{-4\}, \{false\})$
8	Trim the file extension from a file name	({“foo.txt”}, {“foo”})
11	Trim the last character from a string	({“admirer”}, {“admire”})
12	Convert a boolean to an integer	$(\{true\}, \{1\})$, $(\{false\}, \{0\})$
13	Turn a string into a char	({“c”}, {‘c’})
18	Replace a character at a specific location in a string	({“fffoo”, 1, “b”}, {“fbfoo”})
20	Determine if a character is numeric	({‘5’}, $\{true\}$), ({‘F’}, $\{false\}$)
21	Check if one string is a rotation of another (a rotation is when the first part of a string is spliced off and tacked onto the end)	({“$stack''$, “$cksta''$}, $\{true\}$), ({“$stack''$, “$stakc''$}, $\{false\}$)
22	Check if two integers sum to a third	$(\{2, 5, 3\}, \{true\})$, $(\{2, 5, 8\}, \{false\})$
23	Get the next number in a range given a step size (e.g., range [1,10], step = 3, current number = 4. Next = 7)	$(\{4, 1, 10, 3\}, \{7\})$, $(\{4, 1, 10, -3\}, \{1\})$
24	How do you format the day of the month with the proper suffix, for example “th” for “11th” or “nd” for “2nd”?	({11}, {“th”}), ({2}, {“nd”}), ({1}, {“st”})
25	How to append “0” i times at the end of a string? For example,	({3, “fab”}, {“fab000”})
26	I want to determine if a string is not all whitespace.	({“ 5 ”}, $\{true\}$), ({“ ”}, $\{false\}$)
27	How to truncate a string that is beyond a certain length with ellipses?	({“How about a new car”, 15}, {“How about a ...”})
28	How to tell if a character variable is within a string?	({“abc”, ‘c’}, $\{true\}$), ({“abd”, ‘c’}, $\{false\}$)
29	Create a String method to return a value based on an int. Example: return ”young” when age is 18, ”old” when age is 30.	({18}, {“young”}), ({30}, {“old”})
30	Get the string value for a boolean	($\{true\}$, {“true”})
31	I want to convert a String 35634646 to have the thousand ”,” so it should be 35,634,646	({“2345678”}, {“2,345,678”}), ({“12345”}, {“12,345”})

manipulations, such as incrementing a character, we do not support converting a character to a string (though the reverse is supported), and comparing characters lexicographically is not supported. This rules out many of the categories involving characters. Additionally, getting the integer value of a string was a frequent question, and such a solution is unsupported by our implementation (unless the conversion uses a series of if-statements on the string values, which would be a unlikely to find in existing code).

The final set of questions, and sample LS, are shown in Table 7.8. To understand common *themes* among the questions, we looked at the relationships between the inputs and the outputs, categorizing the questions as shown in Table 7.9. The idea is that search algorithms have different strengths in different classes of problems, and this is a potential classification we wanted to explore. The first category, *checking a property of an input*, covers questions like, *Q 1*: "check if two booleans are true," or *Q 22*: "check if two integers sum to a third." The second category, *manipulating the input*, covers questions like *Q 6*: "capitalize the first letter of a string." The final input, *computing something new based on input*, covers type conversion like *Q 30*: "get the string representation of a boolean."

Table 7.9: Dimension 2: Input/output Relationships

Category	Questions	Total
Checking a property of the input	1, 5, 7, 20, 21, 22, 26, 28	8
Manipulating the input	2, 6, 8, 11, 18, 25, 27, 31	8
Computing something new based on input	12, 13, 23, 24, 29, 30	6

As mentioned in Section 7.2.1, due to the cost of the large study, we reduced the levels of factor B (the problems) from 22 to eight, which seemed like a large enough number that our results would not be tied to the particular problems selected (as we explore in Section 7.2.5). The final set of questions were 5, 6, 7, 8, 11, 13, 20, and 21.

We selected the first eight questions for which Satsy returned at least 10 snippets (the number 10 was selected based on our selection of the P@10 metric)[9]. This criteria of requiring matches from Satsy was used out of necessity. It is hard to compare other search algorithms to ours when we do not produce any results. However, this may have biased the results toward our analysis. A discussion on the steps we took to combat this threat follows in threats to validity (Section 9.2.1).

7.2.2.3 Query Generation

The query generators were responsible for generating queries in three formats for each of the questions in Table 7.8. The generators were given paper packets with 22 pages; the questions in each packet were randomly ordered using the Fisher-Yates algorithm to control for learning effects. A sample page is shown in Figure 7.5 for *Q 1*. The question is stated at the top, followed by a box for a descriptive (keyword) query, a type signature for the desired code (for Merobase), and input/output pairs for Satsy. The full instructions given to the query generators are provided in Figure 7.4.

Twelve query generators were used in this phase. These generators were graduate students and staff in Computer Science and Engineering at UNL. Each participant was compensated for their time with a $10 gift card to `Amazon.com`. Since these query generators were not subjects being evaluated, IRB approval was not required for this stage of the research.

The end result was that we obtained 12 queries for each of the 22 problems, for each of the three search approaches. While we collected queries for all 22 questions, as discussed, only 8 problems were used in the implementation of the study. Additionally, only three query generators were assigned to each combination of search approach and problem. These query generators were randomly assigned to groups, with the added

[9] As a point of interest, we investigate the total number of matches by our search in Section 7.2.5.3.

1. Read each problem description
2. Pretend that you are a Java developer who needs to perform the task
3. Instead of starting from scratch, you have chosen to search for an existing solution. Generate search queries in the following formats (examples follow):
   a) Descriptive: using keyword, as you would use when searching Google for a solution.
   b) Signature: using the method name (optional), input types, and output type, as you would use when searching a code-specific search engine like www.merobase.com
   c) Semantic: using example input(s) and output of a method that behaves like the problem description. These are like test cases, and you can use as many input/output pairs as you see fit.
4. In all of the above, limit the data types to String, boolean, char, and int. You have one hour to complete these tasks.

Figure 7.4: Query Generator Instructions

requirement that all the queries within the groups be distinct to encourage variety in the results. This last requirement was added after some of the type signature and input/output queries from the generators, for some problems, turned out to be identical.

Table 7.10 shows the allocation of generators to groups. For example, with Google and problem 5, generators 1, 4, and 6 were assigned. For problem 5 and Merobase, generators 3, 5, and 7 were assigned. For Satsy on problem 5, generators 8, 9, and 10 were assigned.[10]

7.2.2.4 Snippet Retrieval

With the queries created by the generators, the next step was to execute the searches and gather the top 10 code snippets for each search. For each problem and search

[10]Merobase queries were never gathered from generator 12 because, as that person remarked on their packet, they would never search in this way and therefore did not provide those queries.

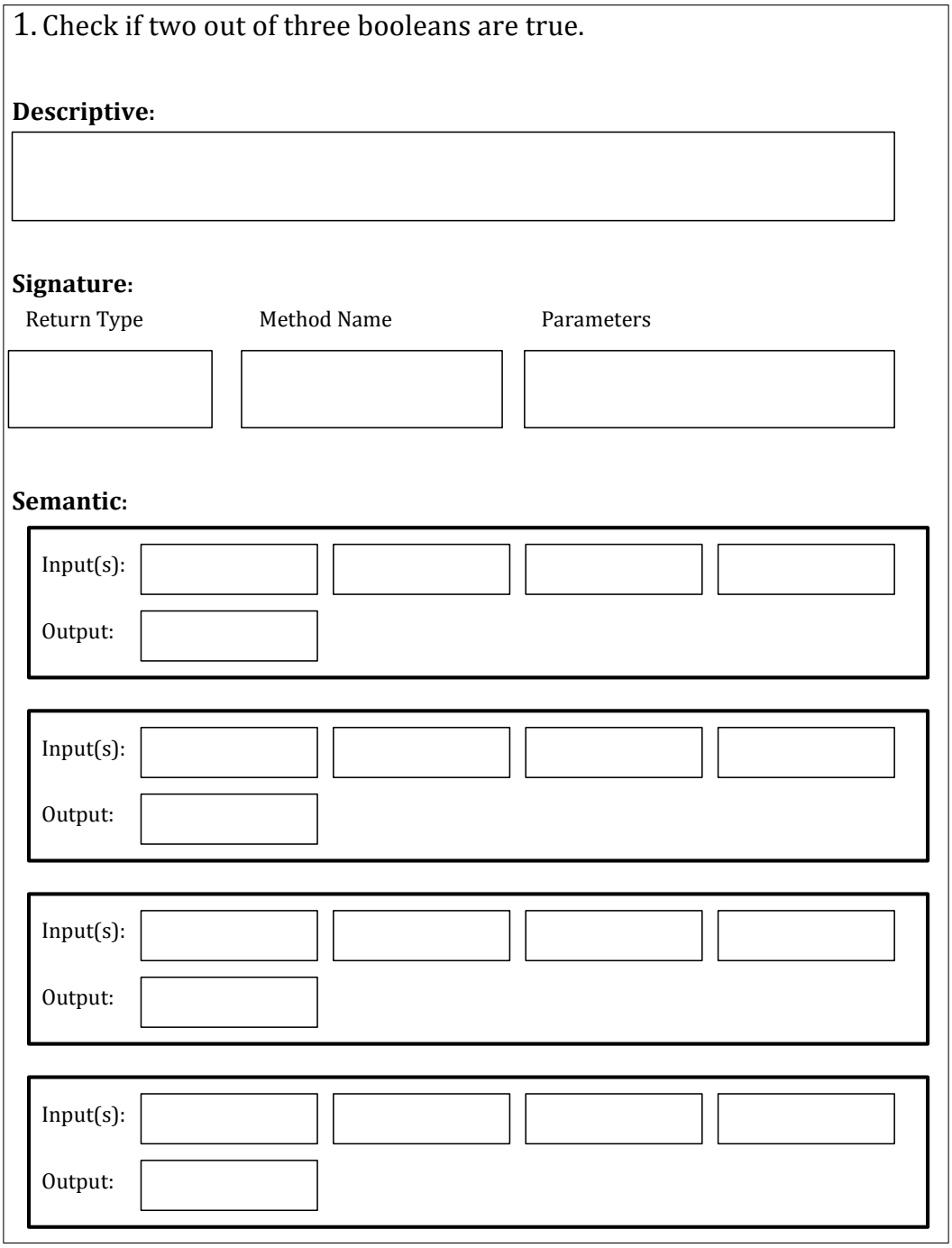


Figure 7.5: Query Generator Packet Page for Q1

Table 7.10: Allocation of query generators to search approach and problem

	Search Approach		
Problem	**Google**	**Merobase**	**Satsy**
5	1, 4, 6	3, 5, 7	8, 9, 10
6	3, 5, 7	7, 8, 9	1, 10, 12
7	8, 9, 10	5, 7, 11	1, 4, 6
8	2, 11, 12	1, 4, 6	3, 5, 7
11	6, 7, 10	1, 3, 10	1, 2, 8
13	1, 2, 3	4, 8, 11	5, 6, 9
20	4, 8, 12	5, 9, 11	2, 7, 10
21	5, 9, 11	6, 7, 10	1, 2, 3

combination, we performed three searches, where the queries were gathered according to the allocation of query generators to groups. That is, if generator 1 is assigned to the Google, Problem 5 group, then we take the keyword query generated by generator 1 for Problem 5, and perform a search.

The following describes how the searches were performed and the snippets gathered for the three search approaches, Google, Merobase, and Satsy. Per the study design, our goal to was to obtain 10 snippets of source code for each search. In some cases, like with Google, this required visiting more than the top 10 results. A discussion of this threat to validity is in Section 9.

Google: The Google queries were conducted using the Google Chrome browser in incognito mode with cleared cookies and browser history.

For each of the top 10 results from a search, we clicked through to the webpage and grabbed the first code snippet and type signature, if available. Often, results would not be provided in Java, but in another language like PHP or C#. If that was the case, the snippet was still collected and provided as a potential result since it may provide an idea for a solution that could be relevant in Java. This addresses,

in part, a threat to validity of the study in Section 7.1.2 where we ignored non-Java results from a Google query. Here, we include these as they may be informative to the programmer. When a webpage had no code snippet, we simply moved to the next one until 10 pages with code snippets had been visited. For some problems and some query generators, we had to visit more than 13 webpages to get enough snippets. With problem 20 and generator 4, there were three pages that returned no code. For problem 21 and query generator 11, there were also three pages that returned no code. However, this may have increased the performance of Google artificially since we are considering snippets outside of the boundaries for the $P@10$ metric. For 13 of the 24 queries issues to Google, we had to visit at least one additional page to find source code. Among those 13 queries, a total of 22 of the top 10 results (approximately 10% of all webpages in the top 10 visited) had no code snippets. We discuss this threat to validity in Section 9.4.2.

When a result would come up that was from the original question asked on `stackoverflow.com`, from which the problem was taken, the first snippet (often the community-accepted response) was collected as a result. This is a change from the studies in Section 7.1 where such results were ignored. We did this so as not to bias the results against Google. The end result was 30 code snippets for each problem (10 from each assigned query generator). If there were duplicate snippets, these were retained in the set of 30. This happens when the same webpage appears in the top 10 results from different queries.

Merobase: For each problem and for each of the signature queries, we conducted a query on `Merobase` on May 19, 2013. As the type signatures created by the query generators assumed Java, we restricted the results to Java only using a flag in the advance search options. For each of the top 10 results, we clicked on the files and

copied the method with the matched signature. In the results, if a file contained multiple matches for a signature, it appeared in the results once for each matched signature. The end result was 30 Java methods for each problem. Duplicates were retained.

Satsy: For each problem and each input/output query, we searched Satsy for source code using the repository described previously in this section. We used a ranking algorithm similar to that in Section 5.3.2.1, and fully described in Section 7.2.5.3. Our initial search was constrained such that the type signature of the query/specification had to be equal to the type signature of the snippets being evaluated (i.e., $TS_q = TS_{LS}$ was a precondition for a path q to be considered as a potential match and sent to the solver, Section 5.3). If not enough matches were returned at this level, we relaxed the matching so $TS_q \supset TS_{LS}$, in order to obtain at least 10 matches. The end result was 30 Java methods for each problem. Duplicates were retained.

7.2.3 Metrics

For *RQ2(c)*, to compare the results across the searches, we use *P@10*. In this study, relevance means *the source code can be easily adapted to solve the problem.* A human participant decides whether or not a code snippet is relevant to a problem description. We introduce an additional metric, *Q@10*, which represents the number of results among the top 10 that *solve* the problem. In this study, solving means *the code seems to work as is, without modification.* A human participant decides whether or not a code snippet solves a problem based on its description.

7.2.4 Implementation

After collecting all the code snippets, calculating the metrics involves human participants. We created experimental tasks that presented people with snippets and programming tasks; their responses allowed us to calculate the *P@10* and *Q@10* metrics. The study was deployed on Amazon's Mechanical Turk (this platform is described in Section 3.1.1).

7.2.4.1 Tasks

Creating experimental tasks for this study involves the composition of basic tasks. A *basic task* presents a participant with a code snippet and asks if that snippet is *relevant* to a problem description, and if the snippet *solves* the problem. The definitions for relevance and solving are included in the basic task, and short-answer responses are required to explain their choices. Figure 7.6 shows a basic task with code snippet and the relevance/solves questions, taken from *Q 21*.

We call a problem description and its three associated basic tasks a *problem group*. Each problem description presented to a participant is accompanied by three basic tasks, where one basic task is created from each search approach. That is, given a problem description, the participant views three snippets, one from Google, one from Merobase, and one from Satsy, and determines if each is relevant to the problem, and if each solves the problem. The participant is not made aware of the origins of the snippets. Additionally, the participants were instructed to generate an input and output for the problem description; this is used by the researchers to assess participant understanding of the problem description, and is part of the problem group. An example of a problem description and the input/output is shown in Figure 7.7 for *Q 11*.

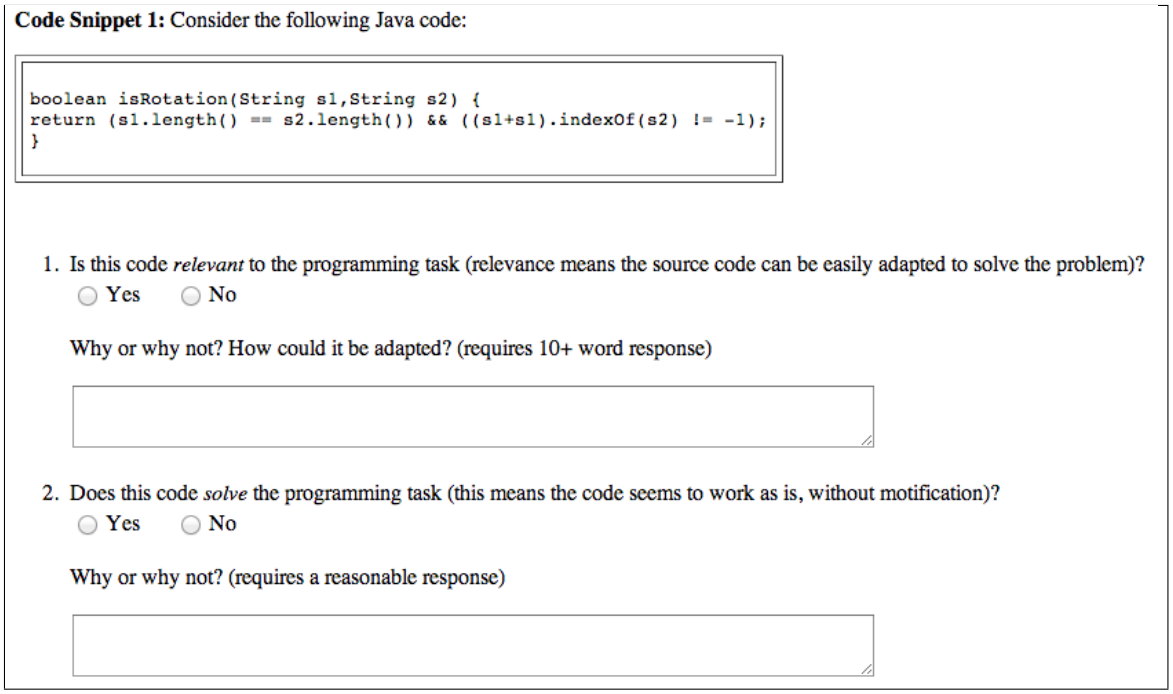


Figure 7.6: Basic Task: Code Snippet and Related Questions

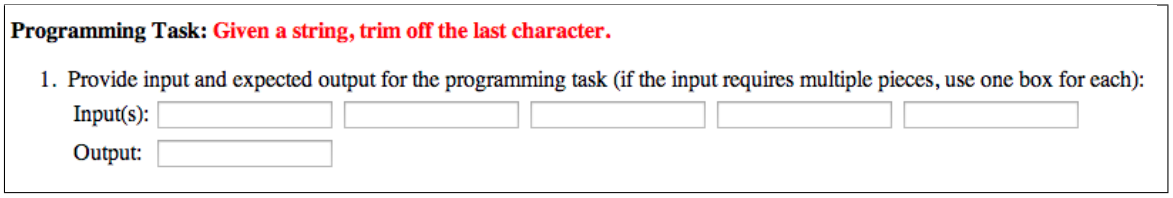


Figure 7.7: Problem Group: Problem Description and I/O Example

An *experimental task* is composed of eight *problem group*s, with one from each problem description used in this phase of the experiment. To create an experimental task, we perform four randomizations. To describe this process, we start with the largest unit, the experimental task, and work our way down to basic tasks. All randomizations are performed to control for learning effects.

Within an experimental task, the ordering of problem groups is randomized. Within each problem group, the ordering of basic task search approaches is randomized. That is, some problem groups show a snippet from Google first, followed by Satsy and then

Merobase. Others show a snippet from Merobase first, followed by a snippet from Google and then from us, and so forth. Within a basic task, the selection of query generator from which the snippet was gathered is random. Within the search results from the query generator, the selection of snippet is random (i.e., it could come from any rank in the results).

7.2.4.2 Deployment

Each experimental task is implemented as a *human intelligence task*, or HIT, on Mechanical Turk. We created 30 HITs, so each search result from each query generator assigned to a search approach / problem group appeared exactly once.[11]. Each HIT paid $3.25[12] and each participant could complete one HIT. The study was available from May 28, 2013 until June 04, 2013.

7.2.4.3 Participants

A prerequisite for participating in the study was to pass a qualification test. This included four Java questions to ensure participants are reasonably competent with Java before participating in the study. Appendix C.1 provides the questions. This is also how the IRB informed consent was delivered; this information is provided in Appendix C.2. Scoring at least 50% on the Java questions, being 19 years of age, and accepting the IRB informed consent allowed participants access to the HITs.

[11] 30 HITs * 8 problems * 3 snippets per problem = 720 snippets. In the study, we had 3 search approaches * 8 problems * 3 generators * 10 search results = 720 snippets

[12] The informed consent in Appendix C.2 was originally written with 22 tasks each paying $0.20, for a total compensation of $4.40. The study was re-designed with one task containing eight problems and paying $3.25. The informed consent remained the same and the change in design was accepted by the IRB.

Recruitment was managed by Mechanical Turk and using a post on the HITsWorthTurkingFor page of `reddit.com` advertising the study. There were 30 Mechanical Turk workers who participated in the experiment.

7.2.5 Results

Toward *RQ2(c)*, we computed P@10 and Q@10 for every generator in every group of search approach and problem. These are the response variables. However, as the P@10 and Q@10 metrics are strongly correlated (Spearman's $r = 0.718138$), we concentrate primarily on P@10 for this analysis.

We begin by reporting descriptive statistics for P@10 in Table 7.11. In this factorial design, the linear model is as follows, with annotations following:

$y_{ijk} = \mu + A_j + B_k + AB_{jk} + \epsilon_{ijk}$.

$j = \{\text{Google, Merobase, Satsy}\}$	The search approaches
$k = \{5, 6, 7, 8, 11, 13, 20, 21\}$	The problems

In Table 7.11, Factor A is represented in the columns, and factor B is represented by the rows. Within each cell, the P@10 statistic is reported for each query generator assigned to the group, along with the cell (or group) mean. For example, $X_{1,g,5} = 0.700$ reports the P@10 value for one query generator assigned to the Google/Problem 5 group. Row means for Factor B are included, as are column means for Factor A. For example, $\bar{x}_{.5} = 0.556$ represents the mean P@10 value across all search approaches for problem 5. For columns $\bar{x}_{g.} = 0.675$ represents the mean across all problems for the Google search approach. The grand mean is, $\bar{x}_{..} = 0.528$, and variance $\sigma^2 = 0.068$.

From the descriptive statistics, Google has the highest P@10, followed by Satsy and then Merobase. Though not reported, for the Q@10 metric, the ranking of search

Table 7.11: P@10 Response Matrix

Problem (Factor B)	Search Approach (Factor A) Google	Merobase	Satsy	
5	$x_{1,g,5} = 0.700$	$x_{1,m,5} = 0.500$	$x_{1,s,5} = 0.200$	
	$x_{2,g,5} = 0.800$	$x_{2,m,5} = 0.800$	$x_{2,s,5} = 0.600$	
	$x_{3,g,5} = 0.400$	$x_{3,m,5} = 0.200$	$x_{3,s,5} = 0.800$	
	$\bar{x}_{g,5} = 0.633$	$\bar{x}_{m,5} = 0.500$	$\bar{x}_{s,5} = 0.533$	$\bar{x}_{.5} = 0.556$
6	$x_{1,g,6} = 0.500$	$x_{1,m,6} = 0.200$	$x_{1,s,6} = 0.400$	
	$x_{2,g,6} = 0.700$	$x_{2,m,6} = 0.100$	$x_{2,s,6} = 0.500$	
	$x_{3,g,6} = 1.000$	$x_{3,m,6} = 0.200$	$x_{3,s,6} = 1.000$	
	$\bar{x}_{g,6} = 0.733$	$\bar{x}_{m,6} = 0.167$	$\bar{x}_{s,6} = 0.633$	$\bar{x}_{.6} = 0.511$
7	$x_{1,g,7} = 0.600$	$x_{1,m,7} = 0.200$	$x_{1,s,7} = 0.300$	
	$x_{2,g,7} = 0.500$	$x_{2,m,7} = 0.200$	$x_{2,s,7} = 0.500$	
	$x_{3,g,7} = 0.600$	$x_{3,m,7} = 0.700$	$x_{3,s,7} = 0.400$	
	$\bar{x}_{g,7} = 0.567$	$\bar{x}_{m,7} = 0.367$	$\bar{x}_{s,7} = 0.400$	$\bar{x}_{.7} = 0.444$
8	$x_{1,g,8} = 0.700$	$x_{1,m,8} = 0.200$	$x_{1,s,8} = 0.800$	
	$x_{2,g,8} = 0.900$	$x_{2,m,8} = 0.600$	$x_{2,s,8} = 0.600$	
	$x_{3,g,8} = 0.400$	$x_{3,m,8} = 0.200$	$x_{3,s,8} = 0.600$	
	$\bar{x}_{g,8} = 0.667$	$\bar{x}_{m,8} = 0.333$	$\bar{x}_{s,8} = 0.667$	$\bar{x}_{.8} = 0.556$
11	$x_{1,g,11} = 0.900$	$x_{1,m,11} = 0.300$	$x_{1,s,11} = 0.400$	
	$x_{2,g,11} = 0.900$	$x_{2,m,11} = 0.800$	$x_{2,s,11} = 0.100$	
	$x_{3,g,11} = 0.400$	$x_{3,m,11} = 0.800$	$x_{3,s,11} = 0.900$	
	$\bar{x}_{g,11} = 0.733$	$\bar{x}_{m,11} = 0.633$	$\bar{x}_{s,11} = 0.467$	$\bar{x}_{.11} = 0.611$
13	$x_{1,g,13} = 0.700$	$x_{1,m,13} = 0.600$	$x_{1,s,13} = 0.900$	
	$x_{2,g,13} = 0.400$	$x_{2,m,13} = 0.700$	$x_{2,s,13} = 0.800$	
	$x_{3,g,13} = 0.700$	$x_{3,m,13} = 0.500$	$x_{3,s,13} = 0.800$	
	$\bar{x}_{g,13} = 0.600$	$\bar{x}_{m,13} = 0.600$	$\bar{x}_{s,13} = 0.833$	$\bar{x}_{.13} = 0.678$
20	$x_{1,g,20} = 0.400$	$x_{1,m,20} = 0.200$	$x_{1,s,20} = 0.400$	
	$x_{2,g,20} = 0.800$	$x_{2,m,20} = 0.400$	$x_{2,s,20} = 0.400$	
	$x_{3,g,20} = 0.800$	$x_{3,m,20} = 0.200$	$x_{3,s,20} = 0.600$	
	$\bar{x}_{g,20} = 0.667$	$\bar{x}_{m,20} = 0.267$	$\bar{x}_{s,20} = 0.467$	$\bar{x}_{.20} = 0.467$
21	$x_{1,g,21} = 0.700$	$x_{1,m,21} = 0.100$	$x_{1,s,21} = 0.100$	
	$x_{2,g,21} = 0.900$	$x_{2,m,21} = 0.200$	$x_{2,s,21} = 0.300$	
	$x_{3,g,21} = 0.800$	$x_{3,m,21} = 0.100$	$x_{3,s,21} = 0.400$	
	$\bar{x}_{g,21} = 0.800$	$\bar{x}_{m,21} = 0.133$	$\bar{x}_{s,21} = 0.267$	$\bar{x}_{.21} = 0.400$
	$\bar{x}_{g.} = 0.675$	$\bar{x}_{m.} = 0.375$	$\bar{x}_{s.} = 0.533$	$\bar{x}_{..} = 0.528$
	$\sigma_g^2 = 0.036$	$\sigma_m^2 = 0.062$	$\sigma_s^2 = 0.065$	$\sigma^2 = 0.068$

algorithms is the same; Google scored the highest with $Q@10 = 0.338$, followed by Satsy at $Q@10 = 0.233$ and Merobase with $Q@10 = 0.075$.

For problem 21 and Google, $\bar{x}_{g,21} = 0.800$. The search results for all three generators had the question on `stackoverflow.com` from which the task was generated, as the first search response. This may have contributed to the high P@10 value. On the other hand, the lowest relevance scores for Merobase and Satsy correspond to problem 21. For problem 11 and Merobase, $\bar{x}_{m,11} = 0.633$. This problem was also Google's second-highest, and for Satsy it ranked in the middle. While relevance was high, $Q@10$ for Merobase was still low at 0.100. Satsy performed the best on problem 13, and this was the best overall performance across all search approaches and all programming problems with $\bar{x}_{s,13} = 0.833$. The highest $Q@10$ metrics over all questions and search approaches come from Satsy on problems 6 and 8, where $Q@10 = 0.500$ for both.

To understand why differences in the means were observed for the three search approaches, we use a 2-factor ANOVA.[13] Table 7.12 presents the ANOVA with P@10 as the dependent variable, considering factor A (Search), factor B(Problem), and the interaction (Search:Problem).[14] The null hypothesis is $H_0 : \mu_g = \mu_m = \mu_s$, where μ_g is the mean for Google, μ_m is the mean for Merobase, and μ_s is the mean for Satsy. The F-ratio is significant only for the Search factor at $\alpha = 0.001$, indicating that the observed differences in means among the search approaches are not likely due to chance.

[13]The design of the experimental set-up is a 3x22 factorial, though several levels of randomization were required to deploy this experiment on Mechanical Turk. This could have been considered a split-plot design where the 3x8 matrix is the whole-plot, split among the 30 Mechanical Turk participants. However, due to lack of replication in that experimental design, we chose the simpler and more conservative analysis of a 3x22 factorial design, perhaps at the cost of power that could have resulted from blocking on the Mechanical Turk participant.

[14]We analyze the study as a full factorial in the R statistical package. The details of the R analysis are found in Appendix C.3.

Table 7.12: Dependent Variable: P@10, test of between subjects effects

Source	**df**	**Sum Sq**	**Mean Sq**	**F-Ratio**	**Pr(>F)**	**Sig**
Search	2	1.08111	0.54056	11.6527	$7.496*10^{-05}$	***
Problem	7	0.52444	0.07492	1.6151	0.1540	
Search:Problem	14	0.99222	0.07087	1.5278	0.1373	
Residuals	48	2.22667	0.04639			

Signif. codes: ***$\alpha = 0.001$, **$\alpha = 0.01$, *$\alpha = 0.05$

We are specifically interested in the performance of Satsy compared to the other search approaches, with the one-sided null hypothesis that $H_0 : \mu_g \geq \mu_s$, and alternative, $H_a : \mu_g < \mu_s$, for Google. Replacing μ_g with μ_m forms the hypotheses for Merobase. Assuming non-normality of the data, the results of this analysis are shown in Table 7.13. We reject the null hypothesis at $\alpha = 0.05$ for Merobase, but we do not reject the null hypothesis for Google. So, we can say that Satsy out-performs Merobase, but not Google.

Table 7.13: Test of Means among Search Algorithms

Response	**Test**	**p-value**	**Test**
P@10	$H_0 : \mu_g \geq \mu_s$	$p = 0.9774$	Mann-Whitney
P@10	$H_0 : \mu_m \geq \mu_s$	$p = 0.0163$*	Mann-Whitney

Signif. codes: ***$\alpha = 0.001$, **$\alpha = 0.01$, *$\alpha = 0.05$

It seems rather clear that Google outperforms the other search approaches in terms of relevance, but it is unclear *why* this is the case. We have several conjectures that we plan to explore in future work, and some that we touch on in the sections that follow.

The query generators were graduate students in computer science and were accustomed to writing Google queries, so it is unclear if Google's good performance was a function of excellent queries. Along this same vein, the problem descriptions were taken from stackoverflow, so the queries from the generators may have been

tightly bound to those descriptions. This would artificially increase the performance of Google. As discussed as part of the snippet retrieval for Google, we bypassed 22 webpages among the top 10 results for the 24 queries (approximately 10% of the results Section 7.2.2.4). This may have had an impact on the relevance for Google, since we assisted its performance by finding source code beyond the top 10 results. Additionally, for all three of the queries for question 21, the top result was the `stackoverflow.com` webpage.

Other factors that could have brought down the performance of Satsy include the richness of the repository, our chosen ranking algorithm, or the expressiveness of the input/output format. These are just some of the future directions to explore in seeking to improve this work.

7.2.5.1 On Types of Problem

While factor B (problem) does not account for the differences in means according to Table 7.12, the search approach performance may be dependent on the problem *type*. Only one of the problems used in the study, problem 13, fell under the *computing something new* category in Table 7.9, but this was the highest performing problem for Satsy. The next highest performances came from questions 6 and 8, both of which involve *manipulating the input*. Our search may be particularly well suited for those types of problem. Further study is needed here, with more coverage on the *computing something new* category to compare performance across problem type.

7.2.5.2 MTurk participants

One concern we had about the reliability of the results related to fatigue on the part of the Mechanical Turk participants. The average completion time for the experimental tasks was 54 minutes and 40 seconds, so fatigue was a concern, providing an internal

threat to validity (see Section 9.1). As a cursory check, we looked at the correlation between when a snippet appeared in the experimental tasks (i.e., its order) and the participants likelihood of answering 'yes' for the relevance question. There was very low correlation between the order and frequency of 'yes' responses (Spearman's $r = 0.0544$). This is an indication that the participants likely did not simply get lazy and start marking 'no' toward the end. Using short-answer questions in each of the basic tasks (Figure 7.6) was intended to combat this as well.

7.2.5.3 On the Impact of Ranking on Results

While commercial search tools like Google and Merobase have the opportunity to observe user behavior and refine their ranking algorithms, the ranking used by Satsy was less refined and based entirely on conjecture. In the ranking, we grouped the matched snippets according to the strength of the match, using a modified ranking to that presented in Section 5.3.2.1. Here, we discuss the relevance of search results at each rank based on search approach. Then, we discuss in further detail the ranking used in Satsy and how that ranking could be improved in the future based on the responses of the Mechanical Turk participants with regards to relevance.

Looking at the relevance among items at various ranks allows us to compare the Satsy ranking algorithm to the others. Table 7.14 shows, for each search approach, the average relevance of snippets that appeared at rank 1, rank 2, and so forth down to rank 10. For all ranks, except 7 and 10, the relevance of the Google results were the best. However, at rank 7, Satsy's relevance was higher and at rank 10, they were tied.

We would expect the relevance at the ranks to follow a decreasing function. Fitting a linear model to the Google relevance yields a slope coefficient of -0.0202. For Merobase and Satsy, on the other hand, the slopes are much flatter at -0.0085 and

-0.0009, respectively. This leads us to conjecture that the ranking algorithms may have an impact on the overall relevance of the results.

For the ranking algorithm used in Satsy, Table 7.15 presents the bucket approach that was used to assign results to ranks. Each bucket number is in parentheses in a cell in the table. The columns represent the amount of LS that is satisfied by a particular problem P. The rows represent the amount of a program P that satisfies LS. The first column says that exactly one $ls \in LS$ is matched, the second column indicates that some $LS_{sat} \subset LS$ and $|LS_{sat}| > 1$. The final column indicates that all $ls \in LS$ are satisfied. The first row indicates that a subset of the paths in P are matched, whereas the second row indicates that all the paths in P are matched by LS_{sat}.

Within the cells for the last two columns, there is a saturation criterion that further divides each bucket. Consider, for example, when $Q_{sat} = Q_P$ and $LS_{sat} = LS$ both hold. We could have the case of bucket (1) when every $ls \in LS$ exercises a distinct path in P, or if there is a path in P that is covered by multiple $ls \in LS$, then it would fit in bucket (2). That is, the paths for a program in bucket (2) are more saturated than a program in bucket (1). For the ranking algorithm used by Satsy in this study, we selected a snippet from each bucket in round-robin fashion, using the following total order on the buckets: 2, 1, 4, 3, 6, 5, 8, 7, 9, 10. The idea is that a program that is more saturated better 'fits' a specification, so for example, bucket (2) appears before bucket (1), though we note the bottom row of Table 7.15 favors single-path

Table 7.14: Average Relevance per Rank for Search Approaches

	Rank									
	1	2	3	4	5	6	7	8	9	10
Google	0.792	0.792	0.708	0.667	0.583	0.792	0.583	0.542	0.708	**0.583**
Merobase	0.333	0.458	0.417	0.417	0.333	0.333	0.542	0.250	0.375	0.292
Satsy	0.708	0.458	0.583	0.375	0.375	0.625	**0.625**	0.458	0.542	**0.583**

Table 7.15: Path Matching Ranking Buckets

	$\|LS_{sat}\| = 1$	$1 < \|LS_{sat}\| < \|LS\|$	$LS_{sat} = LS$
$Q_{sat} \subset Q_P$	(10)	(7) $\|LS_{sat}\| \leq \|Q_{sat}\|$	(3) $\|LS\| \leq \|Q_{sat}\|$
		(8) $\|LS_{sat}\| > \|Q_{sat}\|$	(4) $\|LS\| > \|Q_{sat}\|$
$Q_{sat} = Q_P$	(9)	(5) $\|LS_{sat}\| \leq \|Q_P\|$	(1) $\|LS\| \leq \|Q_P\|$
		(6) $\|LS_{sat}\| > \|Q_P\|$	(2) $\|LS\| > \|Q_P\|$

programs. This round-robin approach was intended to facilitate a post-analysis on the relevance among the buckets.

In the study, we used 3 query generators to write queries for each of 8 problems. For each query, Table 7.16 shows the total results from the repository, classified by the buckets in Table 7.15, considering a tight matching on the type signatures ($TS_q = TS_{LS}$ for all matches, as described in Section 5.3).

On average across all queries, 431 snippets were returned. For seven of the 24 queries, not enough results were returned to fill up the 10 spots required for the analysis. Thus, another search was conducted with a relaxed type signature matching, with $TS_q \supseteq TS_{LS}$. Those results are shown in Table 7.17. After attempting to fill the top 10 slots with the round-robin approach on the results from the more constrained search, the same approach was taken with the relaxed search. The lowest number of matches came from the first human generator assigned to problem 21. The constrained search returned zero matches in the constrained search and only 12 matches were returned in the relaxed search. On the other hand, the third query used for problem 8 return 1,772 results in the constrained, and 2,067 results in the relaxed searches.

For each of the 24 queries, we selected 10 results among the buckets. This yields 240 total snippets from Satsy that were evaluated by Mechanical Turk participants. Table 7.18 shows how these selected snippets fell among the buckets, along with the average relevance among the snippets in each bucket. If a relaxed search was required

Table 7.16: Results From Repository by Bucket

P	QG	1	2	3	4	5	6	7	8	9	10	Sum
5	$x_{1,s,5}$	1	0	1	0	0	0	0	0	0	0	2
	$x_{2,s,5}$	0	154	0	0	2	18	0	1	82	5	262
	$x_{3,s,5}$	0	152	0	0	0	7	0	1	93	6	259
	$\bar{x}_{s,5}$	0.50	77.00	1.00	1.00	1.75	7.75	1.75	2.50	46.00	5.30	144.50
6	$x_{1,s,6}$	0	0	0	0	0	13	0	0	1672	53	1738
	$x_{2,s,6}$	0	13	0	0	0	0	0	0	1687	66	1766
	$x_{3,s,6}$	0	0	0	0	0	0	0	0	14	0	14
	$\bar{x}_{s,6}$	0.00	4.33	0.00	0.00	0.00	4.33	0.00	0.00	1124.33	39.67	1172.67
7	$x_{1,s,7}$	2	145	7	0	0	0	0	0	291	75	520
	$x_{2,s,7}$	0	103	0	0	0	217	4	61	138	14	537
	$x_{3,s,7}$	0	5	0	0	4	445	39	29	19	9	550
	$\bar{x}_{s,7}$	0.67	84.33	2.33	0.00	1.33	220.67	14.33	30.00	149.33	32.67	535.67
8	$x_{1,s,8}$	0	1	0	6	0	0	0	0	1	0	8
	$x_{2,s,8}$	0	5	0	0	0	1	0	1	1689	61	1757
	$x_{3,s,8}$	0	0	0	0	5	0	1	0	1696	70	1772
	$\bar{x}_{s,8}$	0.00	2.00	0.00	2.00	1.67	0.33	0.33	0.33	1128.67	43.67	1179.00
11	$x_{1,s,11}$	0	3	0	2	0	0	0	0	0	0	5
	$x_{2,s,11}$	1	0	0	0	0	0	0	0	15	13	29
	$x_{3,s,11}$	0	1	0	0	0	3	0	1	15	12	32
	$\bar{x}_{s,11}$	0.33	1.33	0	0.67	0.00	1.00	0.00	0.33	10.00	8.33	22.00
13	$x_{1,s,13}$	0	7	0	1	0	0	0	0	0	0	8
	$x_{2,s,13}$	0	7	0	1	0	0	0	0	0	0	8
	$x_{3,s,13}$	7	0	1	0	0	0	0	0	0	0	8
	$\bar{x}_{s,13}$	2.33	4.67	0.33	0.67	0.00	0.00	0.00	0.00	0.00	0.00	8.00
20	$x_{1,s,20}$	0	0	0	0	0	1	1	0	13	27	42
	$x_{2,s,20}$	0	0	0	1	0	11	0	27	3	0	42
	$x_{3,s,20}$	0	0	0	1	0	11	0	25	3	2	42
	$\bar{x}_{s,20}$	0.00	0.00	0.00	0.67	0.00	7.67	0.33	17.33	6.33	9.67	42.00
21	$x_{1,s,21}$	0	0	0	0	0	0	0	0	0	0	0
	$x_{2,s,21}$	0	1	0	0	2	264	0	4	0	0	271
	$x_{3,s,21}$	0	1	1	0	0	0	0	0	253	7	262
	$\bar{x}_{s,21}$	0.00	0.67	0.33	0	0.67	88.00	0.00	1.33	84.33	2.33	177.67
	$\bar{x}_{s\cdot}$	0.46	25.00	0.50	0.67	0.75	41.54	2.17	6.58	320.54	17.92	431.83

(Section 7.2.2.4), we indicate this with an 'r' in the bucket name, and compute the relevance separately. A total of 40 snippets were collected from relaxed searches, and these are split among relaxed buckets (1r), (2r), (3r), and (4r). Consider, for example, bucket (9). Of the 240 snippets presented to Mechanical Turk participants, 57 would be classified under bucket (9) level matching. The average relevance among these individual snippets was 0.456.

Referring back to Table 7.11, the overall P@10 for Satsy was 0.533. As shown in Table 7.18, there are six buckets that have a higher average relevance than the P@10, and these are, in order from highest to lowest relevance, (4), (2), (4r), (6), (5), and

Table 7.17: Results From Repository by Bucket, Relaxed

P	QG	1r	2r	3r	4r	5r	6r	7r	8r	9r	10r	Sum
5	$x_{1,s,5}$	15	0	2	0	0	0	0	0	0	0	17
	$x_{2,s,5}$	0	168	0	0	2	23	0	1	84	5	283
	$x_{3,s,5}$	0	168	0	0	0	12	0	1	95	7	283
	$\bar{x}_{s,5}$	5.00	112.00	0.67	0.00	0.67	11.67	0.00	0.67	59.67	4.00	194.33
6	$x_{1,s,6}$	2	185	0	2	0	16	0	0	1768	59	2032
	$x_{2,s,6}$	3	198	0	1	0	2	0	0	1782	73	2059
	$x_{3,s,6}$	1	182	0	3	0	3	0	0	39	1	229
	$\bar{x}_{s,6}$	2.00	188.33	0.00	2.00	0.00	7.00	0.00	0.00	1196.33	44.33	1440.00
7	$x_{1,s,7}$	52	294	11	0	0	0	0	0	337	71	765
	$x_{2,s,7}$	4	298	0	4	8	255	5	62	138	8	782
	$x_{3,s,7}$	4	200	0	5	10	481	41	33	19	9	802
	$\bar{x}_{s,7}$	20.00	264.00	3.67	3.00	6.00	245.33	15.33	31.67	164.67	29.33	783.00
8	$x_{1,s,8}$	8	198	0	12	0	0	0	0	3	0	221
	$x_{2,s,8}$	3	208	0	3	0	3	0	1	1767	66	2051
	$x_{3,s,8}$	3	175	1	1	7	36	3	0	1768	73	2067
	$\bar{x}_{s,8}$	4.67	193.67	0.33	5.33	2.33	13.00	1.00	0.33	1179.33	46.33	1446.33
11	$x_{1,s,11}$	8	201	0	8	0	0	0	0	1	0	218
	$x_{2,s,11}$	10	168	1	2	0	0	0	0	56	17	254
	$x_{3,s,11}$	1	177	0	3	0	34	0	3	27	14	259
	$\bar{x}_{s,11}$	6.33	182.00	0.33	4.33	0.00	11.33	0.00	1.00	28.00	10.33	243.67
13	$x_{1,s,13}$	0	42	0	2	0	0	0	0	0	0	44
	$x_{2,s,13}$	0	42	0	2	0	0	0	0	0	0	44
	$x_{3,s,13}$	42	0	2	0	0	0	0	0	0	0	44
	$\bar{x}_{s,13}$	14.00	28.00	0.67	1.33	0.00	0.00	0.00	0.00	0.00	0.00	44.00
20	$x_{1,s,20}$	0	13	0	1	0	1	1	0	13	27	56
	$x_{2,s,20}$	0	13	0	2	0	11	0	27	3	0	56
	$x_{3,s,20}$	0	13	0	2	0	11	1	24	3	2	56
	$\bar{x}_{s,20}$	0.00	13.00	0	1.67	.000	7.67	0.67	17.00	6.33	9.67	56.00
21	$x_{1,s,21}$	0	12	0	0	0	0	0	0	0	0	12
	$x_{2,s,21}$	0	15	0	0	2	271	0	4	0	0	292
	$x_{3,s,21}$	0	15	1	0	0	0	0	0	260	7	283
	$\bar{x}_{s,21}$	0.00	14.00	0.33	0.00	0.67	90.33	0.00	1.33	86.67	2.33	195.67
	$\bar{x}_{s\cdot}$	6.13	129.78	0.69	2.30	1.26	50.39	2.22	6.78	354.91	19.09	573.57

(1) (in fact, the first three are higher than the overall relevance for Google at 0.675). When it comes to matching the entire specification (last column), it would seem that saturation is preferred where some $ls \in LS$ actually cover the same path in P; this observation follows from the fact that the relevance for bucket (2) is greater than for (1), though this might be a function of sample size.

We conjecture that if Satsy had access to the information presented in Table 7.18, we could have increased the P@10 by using more programs from some of the more relevant buckets. To test this hypothesis, we re-computed the top 10 results from the Satsy queries using a ranking algorithm based on Table 7.18. Instead of round-robin,

Table 7.18: Path Matching Bucket Counts and Relevance

	$\|LS_{sat}\| = 1$	$1 < \|LS_{sat}\| < \|LS\|$	$LS_{sat} = LS$
Q_{sat} $\subset Q_P$	(10) - 34 - 0.500	(7) - 4 - 0.500	(3) - 0 - 0.000, (3r) - 3 - 0.333
		(8) - 18 - 0.444	(4) - 5 - 1.000 , (4r) - 3 - 0.667
Q_{sat} $= Q_P$	(9) - 57 - 0.456	(5) - 9 - 0.556	(1) - 9 - 0.556, (1r) - 9 - 0.444
		(6) - 28 - 0.571	(2) - 36 - 0.889, (2r) - 25 - 0.200

however, we used a greedy approach and grabbed as many results as possible from each bucket, starting from the most relevant. That is, for each query, we grabbed all results from bucket (4), followed by bucket (2), then (4r), and so forth until 10 results were obtained. Next, we computed the *hypothetical* relevance of each result based on its bucket. For example, if a result came from bucket (2), we assigned it a relevance of 0.889. A result from bucket (2r) was assigned a relevance of 0.200. Taking the average of these relevances forms a hypothetical P@10.

We show the hypothetical changes in relevance in Table 7.19, alongside the response variables for Satsy from Table 7.11. The new column, *Hypothetical Satsy*, shows the hypothetical P@10 values for each of the queries. In most cases, the hypothetical P@10 was higher than the P@10 for the study, yet for seven of the 24 queries, the original search results had a higher P@10. This was particularly noticeable for Problem 13, which is the only problem for which the relevance for the problem decreased with the new ranking, from $\bar{x}_{s,13} = 0.833$ to $\bar{x'}_{s,13} = 0.660$. This was an interesting case, since many of the results came from Bucket (1), but this was the only problem for which Mechanical Turk participants rated snippets from Bucket (1) as relevant; for every other query with responses from Bucket (1), those were marked as irrelevant. Thus, the average relevance for Bucket (1) was relatively low at 0.556, which brought down the average relevance for this problem.

Table 7.19: Hypothetical P@10 Response Matrix for Satsy Under New Ranking

Problem	Search Approach	
	Satsy	Hypothetical Satsy
5	$x_{1,s,5} = 0.200$	$x'_{1,s,5} = 0.456$
	$x_{2,s,5} = 0.600$	$x'_{2,s,5} = 0.889$
	$x_{3,s,5} = 0.800$	$x'_{3,s,5} = 0.889$
	$\bar{x}_{s,5} = 0.533$	$\bar{x'}_{s,5} = 0.744$
6	$x_{1,s,6} = 0.400$	$x'_{1,s,6} = 0.590$
	$x_{2,s,6} = 0.500$	$x'_{2,s,6} = 0.889$
	$x_{3,s,6} = 1.000$	$x'_{3,s,6} = 0.550$
	$\bar{x}_{s,6} = 0.633$	$\bar{x'}_{s,6} = 0.676$
7	$x_{1,s,7} = 0.300$	$x'_{1,s,7} = 0.889$
	$x_{2,s,7} = 0.500$	$x'_{2,s,7} = 0.889$
	$x_{3,s,7} = 0.400$	$x'_{3,s,7} = 0.778$
	$\bar{x}_{s,7} = 0.400$	$\bar{x'}_{s,7} = 0.852$
8	$x_{1,s,8} = 0.800$	$x'_{1,s,8} = 0.889$
	$x_{2,s,8} = 0.600$	$x'_{2,s,8} = 0.713$
	$x_{3,s,8} = 0.600$	$x'_{3,s,8} = 0.544$
	$\bar{x}_{s,8} = 0.667$	$\bar{x'}_{s,8} = 0.715$
11	$x_{1,s,11} = 0.400$	$x'_{1,s,11} = 0.767$
	$x_{2,s,11} = 0.100$	$x'_{2,s,11} = 0.498$
	$x_{3,s,11} = 0.900$	$x'_{3,s,11} = 0.571$
	$\bar{x}_{s,11} = 0.467$	$\bar{x'}_{s,11} = 0.612$
13	$x_{1,s,13} = 0.900$	$x'_{1,s,13} = 0.729$
	$x_{2,s,13} = 0.800$	$x'_{2,s,13} = 0.729$
	$x_{3,s,13} = 0.800$	$x'_{3,s,13} = 0.522$
	$\bar{x}_{s,13} = 0.833$	$\bar{x'}_{s,13} = 0.660$
20	$x_{1,s,20} = 0.400$	$x'_{1,s,20} = 0.524$
	$x_{2,s,20} = 0.400$	$x'_{2,s,20} = 0.581$
	$x_{3,s,20} = 0.600$	$x'_{3,s,20} = 0.581$
	$\bar{x}_{s,20} = 0.467$	$\bar{x'}_{s,20} = 0.562$
21	$x_{1,s,21} = 0.100$	$x'_{1,s,21} = 0.200$
	$x_{2,s,21} = 0.300$	$x'_{2,s,21} = 0.571$
	$x_{3,s,21} = 0.400$	$x'_{3,s,21} = 0.530$
	$\bar{x}_{s,21} = 0.267$	$\bar{x'}_{s,21} = 0.434$
	$\bar{x}_{s\cdot} = 0.533$	$\bar{x'}_{s\cdot} = 0.657$

Overall, had the ranking approach changed based on Table 7.18, and the participant responses remained the same (according to bucket), the overall $P@10$ for Satsy would increase from 0.533 to 0.656, which is nearly as high as the $P@10$ for Google. Thus, *with smarter ranking, our search approach would likely be very competitive with Google.*

7.2.5.4 On the Influence of Repository Size on Results

One reason Satsy may not have outperformed Google is the limited size of our repository. To evaluate this conjecture, we manipulated the size of our repository artificially and observed the impact on the number of matches that are found for search queries.

We conducted 24 searches, corresponding to the 24 queries used on Satsy. The independent variable was the size of the repository being used in the search, as a percentage of the full Java repository. The dependent variable is the percentage of matches Satsy finds in the reduced repository over the total matches in the full repository, for each query.

We conducted this experiment with a step size of 0.01, hence exploring 100 different repository sizes. For each query and repository size, we computed a new repository five times, and took the average of the matches found in each repository. The results are shown in Figure 7.8.

Each query is represented as a line, averaged over the five replications. Using the averages across all the queries, we obtain a linear model, $y = ax + b$, with a slope coefficient of $a = 1.0043$ and an intercept of $b = -0.0027$.

It is not surprising to find that the number of matches grows proportionally to the repository size, given the small size of the repository. There are two reasons this might be true. The first is that our repository has not begun to saturate the problem

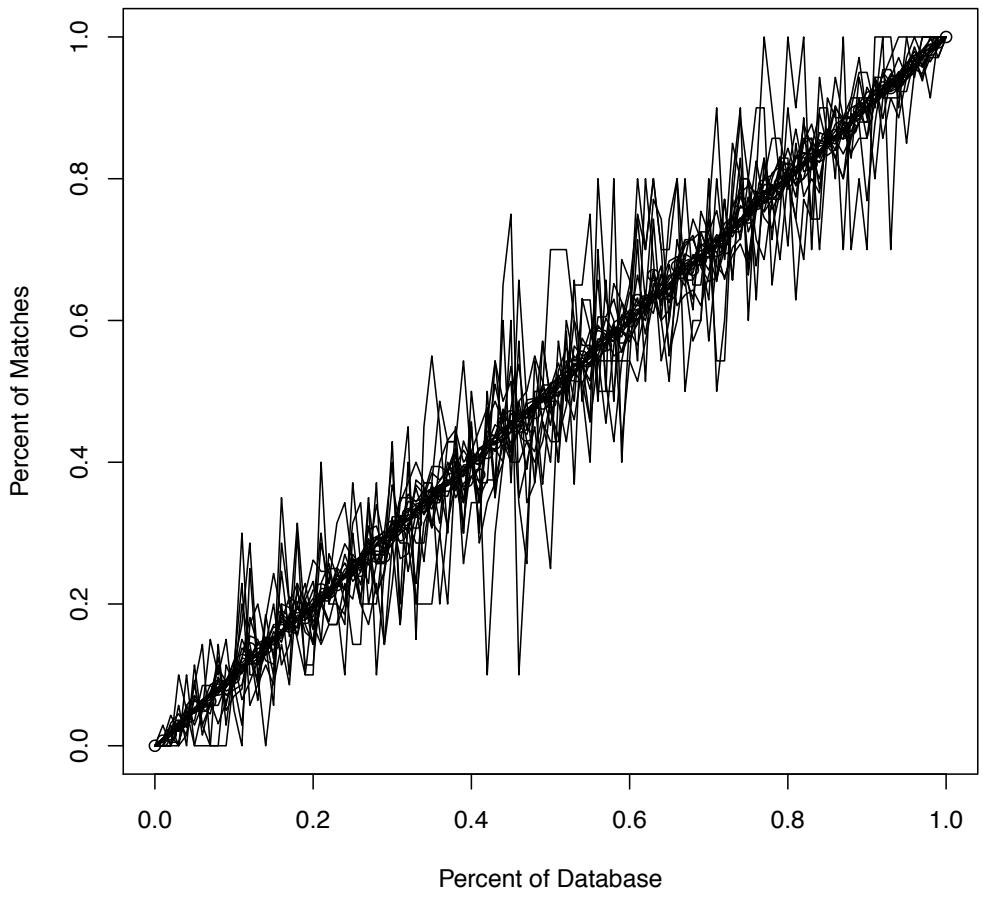


Figure 7.8: Impact of Changes in Repository Size on Matches, Step size = 0.01, Reps=5

domain, but the other is that the problems are so basic, or our matching criteria so loose, that there will always be source code that can be identified as a result.

What this analysis does not capture is the diversity of the matches. That is, we could have two results with the exact same bytecodes, but different naming conventions, and these are marked as two separate matches. If we first cluster the programs based on bytecodes and only report one match per cluster, our conjecture is that the graph would flatten out quickly since "duplicates" would be ignored. Further study is needed here.

7.3 Summary

We have presented the design and results of two studies comparing the relevance of results from our new search tool, Satsy, to those found with Google, Google with a constrained repository, and with Merobase.

In *RQ2(a)*, the query generator was simply a question posted on `stackoverflow.com`, and presumably a question for which the question asker could not find a response online. However, the title of the question may not be representative of the query that would be issued to find this information; this threat to external validity is discussed further in Section 9.2.2.

The study toward *RQ2(c)* was designed to address some of the threats to validity of the study for *RQ2(a)* by, for example, including human generators, using a richer repository, and using regular Google. However, the threat that the query generator is separated from the query evaluator remains.

Toward *RQ2(c)*, while Google outperformed the others in terms of the collected metrics, we have reason to believe that refinement of the ranking algorithm used in Satsy and increasing the language coverage for Java would make Satsy more competitive with Google. Of course, we cannot ignore the fact that the query model necessary for Satsy is drastically different from that for Google. This needs to be studied and explored further to evaluate the impact Satsy could have on state-of-the-practice code searches.

Chapter 8

Yahoo! Pipes Evaluation

In the evaluation of this approach, we have designed a series of studies to assess and highlight some key aspects of the approach through the two implemented domains: Java and Yahoo! Pipes. This chapter focuses on the Yahoo! Pipes implementation in Satsy, and aims to address two general research objectives. *RQ2* was also the focus of Chapter 7. *RQ3* is new for this domain:

RQ2: How effective is Satsy at returning relevant source code?

- **RQ2(d)** What is the impact of varying the abstraction level for the encoded programs in the repository?
- **RQ2(e)** What is the impact of solver time on the effectiveness of the search?

RQ3: What is the cost of defining a query in terms of input/output for the programmer?

- **RQ3(a)** How *accurately* can programmers compose input/output specifications?
- **RQ3(b)** How *efficiently* can programmers compose input/output specifications?

Every search conducted in Satsy toward *RQ2*[1] uses an input/output query. We measure the *effectiveness* of a search in terms of code semantics, analyzing the returned source code and determining if it would behave as specified by the input/output example, and reporting the number of results, precision, and recall, where the baseline is our search at the most abstract encoding level. This is different from the Java studies (Section 7) because we are not comparing against another search again, but rather are comparing against Satsy with the most flexible program encoding.

The Yahoo! Pipes domain is better suited than Java to evaluate *RQ2(d)* and *RQ2(e)* for two reasons. First, in our current Java evaluation, the specifications and snippets are small, so the solver times are fast. In Yahoo! Pipes, the specifications can get quite large, and solving can take minutes. Second, the Java code snippets we encode are taken out of context, so many variables are already symbolic. The Yahoo! Pipes programs, on the other hand, are encoded in their entirety, which leaves much more opportunity for abstraction.

In *RQ3*, we measure and assess the correctness and efficiency of programmers writing Yahoo! Pipes and SQL specifications as part of an empirical user study. We included SQL as part of this study as it is a domain we plan to target more extensively with the future version of Satsy. It also provides an additional domain against which the results from Yahoo! Pipes can be compared.

For each research question, we describe how the repositories were built, the metrics we use, and the results. All of the study artifacts are available online.[2] Next, we describe the experimental contexts for these studies.

[1] RQ1 was introduced and discussed in Section 3.1

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

8.1 Yahoo! Pipes Study 1

In Yahoo! Pipes, the specifications can be quite large and complex, and tweaking the search parameters can have a profound impact on the results. The goal of this study is to analyze the impact of tweaking search parameters, specifically abstraction and solver time, for the purpose of evaluation with respect to effectiveness in terms of precision and recall from the point of view of the researcher in the context of searching a controlled, local repository of Yahoo! Pipes programs using input/output queries [67, 69, 70]. In this study, we compare and contract different configurations of Satsy considering levels of abstraction and solver time. Toward this goal and to refine *RQ2*, we aim to address *RQ2(d)* and *RQ2(e)*.

8.1.1 Artifacts

To evaluate *RQ2*, we require a repository of Yahoo! Pipes programs and specifications to search the repository. In a previous study with Yahoo! Pipes [71] we scraped 32,887 pipes programs from the public repository by issuing approximately 50 queries against the repository and removing all duplicates. Among these pipes, 2,859 are supported by our encoding (Section 6.4), which forms the local repository.

To perform the searches for the study, we gathered specifications from five representative pipes in the repository. These are intended to be "typical" pipes in the repository based on their 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 first 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 154 clusters. The pipes within the clusters were then refactored[3]. Clusters with pipes that

[3]Refactorings focused on decreasing the size of the pipes and standardizing them according to the community standards [68].

have identical structures post-refactoring were merged. Then, clusters that had fewer than five pipes were removed, as were the most and the least popular clusters. This was done to identify pipes that are "typical" yet diverse. From each of the remaining five clusters, we randomly selected one pipe, forming the five example pipes used for evaluation. Each specification was obtained by retrieving the RSS feeds from the URLs in the *fetch feed* modules to form the input list(s) and executing the pipe to capture the output list. Next, we describe the derived specifications for each.

Pipe 1 has two URLs and gathers weather information that has "10-Day" or "Current" in the title. The URLs for this pipe, and for the other pipes, are shown in Table 3.5. For Pipe 1, only one URL is listed, yet in the specification there are two URLs. Since both URLs have the same subdomain and domain names, for the purposes of the search results in Table 3.5, only one needed to be listed. The specification, then, has two URLs in the input, and the output is obtained by executing the pipe. Table 8.1 shows, for each of the pipes, its specification where each x_i represents an RSS item like the one in Figure 2.2. For Pipe 1, there are two input lists, each with five items. The output list has two items with one from each of the input lists.

Pipe 2 has two URLs and looks for hotel information, retaining RSS items that contain "hotel" in the description field, sorting the list by publication date. As with pipe 1, the URLs are similar so only one is listed in Table 3.5. The output has three items, as shown in Table 8.1. Pipe 3 has three URLs and retains up to the first three items from each feed. Thus, the output has nine items. Pipe 4 has one URL that grabs the third item in the list, so the output has one item. Finally, pipe 5 looks for items from one URL that have "au" in the description; the output has seven items. Note that each input list is limited in size to five items to control the sizes of the specification and thus the solver time.

Table 8.1: Specifications for Example Pipes

$LS_1 = (\{[x_0, x_1, x_2, x_3, x_4], [x_5, x_6, x_7, x_8, x_9, x_{10}]\}, \{[x_0, x_5]\})$
$LS_2 = (\{[x_0, x_1, x_2, x_3, x_4], [x_5, x_6, x_7, x_8, x_9, x_{10}]\}, \{[x_9, x_1, x_2]\})$
$LS_3 = (\{[x_0, x_1, x_2, x_3, x_4], [x_5, x_6, x_7, x_8, x_9, x_{10}], [x_{11}, x_{12}, x_{13}, x_{14}, x_{15}]\},$
$\{[x_0, x_1, x_2, x_5, x_6, x_7, x_{10}, x_{11}, x_{12}]\})$
$LS_4 = (\{[x_0, x_1, x_2, x_3, x_4]\}, \{[x_2]\})$
$LS_5 = (\{[x_0, x_1, x_2, x_3, x_4], [x_5, x_6, x_7, x_8, x_9, x_{10}]\}, \{[x_1, x_5, x_6, x_2, x_7, x_8, x_9]\})$

8.1.2 Metrics

To address *RQ2(d)* and *RQ2(e)*, we manipulate two search parameters, the abstraction of the program encodings (*RQ2(d)*) and the maximum solver time (*RQ2(e)*). These are shown as parameters to Satsy in Figure 6.1. We report the number of pipes in the local repository that return *sat*, *unsat*, and *unknown* (?) at each of four solver times, 1, 10, 100, and 1,000 seconds (*sec.*), considering two levels of abstraction on the program encoding as described in Section 5.2.3.2. These levels are all concrete and all symbolic, corresponding to **BCIS** and **BCIS** (symbolic integers and strings), respectively. As described in Section 6.4.2.3, our implementation in Yahoo! Pipes supports symbolic values on integers and strings only. These search parameters can be manipulated only using Satsy, so we cannot directly compare Satsy's results to a syntactic search on the local repository.

We also calculate precision and recall, where $precision = \frac{relevant \cap sat}{sat}$ and $recall = \frac{relevant \cap sat}{relevant}$, using the results of our own search as a baseline. That is, the relevant results are defined as those that will eventually (given infinite time) return *sat* with a symbolic encoding, which represents the pipes for which some instantiation of the module field values can achieve the desired behavior. The precision of Satsy will always be 1.00 as we protect against spurious results by design and base the set of relevant results on the symbolic encodings, which is a superset of the results returned for a concrete encoding. The recall value does vary based on the parameter settings.

We chose precision and recall as metrics since we were able to characterize repository in this domain in terms of what is relevant. In the Java studies, our goal was to compare against existing searches so $P@10$ was appropriate since we cannot calculate precision and recall for search engines we do not control, such as Google. Here we are interested in the impact of various Satsy configurations.

8.1.3 Implementation

This evaluation was an artifact-only evaluation and required no human input, outside of artifact collection. The process was as follows. First, we encoded each pipe specification. Second, for a given abstraction level and maximum solver time, we searched the local repository for matches and collected the metrics. Third, we calculated precision and recall. The oracle of *relevant* programs was created by executing each specification with a maximum solver time of 1,000 seconds on a repository encoded at the **BCIS** level of abstraction. For those (very few) programs that did not return, they were checked by hand for relevance given the specification. Our data were collected under Linux on 2.4GHz Opteron 250s with 16GB of RAM.

8.1.4 Results

RQ2(d) and *RQ2(e)* aim to explore the impact of solver time and the abstraction level of program encodings on the precision and recall of the search. Satsy searches the repository using the five pipe specifications, given the solver times and abstraction levels described, and calculates precision and recall.

For each combination of solver time and abstraction, we report the results for each specification in Table 8.2. The first set of columns reports the results for the *Concrete* abstraction level, and the second set for the *Symbolic* abstraction level. Each row

Table 8.2: Pipe Specification Search Results

Specification 1

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

Specification 2

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

Specification 3

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

Specification 4

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

Specification 5

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

represents the results given a specific maximum solver time. The number of matches is shown in the *Sat* column, and the number of discarded programs is shown in the *Unsat* column. If the solver was stopped before it could make a decision, then the solver returned *unknown*, which is shown in the *?* column, followed by the recall metric.

Matches from the repository can come from pipes with various topologies. For example, with *Specification 1* and the concrete encoding at 1000 seconds, there are 17 matches. Of those, 13 matches permit records that contain the substring `http://` in the description, which happens to be the case for all records in o. Three others simply grab the first element from each input list, and the last result is P_1 itself.

Overall, we observe that for all searches and abstraction levels, at least one match is found with the maximum solver time of 1000 seconds, which is fitting as each specification was derived from a pipe in our local repository. Additionally, using symbolic encodings usually yields more results than concrete. Next, we look into the impact of abstraction level and solver time on recall.

8.1.4.1 RQ2(d): Impact of Varying Abstraction Levels

We now discuss the results in the context of *RQ2(d): What is the impact of varying the abstraction level for the encoded programs in the repository?*. At any given level of maximum solver time, using symbolic encodings usually yields more results than concrete. For instance, with *Specification 4* in Table 8.2 at 1000 seconds, all the programs have been determined to be *sat* or *unsat* for the concrete and symbolic encodings (*?* $= 0$ for both). Yet, the symbolic encoding yields 89 possible matches while the concrete encoding only finds one, since there is only one pipe in the repository that will return *sat* with the concrete encoding. With the symbolic encoding, the title string in the *filter* modules are symbolic so pipes that return all records with a

particular title return *sat* by the solver. Thus, the symbolic encodings may produce more results and more quickly, but possibly at the cost of additional adaptation.

8.1.4.2 RQ2(e): Impact of Solver Time on Search Effectiveness

We now discuss the results in the context of *RQ2(e): What is the impact of solver time on the effectiveness of the search?*. For all searches and abstraction levels, at least one match is found with the maximum solver time of 1000 seconds, though it does take some time for results to be found. For *Specification 2* and *Specification 3*, the first result is returned when the solver time is set to 1000, but not before. Yet, for the other three specifications, a result is returned in the concrete encoding within 100 seconds, and in the case of *Specification 4*, within 10 seconds. More results can be found with the symbolic encoding, but it takes about the same amount of time for matches to be returned.

There is a time advantage of using a concrete encoding, as it can discard programs that do not match faster than the symbolic encodings, in part because the constraint system is tighter and there are fewer decisions the solver has to make. For all examples, and all ranges of solver times, the number of *unsat* programs for the concrete encoding is always greater than or equal to the number of *unsat* programs for its symbolic counterpart. This makes sense for the longer solver times since the symbolic encoding has the potential to recognize more matches, yet, the trend is also true for the smaller solver times.

8.2 Yahoo! Pipes Study 2

The goal of this study is the analyze the use of input/output queries for search in Yahoo! Pipes and SQL for the purpose of evaluation with respect to accuracy and time

from the point of view of the researcher in the context of programmers determining the expected output when given a problem description and input. Toward this goal and *RQ3*, we aim to address *RQ3(a)* and *RQ3(b)*.

We designed a user study around ten tasks that, given a behavioral description of a desired functionality and an input, ask the participants to select the output. The experimental tasks were split among the two targeted domains, Yahoo! Pipes and SQL, with five tasks in each.

Note that the participants specify only the output, yet we aim to evaluate the input/output specification model as a whole. The rationale for this choice is twofold. First, for the domains being evaluated, the inputs can easily be obtained by the programmer naturally from their context (from a target URL in the case of a pipe, or from a database table in the case of SQL). So, having the participants start with an existing input but with the challenge of computing an output mimics how a semantic search with an input/output model would be implemented in practice for these domains. Second, given a problem description, by fixing the input we can automate the accuracy assessment since we know the expected output. This reduces the cost of executing the experiment and the chances of accuracy measurement errors. We discuss the experimental aspects related to this point, including the threat to construct validity, in Section 9.3.

8.2.1 Design

We designed an experiment according to the experimental design in Table 8.3. Each of the ten experimental tasks uses one of the targeted domains, L_1 for Yahoo! Pipes, and L_2 for SQL. The assignment of participants to tasks was done using self-selection (SS). In each task, we presented the participant with an input (table or list, denoted

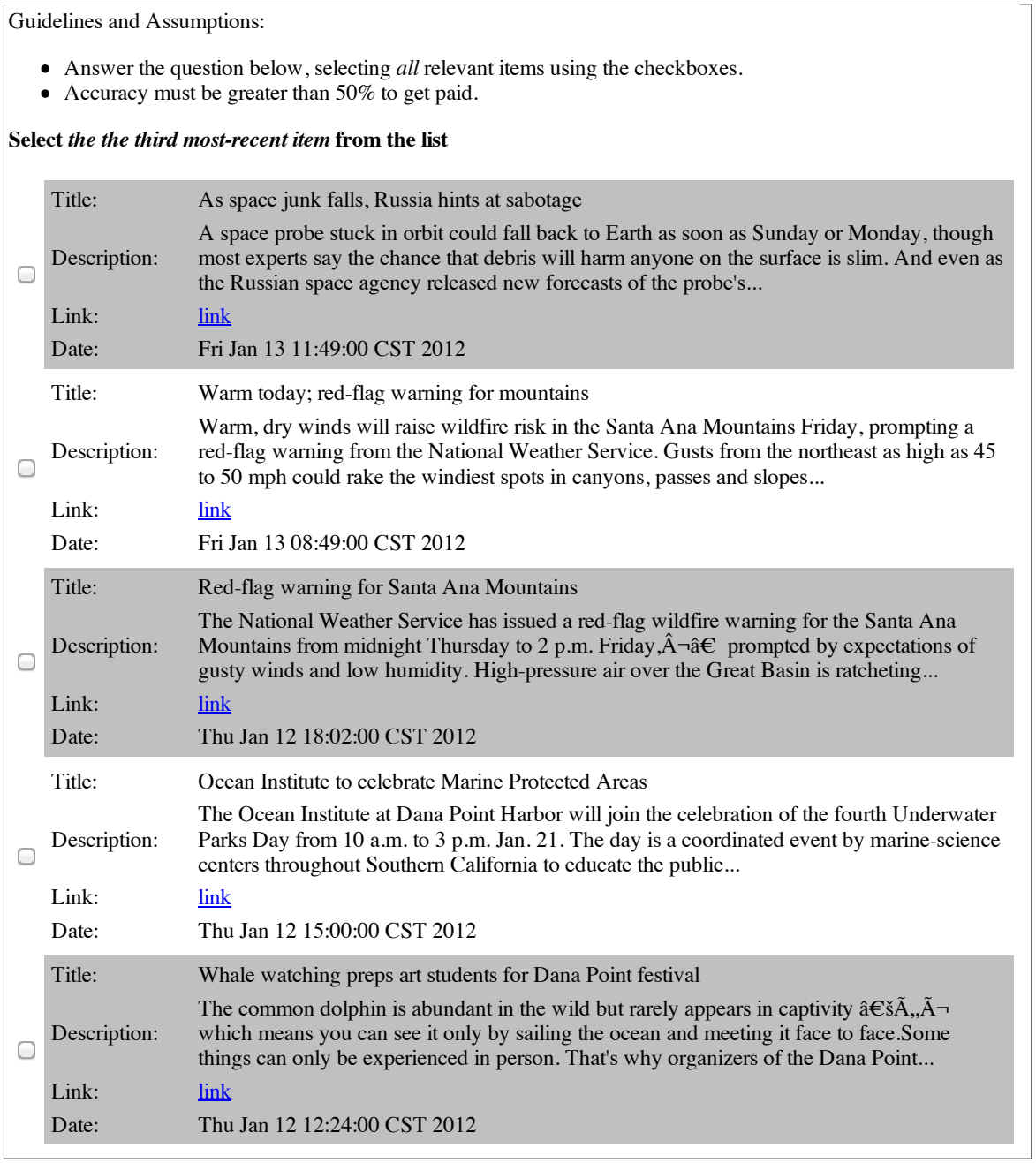
Table 8.3: User Study Design

Task	Domain	Assign	Pretest Measures	Objects	Post test Measures
1	L_1	SS	$O_1 \ldots O_{10}$	I_1 D_1	O_{11}, O_{12}
2	L_1	SS	$O_1 \ldots O_{10}$	I_2 D_2	O_{11}, O_{12}
3	L_1	SS	$O_1 \ldots O_{10}$	I_3 D_3	O_{11}, O_{12}
4	L_1	SS	$O_1 \ldots O_{10}$	I_4 D_4	O_{11}, O_{12}
5	L_1	SS	$O_1 \ldots O_{10}$	I_5 D_5	O_{11}, O_{12}
6	L_2	SS	$O_1 \ldots O_{10}$	I_6 D_6	O_{11}, O_{12}
7	L_2	SS	$O_1 \ldots O_{10}$	I_6 D_7	O_{11}, O_{12}
8	L_2	SS	$O_1 \ldots O_{10}$	I_6 D_8	O_{11}, O_{12}
9	L_2	SS	$O_1 \ldots O_{10}$	I_6 D_9	O_{11}, O_{12}
10	L_2	SS	$O_1 \ldots O_{10}$	I_6 D_{10}	O_{11}, O_{12}

I), and a problem description (D), which serve as the objects. The participant was asked to select the output from the input list considering the problem. Since this study aims to characterize the performance and accuracy of the input/output query model, there are no treatments. The pretest measures are the results of a survey that asks about programming experience and search behavior ($O_1 \ldots O_{10}$), and were used in the analysis reported in Section 3.1. The post test measures were accuracy (O_{11}) and time (O_{12}), and are used to address *RQ2(d)* and *RQ2(e)*.

8.2.1.1 Yahoo! Pipes

A task in Yahoo! Pipes, covering Tasks 1-5, presents the participant with a problem description (D) an input list of RSS items (I). Considering the problem description, the participant is instructed to select what output should be using check-boxes next to each RSS item in the input I. An example task is shown in Figure 8.1, which represents Task 4 in the user study. The problem description is, "Select the third-most-recent item from the list," and the input list consists of five RSS items. The expected output is the third item with a publication date of `Thu Jan 12 18:02:00 CST 2012`; this forms an output list of size one.


Guidelines and Assumptions:

- Answer the question below, selecting *all* relevant items using the checkboxes.
- Accuracy must be greater than 50% to get paid.

Select *the the third most-recent item* from the list

☐	Title:	As space junk falls, Russia hints at sabotage
	Description:	A space probe stuck in orbit could fall back to Earth as soon as Sunday or Monday, though most experts say the chance that debris will harm anyone on the surface is slim. And even as the Russian space agency released new forecasts of the probe's...
	Link:	link
	Date:	Fri Jan 13 11:49:00 CST 2012
☐	Title:	Warm today; red-flag warning for mountains
	Description:	Warm, dry winds will raise wildfire risk in the Santa Ana Mountains Friday, prompting a red-flag warning from the National Weather Service. Gusts from the northeast as high as 45 to 50 mph could rake the windiest spots in canyons, passes and slopes...
	Link:	link
	Date:	Fri Jan 13 08:49:00 CST 2012
☐	Title:	Red-flag warning for Santa Ana Mountains
	Description:	The National Weather Service has issued a red-flag wildfire warning for the Santa Ana Mountains from midnight Thursday to 2 p.m. Friday,Â¬â€ prompted by expectations of gusty winds and low humidity. High-pressure air over the Great Basin is ratcheting...
	Link:	link
	Date:	Thu Jan 12 18:02:00 CST 2012
☐	Title:	Ocean Institute to celebrate Marine Protected Areas
	Description:	The Ocean Institute at Dana Point Harbor will join the celebration of the fourth Underwater Parks Day from 10 a.m. to 3 p.m. Jan. 21. The day is a coordinated event by marine-science centers throughout Southern California to educate the public...
	Link:	link
	Date:	Thu Jan 12 15:00:00 CST 2012
☐	Title:	Whale watching preps art students for Dana Point festival
	Description:	The common dolphin is abundant in the wild but rarely appears in captivity â€šÃ„Ã¬ which means you can see it only by sailing the ocean and meeting it face to face.Some things can only be experienced in person. That's why organizers of the Dana Point...
	Link:	link
	Date:	Thu Jan 12 12:24:00 CST 2012

Figure 8.1: Task 4 in Yahoo! Pipes Study

8.2.1.2 SQL

In SQL the input and output take the form of database table(s). The output table would be the selected column(s) and row(s) from the input table(s), such as the

Input Table:

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

Output Table:

username
rekin76
avcio
shantee

Figure 8.2: SQL Input/Output Specification

example shown in Figure 8.2. The input table has four fields, `id`, `username`, `balance`, and `status`. The output table has one field, `username`, which contains a subset of the records from the input table. An example SQL query that produces the output from the input is:

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

For Tasks 6-10, the participant was presented with a problem description (D) and an example input table (I). For all tasks in SQL, the same table was used as the object (I_6), but the problem descriptions differed. This choice was made to ease the burden on the participants, though it may have led to learning effects. This threat to validity is discussed in Section 9.1.4.

Considering the problem description and the input, participants selected the output using check-boxes next to the relevant rows or columns. An example task is shown in Figure 8.3, which corresponds to Task 6 in the user study. Here, the problem statement is, "Select the rows with a price per unit greater than $1.00 from the table." Selecting rows 3, 5, 6, and 8 is the expected response.

Guidelines and Assumptions:

- Answer the question below, selecting *all* relevant items using the checkboxes.
- Accuracy must be greater than 50% to get paid.

Select **the rows with a price per unit greater than $1.00** from the table below, using the checkboxes next to each row.

	Id	Fruit	Variety	Vendor	Price	Unit
☐	1	Apple	Fuji	John's Produce	$0.67	lb
☐	2	Apple	Jonathan	John's Produce	$0.84	lb
☐	3	Apple	Fuji	Fran's Fruit	$1.23	kg
☐	4	Pear	Green Anjou	John's Produce	$0.98	lb
☐	5	Pear	Bartlett	Pearly Pears	$2.01	kg
☐	6	Grapes	Champagne	Eduardo's Uvas	$5.21	kg
☐	7	Bananas	Plantain	Pato's Plantains	$0.50	lb
☐	8	Melon	Honeydew	Oregon Melon, Inc.	$1.10	lb

Figure 8.3: Task 6 in SQL Study

8.2.2 Task Creation

Each experimental task in Table 8.3 requires three pieces of information: the problem description, the input, and the expected output. When a task is presented to the participant, the problem description and input are shown (these are the objects). The participant then selects the output based on the information presented. This is the response variable. We use the expected output as an oracle against which the participant's response is compared using the metrics (Section 8.2.5).

In order to create realistic experimental tasks, we begin with programs (artifacts) from which inputs, expected outputs, and problem descriptions are created. In this section, we describe how this process works and how we selected the programs used to create the experimental tasks.

8.2.2.1 Object and Output Creation

Object creation involves taking a program, either extracting or constructing an input and an output that matches the behavior, and generating a problem description. Each experimental task is built from a different program. Here, we describe this process for Yahoo! Pipes and for SQL.

Yahoo! Pipes. An input is a list of RSS items. Given a Yahoo! Pipes program, the input is derived by extracting the URLs in the pipes and accessing the RSS feeds. Executing the pipe program generates the output list, which is used as the oracle. The problem description was generated by the researchers to concisely reflect the semantics of the program.

Table 8.4 shows the problem descriptions used in Tasks 1-5. To capture the behavior of the pipes while keeping the input list a reasonable size, since the participants would be evaluating each RSS item, we limited the number of records from each URL to seven, though some RSS feeds returned fewer than seven items. An additional bound of 100 characters was imposed on the string lengths. In some cases, there were items in the output that did not originate from the input list, due to the imposed item restriction. Those items were removed so the output reflected the input.

The input and output lists' sizes, in terms of items, are shown in Table 8.5. The range of input sizes is from five to 21 items, and the range of output sizes is from one to nine items.

SQL. An input in a SQL task is a table. An output is either a subset of that table or a new table that was built from the input. Given a SQL select statement (i.e., the SQL program), the input (I_6) and output are generated so that the tables are distinct

Table 8.4: Cost of Providing Specifications, Pipes and SQL

Task	**Textual Description**
1	Select all records that show the Current Weather Conditions or the 10-Day Forecast for Malibu, Exeter, or Camarillo
2	Select the four most-recent records from the list that contain information about a hotel
3	Select the first three records from each source, where the sources are indicated using different background colors
4	Select the the third most-recent record from the list
5	Select all records with the pink background, and those items from the grey background with "au" in the description
6	Select the rows that match some criteria.
7	Select the rows and a subset of columns that match some criteria.
8	Counts the number of rows that match some criteria.
9	Computes an average for different groups of rows.
10	Identifies distinct values from the table subject to some criteria.

Table 8.5: Input/Output Sizes for Experimental Tasks

	Task									
	1	2	3	4	5	6	7	8	9	10
Input Size	21	21	14	5	11	8	14	5	5	5
Expected Output Size	6	4	9	1	4	4	7	1	1	1

and the output is returned when the SQL program is executed on the input. The output forms the oracle against which the participant responses are compared.

In the study design shown in Table 8.3, the input table is the same for all SQL experimental tasks. The problem descriptions, $D_6 - D_{10}$, were generated by the researchers to reflect the semantics of a given SQL select statement, and were designed to be as concise as possible. These are shown in the *Problem Description* column of Table 8.4 for Tasks 6 - 10.

The sizes of the inputs and expected outputs are shown in Table 8.5. The input table is the same for all tasks, but the input sizes differ. Task 6 asks the participant to only select rows, so there are only 8 checkboxes. Task 7, on the other hand, asks about columns and rows, so 14 checkboxes are available. In Tasks 8 - 10, five possible

outputs are presented to the participant, who selects one or more of the options as the output.

8.2.2.2 Artifact Selection

Each experimental task is derived from on an artifact. Here, we describe the selection process.

Yahoo! Pipes. To create the objects for each task related to L_1, we had to collect five pipes and extract the input/output. These pipes are intended to be "typical" pipes in the repository based on structural uniqueness and their popularity, and so we reuse the five Yahoo! Pipes artifacts from the previous Yahoo! Pipes study (Section 8.1.1).

SQL. In SQL, the artifacts were created since there is no existing reference SQL repository. The creation of these SQL artifacts was based on language coverage, using many of the common constructs in SQL select statements. The first artifact, corresponding to Task 6, is a SQL select statement that chooses all columns with a basic inequality condition:

```
SELECT * FROM table WHERE price > 1.00;
```

The second artifact, corresponding to Task 7, selects a subset of the possible columns and includes a conjunction on the conditions in the `WHERE` clause:

```
SELECT Id, Fruit, Variety FROM table WHERE price >= 0.75 AND price <= 1.25;
```

Some select statements return a table of data, while others, such as those including the `count` construct, return a single value, such as the following artifact that corresponds to Task 8:

```
SELECT count(*) FROM table WHERE Unit = 'lb';
```

The fourth artifact corresponds to Task 9 and combines multiple rows using the `avg` and `group by` components:

```
SELECT avg(Price) FROM table GROUP BY Fruit;
```

The last artifact, corresponding to Task 10, weeds out certain rows based on the distinct values in a particular column:

```
SELECT distinct(Vendor) FROM table WHERE Fruit = 'apple';
```

8.2.3 Participants

The participants in this study were solicited from two populations, undergraduate computer science classes at UNL and workers on Amazon's Mechanical Turk [44]. Recruitment of student participants was done by asking the permission of the faculty instructors of the classes, one in embedded systems and the other in software engineering. Recruitment of Mechanical Turk workers was performed by making the tasks available on their server. From there, participation comes from self-selection.

The first part of the study involved a 10-question survey about the participants' programming habits (pretest measures $O_1 \ldots O_{10}$). The Mechanical Turk workers were paid for their tasks, whereas the student participants were not. Participation was restricted to those who accepted the informed consent. [4]

The counts of participants and gender are shown in Table 3.1. Of the 109 participants, 43 came from junior/senior undergraduate classes at UNL and the rest came from Mechanical Turk. 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.

[4] The tasks, informed consent, and UNL Institutional Review Board approved process can be found in Appendix B.

8.2.4 Implementation

Study participants were presented with a problem description, D, and an input, I, and asked to determine what the output should be. This study delivery was based on the participant group. The students performed the study in the classroom. The Mechanical Turk workers performed the study online.

8.2.4.1 Classroom

Implementing this study in the classroom required the creation of paper packets that delivered the IRB informed consent, pretest survey, and the experimental tasks. Each packet was numbered at random and distributed to a classroom of students. The students had 15 minutes in total to complete as much of the packet as possible. The first pages of the packets contained the informed consent and pretest survey. The experimental tasks followed and the order was the same for all packets, starting with the SQL Tasks 6 - 10, and then the Yahoo! Pipes Tasks 1 - 5.

8.2.4.2 Mechanical Turk

In the Mechanical Turk implementation, the pretest survey was implemented as a qualification exam. In addition to the survey questions, there were two competency questions, one about Yahoo! Pipes and another about SQL, which had to be answered with 50% accuracy. The IRB form was signed electronically. When a qualification test was completed, the results were sent to the researchers for evaluation. A passing score and IRB acceptance allowed the worker access to the experimental tasks.

Each experimental task in the study was implemented as a *human intelligence task*, or HIT. Participants had a maximum of 10 minutes to complete each HIT, and were paid $0.26 if their work was accepted. To prevent participants from 'gaming' the

tasks, 50% accuracy was required for payment (Section 8.2.5 describes the scoring). The study was available for three weeks.

8.2.5 Metrics

Two metrics, time and accuracy, were collected as post test measures. Time was collected by the Mechanical Turk server on a per-task basis. In the classroom implementation, the students worked at their own pace through the packets until the 15 minute mark, so we were unable to collect timing information.

Participants selected the output from each experimental task using checkboxes next to each RSS item (Yahoo! Pipes) or columns and rows (SQL) in the input. Accuracy was measured by scoring the participants' responses against the oracles (Section 8.2.2.1). 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. For example, consider the following score sheet for a list of 5 items, which represents the five RSS items shown in Figure 8.1:

Item	Oracle	Response	Points Awarded
1	0	0	1
2	0	1	0
3	1	1	1
4	0	0	1
5	0	0	1
		Score:	4/5 = 80%

The *Item* column represents a list of items, the *Oracle* column indicates the expected output generated during object creation (Section 8.2.2.1), the *Response* column indicates the participant's response, and the *Points Awarded* column indicates how many points were awarded per item. In the oracle, only item 3 is selected. The

participant selected items 2 and 3. Points were awarded for answering correctly on items 1, 3, 4, and 5, leading to an overall score of 80% (item 2 was mis-selected). In the Mechanical Turk implementation, the participant would have gotten paid for this work since the score is at least 50%.

8.2.6 Results

Here, we present the results of the study with respect to *RQ3: What is the cost of defining a query in terms of input/output for the programmer?*, in the two targeted languages, Yahoo! Pipes and SQL. Specifically, we target two sub-questions of *RQ3*, *RQ3(a)* and *RQ3(b)*.

RQ3(a) How *accurately* can programmers compose input/output specifications?

RQ3(b) How *efficiently* can programmers compose input/output specifications?

The target population of this study is novice programmers as the languages being evaluated are domain-specific, end-user programming languages. However, the participants had more programming experience than the typical end-user programmer (see Section 3.1.1). Further study with actual novice programmers is needed.

8.2.6.1 Yahoo! Pipes

The accuracy and timing information for Yahoo! Pipes Tasks 1 - 5 are summarized in Table 8.6. The total number of participants per task is indicated by the n column, followed by the mean and median accuracy for those tasks aggregated across all participants (*RQ1*). An average of 67 participants performed each Pipes task, with a range from 63 to 72.

RQ3(a): Accuracy. Aggregated across all Yahoo! Pipes tasks, the average accuracy was 92%. Participants performed the best on Task 4, with 96% average accuracy and a median accuracy of 100%. Among the Pipes tasks, Task 4 had the smallest input list and the smallest expected output (Table 8.5). There is a negative correlation between task accuracy and the input size (Spearman's $r = -0.2946$), but not between the task accuracy and the output size (Spearman's $r = -0.0136$).

Table 8.6: Cost of Providing Specifications, Pipes and SQL

Task	**Accuracy**			**Timing** (m:ss)		
	n	**Mean**	**Median**	n_2	**Mean**	**Median**
Pipes 1	63	90%	94%	30	2:30	1:48
Pipes 2	60	90%	90%	24	3:48	2:55
Pipes 3	65	93%	100%	29	1:20	0:46
Pipes 4	72	96%	100%	30	1:14	0:47
Pipes 5	70	95%	100%	30	2:26	2:05
SQL 6	72	99%	100%	30	0:41	0:34
SQL 7	71	88%	93%	30	1:37	1:29
SQL 8	63	86%	100%	21	1:08	0:52
SQL 9	68	94%	100%	26	1:44	1:21
SQL 10	72	92%	100%	30	1:22	1:03

RQ3(b): Efficiency. As mentioned in Section 8.2.4, the timing data is only available for the Mechanical Turk participants, so n_2 in Table 8.6 indicates the number of participants from whom we gathered timing data (*RQ2*). For the Yahoo! Pipes tasks, n_2 ranged from 24 to 30. The average time per task was 2:12, but the range was from 1:14 to 3:48. The timing appears to be dependent, in part, on the size of the input presented to the participant (Table 8.5). The Spearman correlation between the time and the size of the input reveals a strong relationship $r = 0.3565$ (that is not the case for the time and output whose $r = -0.0397$).

To give more context to the timing data of formulating an input/output query, consider that understanding irrelevant pipe matches of similar complexity may take

on the order of 16 minutes [66]. Although full understanding of a pipe is not needed to discard irrelevant matches, the cost of this pruning activity is such that investing in a query formulation that takes a couple of minutes but only returns semantically relevant results seems promising. As we discuss later, a comprehensive study on this input/output query format against other query models will be one of the targets of our future work.

Summary Overall, the most accurate task has the shortest completion time and the least accurate task had the longest completion time. This fits with the observation that there is a negative correlation between the length of time and the accuracy of the tasks. There is also a positive correlation between completion time and the input size. We conjecture that it was easier for the participants to get distracted when the input size is large, which led to longer completion times and lower accuracy. On the other hand, with a small input size, the task could be answered quickly and accurately. This may imply that the input/output query model is more appropriate for tasks that can be characterized through small examples, or perhaps a series of small examples rather than a single large one.

8.2.6.2 SQL

The accuracy and timing information for Tasks 6 - 10 are summarized in Table 8.6. An average of 69 participants completed each of the SQL tasks.

RQ3(a): Accuracy Across all SQL tasks, as with Yahoo! Pipes, the average accuracy was 92% (*RQ1*). SQL participants performed best on Task 6, with an average score of 99% and a median of 100%. The lowest accuracy came from Task 8, with an average of 86% and a median of 100%. Since the input table was the same

for all the SQL tasks, we could not draw conclusions about the relationship between accuracy and input size.

RQ3(b): Efficiency The timing data reveals an average completion time of 1:18. Participants provided the fastest responses on Task 6, with an average of 41 seconds and a median of 34 seconds. Task 9 took the longest, with an average of 1:44. In general, longer times were associated with lower accuracy (Spearman's $r = -0.1970$), just as we observed with Yahoo! Pipes. It likely took the participants longer to guess the answer than if they knew the answer outright.

8.2.6.3 Comparing Results

While the focus or *RQ3* is on the feasibility of the specification model, some interesting results emerged from the study pertaining to the use of two diverse populations, students and Mechanical Turk workers. Since this study involves two different implementations, we wanted to know if there are differences among the accuracy results based on the population.

Analysis We segmented the results based on population and computed the mean for each task, for each domain, and overall. The mean accuracy for the Mechanical Turk population is μ_{mt} and the mean for the student population is μ_s. For each task, each domain, and overall, we performed a Mann-Whitney-Wilcoxon [5] test with the null hypothesis, $H_0 : \mu_{mt} = \mu_s$. The alternative is, $H_a : \mu_{mt} \neq \mu_s$.

Results Table 8.7 presents the results of this comparative analysis. Overall, no difference between the groups in terms of accuracy was observed (*Overall*, $p = 0.2122$).

[5] We tested the normality of the data using the Shapiro-Wilk test with a null hypothesis that the population is normal. The null hypothesis was rejected at $\alpha = 0.05$, and so we used the non-parametric Mann-Whitney-Wilcoxon test.

Table 8.7: Differences in Results Based on Implementation. $H_0 : \mu_{mt} = \mu_s$

Task	**Mechanical Turk**	μ_{mt}	**Students**	μ_s	**p-value**
1	30	0.893	35	0.896	0.7144
2	24	0.924	39	0.841	0.0021***
3	29	0.985	36	0.893	0.0304**
4	30	0.947	42	0.962	0.6220
5	30	0.964	40	0.939	0.7773
YP	143	0.943	192	0.908	0.0095***
6	30	1.000	42	0.991	0.2366
7	30	0.883	41	0.886	0.8778
8	21	0.809	42	0.881	0.1876
9	26	0.908	42	0.967	0.0637*
10	30	0.947	42	0.947	0.5660
SQL	137	0.916	209	0.926	0.4160
Overall	280	0.929	401	0.917	0.2122

$\alpha = 0.1$* $\alpha = 0.05$** $\alpha = 0.01$***

This clearly provides support for the use of large-scale and cost-effective crowdsourcing environments like Mechanical Turk to support the execution of empirical studies that serve at least to complement other studies.

Care needs to be taken, however, as the population gets smaller or the requirements more specific. As we show next, splitting the analysis by domain reveals that the accuracy of the tasks was significantly different for the Yahoo! Pipes domain, but not for the SQL domain.

Yahoo! Pipes: In this domain (YP), we reject the null hypothesis that there is no difference between the populations at $\alpha = 0.01$. We observe that the performance of the students is lower than the performance of the Mechanical Turk participants. Breaking this down further by task, we observe significant differences for Tasks 2 and 3 with $\alpha = 0.01$ and $\alpha = 0.05$, respectively. The other tasks did not reveal significant differences between the groups.

We conjecture that these differences manifest as a result of the delivery mechanisms used in the study. Recall that for the students we used hard-copies of the tasks, while

for the Mechanical Turk participants we had online versions of the tasks. For Task 2, the participants were asked to select the most-recent items. In the student version, the list of items spanned multiple pages and it may have been hard to keep track of the dates. On the web version, the participants had an easier time scrolling through the data and searching for keywords with the browser find function. For Task 3, lists of data in the web format had different background colors, but the students' gray-scale copies did not differentiate the lists so clearly and again they spanned multiple pages. We conjecture that this may have contributed to a lower student performance.

SQL: In this domain (*SQL*), we did not reject the null hypothesis and found that there was no difference between the student results and the Mechanical Turk results. However, the average Mechanical Turk score was lower than the average student score, even though μ_{mt} may be artificially high.

We say it may be artificially high because, to control for participant quality of an unknown population in Mechanical Turk, we rejected HITs and did not consider the results when the accuracy was less than 50%. Among the SQL tasks, 25 HITs were rejected (see Section 9). These were not included in μ_{mt}, and thus it may be artificially high. Among the students, only three student results were below 50% (one from Task 8 and two from Task 10), and these were considered as part of μ_s. Thus, there may be an unobserved significant difference in which the students outperformed the Mechanical Turk workers.

8.3 Summary

We have presented two studies on Yahoo! Pipes targeting different aspects of the search approach. In the first, we manipulated search parameters sent to Satsy and measured

the impact on precision and recall. In the second, we evaluated the feasibility of using input/output specifications in Yahoo! Pipes and SQL.

From evaluating *RQ2(d)*, we found that symbolic encodings may produce more results and more quickly, but possibly at the cost of additional adaptation. From *RQ2(e)*, we learned there is a time advantage of using a concrete encoding, as it can discard programs that don't match faster than the symbolic encodings, in part because the constraint system is tighter and there are fewer decisions the solver has to make. Overall, solver time has a clear impact on recall. Since cutting the solver time before it has reached a conclusion returns *unknown*, the recall is reduced as only the *sat* pipes are returned to the programmer. Treating the *unknown* pipes as results will increase recall to 1.00, but at the cost of precision. Studying this tradeoff is left as future work.

Toward *RQ3*, our study shows that programmers can generate outputs when given input and a program description in the Yahoo! Pipes and SQL domains. Further, this can be done with high accuracy and in a reasonable amount of time.

Chapter 9

Threats to Validity

Our evaluation explores different aspects of our approach in each of the supported languages, and each comes with its own limitations and threats to validity. Throughout the presentation of the studies and results in this thesis, we have identified several threats to validity. In this section, we enumerate and elaborate on those more completely. As each threat references its related research question, we re-state those here for convenience.

RQ1: How and why do programmers search for source code? (Section 3.1)

- **RQ1(a):** How frequently do programmers search for code? (Section 3.1.2)
- **RQ1(b):** Why do programmers search for source code? (Section 3.1.3)
- **RQ1(c):** Which tools to programmers use to search for code? (Section 3.1.4)

RQ2: How effective is Satsy at returning relevant source code? (Chapter 7, Chapter 8)

- **RQ2(a)** How do search results from Satsy compare to those found using Google when searching over the same repository of programs? (Section 7.1.1)

RQ2(b) How much can existing syntactic search approaches be improved by augmenting results with Satsy? (Section 7.1.2)

RQ2(c) How do search results from Satsy compare to those found using Google and those found with the code-specific search engine, Merobase, from the perspective of a programmer? (Section 7.2)

RQ2(d) What is the impact of varying the abstraction level for the encoded programs in the repository? (Section 8.1)

RQ2(e) What is the impact of solver time on the effectiveness of the search? (Section 8.1)

RQ3: What is cost of defining a query in terms of input/output for the programmer? (Section 8.2)

RQ3(a) How *accurately* can programmers compose input/output specifications? (Section 8.2)

RQ3(b) How *efficiently* can programmers compose input/output specifications? (Section 8.2)

9.1 Internal

These threats impact the ability to confirm that independent variables influence the dependent variables.

9.1.1 Selection Bias

For all studies involving Mechanical Turk (*RQ1*, *RQ2(c)*, *RQ3*), the results are subject to self-selection bias by the participants. This is also true for the student populations

in *RQ3*. In the study toward *RQ3* (which impacts *RQ1*), for both the student and Mechanical Turk populations, the participants chose which tasks to complate and in which order. It is also possible that there was some overlap among Mechanical Turk participants and the classroom participants. Toward *RQ2(c)*, the participants performed large experimental tasks and there was only one available per Mechanical Turk worker, but they still self-selected as participants.

Additionally for *RQ2(c)*, the query generators were computer science graduate students, and potentially more expert than the average programmer who would be searching for source code. Thus, the results associated with more familiar query formats (i.e., keyword for Google) may be biased by better-than-average queries leading to better-than-average search results.

The results toward *RQ1* are based on self-reported answers on a survey and may not be representative of actual behavior. Observing programmer search habits in practice would allow us to validate the results.

9.1.2 Instrumentation

Toward *RQ2(c)*, we may have biased results in favor of Google by collecting snippets of source code for pages beyond the top 10 results, in the event that some webpages did not contain any source code (Section 7.2.2.4).

Toward *RQ3*, the instrumentation may have played a role in the results, since the study was performed on paper by one group and online for another group. On the one hand, it may lead to increased accuracy on the overall results since we are not over-constraining the study context. On the other hand, some tasks may be better suited for one context versus another, and this may have impacted the results.

When building the repository of encoded programs for Java, we remove each code snippet from its context. Thus, the encoded programs may not be entirely representative of the behavior of the source code in its original implementation. For example, if a method accesses a field and it has a static value defined within a class, we ignore that value and represent the field value symbolically. Addressing this threat and considering a broader scope of code snippets is part of future work.

Implementation errors are a risk for all studies involving Satsy (*RQ2*). To combat this error, we have developed extensive test suites and validated all Satsy search results (a reasonable endeavor considering the space of matches was relatively small). While this process is manual, it helped identify, and correct, for false positives. However, missed results are harder to diagnose, and for that we relied exclusively on test suites to execute and diagnose that behavior.

One implementation limitation discussed in Section 6.2 deals with string bounds. In the implementation used for all the studies in Section 7 and Section 8, it was possible for the solver to assign character values to a string outside of the bounds of the string length. While this was not observed to be the case for any of the test cases or study results examined by the researchers, this could have led to some false positives when the solver assigned string values, as was the case when strings are made symbolic via abstraction. However, since the checks on string values all occur within the bounds of a string (as shown by the inclusion of bounds on all string method encodings in Appendix A.1), the impact should be limited.

9.1.3 Resentful Demoralization

For the experimental tasks in *RQ2(c)*, the average completion time for the experimental tasks was 54 minutes and 40 seconds, which is longer than anticipated. With a payoff

of $3.25 per task, participants may have decided not to perform as well toward the end. To combat this threat, we randomized the ordering of the programming tasks that appeared in each experimental task.

9.1.4 History

In the evaluation of *RQ3* in the SQL domain, the input tables were the same for all tasks. While this was intended to ease the burden on the participant, it may have led to learning effects in the results, especially since the ordering of the tasks was not random for the classroom participants. It was, however, random for the Mechanical Turk participants.

9.2 External

External threats to validity deal with the generalizability of results. That is, do our results extend to other contexts?

9.2.1 Interaction of Selection and Treatment

For all studies, we may have used populations that are not representative of the populations we want to study.

For *RQ2(c)*, we did not collect demographic information for the human participants on Mechanical Turk. However, we did require them to pass a pretest including Java competency questions, which indicates that the participants are reasonably competent with Java.

For *RQ3* and *RQ1*, the participants in the study were Mechanical Turk workers and computer science undergraduate students. While only 8% of the participants had no

programming experience, these populations may not be representative of programmers who search for code. We should note, however, that 72% of the participants search for source code at least weekly.

For *RQ2(c)*, we selected eight of the original 22 questions for use in our study, based on the fact that our search returned some results (Section 7.2.2.2). This may have biased the questions toward those that are well-suited for our study. To combat this, we took a two-staged approach. First, when looking for 10 snippets with Satsy, we started with the constrained search where the type signatures matched. If not enough matches were returned (which was the case for seven of the 24 queries per Table 7.16), we relaxed the search to consider programs where $TS_p \supset TS_{LS}$ (Section 5.3) to fill out the 10 slots. Second, when Google did not return source code in some of the top 10 results, we continued looking at search results until 10 results were obtained (Section 7.2.2.4). Regardless, the bias remains.

9.2.2 Interaction of Setting and Treatment

For the studies toward *RQ2(a)* and *RQ2(c)*, we chose to compare Satsy to Google and Merobase, under the assumption that tools such as these are used for code search in practice. According to previous work and our own survey, Google is reported to be the most common tool used for code search, though code search engines like Merobase are also used. There could still be discrepancies between the self-reported tools and those actually used.

For all studies related to *RQ2* and *RQ3*, the selection of experimental tasks and questions may not be representative of tasks for which programmers would search for answers. For *RQ2(a-c)*, all tasks came from `stackoverflow.com`, so the assumption is that these are questions programmers actually ask. For *RQ2(d-e)*, the Yahoo!

Pipes queries were built from artifacts gathered from the community; since someone went through the effort of creating those artifacts, the assumption is that these are representative of real tasks. For *RQ3* and SQL, the questions were created by the researchers with the goal of language coverage; these may not be representative of actual SQL queries. Replication of this study with other tasks is needed to generalize further.

For *RQ2*, we have evaluated various aspects of the approach in just two domains, Java and Yahoo! Pipes. While these domains are diverse, it remains to be shown if the effectiveness of this approach generalizes to other domains. For *RQ3*, we have evaluated the input/output query model in two domain-specific end-user programming languages, Yahoo! Pipes and SQL. The extent to which this query model extends to other programming languages is yet to be explored.

For *RQ2(a)* and *RQ2(b)*, the syntactic queries were taken from the titles of questions on `stackoverflow.com`, and may differ from queries issued by programmers. We aimed to address this threat with *RQ2(c)* by using humans to generate the queries. Here, there is an additional threat to validity in that the questions posted on `stackoverflow.com` represent problems for which, presumably, the question askers could not find answers. Thus, it may bias the results against Google to look for results that are hard to find, and these may not be representative of questions asked to Google in general. To combat this, in *RQ2(c)*, we would skip over webpages that did not have source code until 10 snippets of source code were obtained. This way, the 10 snippets from Google could be compared to 10 snippets from Merobase and Satsy.

9.3 Construct

These threats to validity impact the ability to generalize the constructs of the experiment to the theory behind the experiment. That is, are we measuring what we intend to measure?

9.3.1 Mono-method Bias

For *RQ2(a)* in repository building, we retrieved only one-line solutions. Yet, some queries may require more than a one line solution, so those potential solutions were ignored. We addressed this threat in part with *RQ2(c)* by encoding entire methods, but this brings another threat to validity. Some methods may contain code that is 90% supported by our approach and that solves a query, but since we do not support the entire method, it is ignored. Thus, the repositories may be biased toward certain types of questions or problems.

For *RQ2(c)*, the questions we gathered from `stackoverflow.com` were classified according to their type in an effort to get a broad set of questions. This classification was subjective and may have biased the final set of questions used for the experiment.

For *RQ3*, the accuracy results for Tasks 8, 9, and 10 may be inflated by design. As described in Section 8.2.6.3, we did not collect results for those tasks performed in Mechanical Turk if the accuracy was lower than 50%. This was done to control for quality, but it also means that the accuracy of the tasks is biased. For these three tasks, we rejected the work of 12, 8, and 5 workers, respectively, based on accuracy.

We use P@10 extensively as a metric to indicate relevance of search results. Yet, this might not capture other aspects of search, such as the cost of looking at irrelevant results. Further, the P@10 metric may have been inflated by some of our decisions related to Google snippet collection toward *RQ2(c)* (Section 7.2.2.4).

9.3.2 Mono-operation Bias

For *RQ3*, the participants specified only the output in the tasks, whereas the goal was to evaluate the input/output query model. 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.

Toward *RQ2* with Yahoo! Pipes, the specifications collected from the Pipes artifacts represent the state of the RSS feeds on a particular day. Gathering the RSS feeds on a different day or at a different time can yield a different input/output, which can lead to a different set of relevant results.

9.3.3 Hypothesis Guessing

In the study toward *RQ2(c)*, the query generators were taken from the pool of graduate students and staff associated with the ESQuaReD software engineering research lab. Some of them may have been familiar with the goals of this research, so the input/output queries provided may have been of superior quality in an effort to support this research, or of inferior quality intended to sabotage it.

9.4 Conclusion

Threats to conclusion validity concern our ability to draw conclusions about the outcome of an experiment based on the treatment.

9.4.1 Random Irrelevancies in Experimental Setting

For those studies involving Mechanical Turk (*RQ1*, *RQ2(c)*, and *RQ3*), we could not control the context in which the participants interacted with our study tasks. Thus, there could have been elements outside of the tasks, such as noise or distractions, that influenced the results.

9.4.2 Reliability of Measures

For the survey toward *RQ1*, the questions asked may have been poorly worded or may have been misunderstood by participants. Thus, we may not have captured actual information about participant search habits.

Toward *RQ3*, the accuracy and timing data were highly dependent on the quality and clarity of the problem description. While the researchers tried to make the descriptions succinct, these measures may be unreliable. A similar threat exists for *RQ2(c)*, where the problem descriptions may have been misunderstood by the query generators. This is why we used multiple query generators for each combination of search approach and problem factors in the experimental design (Section 7.2.1.2).

Toward *RQ2(c)*, the Mechanical Turk participants who judged the snippets were asked if each snippet was relevant to the problem, and if it solved the problem. Definitions were given for each of these words (Section 7.2.4.1), yet, the participants were judging the source code in a static view and did not need to work with the code in any context outside of a textual description of the programming task. Therefore, their judgement of relevance and solving may not be reliable.

9.4.3 Reliability of Treatment Implementation

Toward *RQ3*, we used two pools of participants, Mechanical Turk and students, which required two different implementations of the study. In the classroom implementation, two different researchers presented the study to the different classes on two different occasions. By IRB protocol, there was a script to be used when introducing the study to ensure consistency. As an aside, there are no significant differences in accuracy between the students and the Mechanical Turk participants at $\alpha = 0.10$ on all tasks except for *Pipes Task 3* in which the student results are significantly lower ($p = 0.0212$). We suspect this is a result of differences in instrumentation (i.e., paper vs. online) for the two groups.

For *RQ1*, the search habits are self-reported and based on the participants' best recollection. These may not be representative of their behavior in practice. Actually observing participants searching for code is part of the future work to gain an understanding of what programmers use, have, and need when searching for source code.

Chapter 10

Related Work

This work involves the combination of two diverse software engineering processes, searching for source code and program analysis. In this section, we discuss related work within each of these categories.

10.1 Code Search and Repository Analysis

Searching for source code as a human process involves specifying queries, retrieving information from repositories, and delivering the search to the programmer. In this section, we discuss related work for each of these research areas.

10.1.1 Code Search

We have described an approach to code search that is semantic and uses input/output examples to define the queries, a repository of programs to define the set of potential matches, and an SMT solver to identify the matches. There are two areas of related work here, syntactic code search and semantic code search.

10.1.1.1 Syntactic Code Search

Recent studies have revealed that programmers typically use general search engines to find code for reuse [59]. More specialized code search engines (e.g., Koders, Krugle, ohloh) incorporate various filtering capabilities (e.g., languages, libraries) and program syntax into the query to better guide the matching process [59]. Other approaches add natural language processing to increase the potential matches [19, 43]. For example, Exemplar is a search tool that takes a natural-language query containing high-level concepts (e.g., MIME, data sets) as input, then uses information retrieval and program analysis techniques to retrieve applications that implement these concepts [19]. Our work is different in that we perform a more semantic search, but as we show (Section 7.1.2), both approaches are complementary and can be easily combined.

Other code search approaches are targeted at a specific code search sub-problem [45, 55]. XSnippet allows programmers to search over a repository for sample code related to object instantiation [55]. The query model exploits the type and hierarchy of the provided object, and matches are based on mining code for instantiations of that object. By limiting the scope of the search to object instantiation, this search can generalize the query to return code for instantiations of an object with a similar type and interface. Another approach focuses on searching for source code with specific API usage [45]. By mining temporal specifications from code snippets, a query uses a partial program and an automata-based approach finds source code that uses the constructs in the partial program.

Instead of a program explicitly providing a query, another specification model involves extracting code structures from a developer's work environment. Recent work has used such structural characteristics to locate example code in a repository [25, 26]. Method signatures, return types, and fields are used to find potential code to reuse. To

increase the chance that the recommended code behaves according to the programmer's needs, heuristics are used to match the code already written by the programmer against a repository of source code to better identify structurally matching code that might be relevant [26]. We may be able to use similar principles in a ranking algorithm for results that would consider the programmer context (Section 5.3.2.3).

Instead of recommending source code, other research has sought to recommend code-related web pages [57]. Our search approach operates under a similar assumption, that programmers frequently search the web for source code. This related work has shown that when programmers search for code, nearly 14% of the webpages visited are actually re-visits; their goal is to help programmers find those revisited webpages faster. Rather, our work aims to help programmers find source code to reuse.

10.1.1.2 Semantic Code Search

Early work in semantic search required developers to write complex specifications of the desired behavior using first-order logic or specialized languages (e.g., [17, 48, 81]), which can be expensive to develop and error-prone. The most similar work to ours is that on specification matching by Zaremski and Wing [81, 80]. They propose an approach for specification matching for component retrieval, where the specifications could be defined by signature [80] or by pre/postconditions [81], and our work is most similar to the latter. In their approach the pre/postconditions for the desired component are written by hand, and their approach matches the specifications with the components using a theorem prover to show equivalence. In a relaxation process, they weaken or strengthen the matching on the pre/postconditions. In our approach, we weaken the encodings of the programs (analogous to components) in addition to the specifications. Similar to their approach, the components searched are first filtered by type signature, which is a technique we apply when searching our Java repository in

which inputs and outputs can have different datatypes (Section 6.4). Their definition of code reuse, *adapt[ing] a component to fit its environments constraints*, is similar to the definition of *relevant* code we use for calculating precision and recall (Section 8.1.4), corresponding to the *copy/paste and modify* behavior that nearly half the programmers we surveyed reported to do with useful code once found (Table 3.3). Building on this search model, Penix, et al. [48] showed that for component reuse, automated reasoning can be used to determine what changes are necessary to reuse a component and guide component adaptation, using an extended version of the lattice from the previous work to identify closely, but not necessarily exactly, matching components [81]. In the evaluation, the authors use a data-flow language abstraction with many similarities to the Yahoo! Pipes components supported in our approach. If no components match the specs of the query, the first step they take is to weaken the requirement for feature-equivalence, in essence finding the same operations performed on different data types. The second step is to reduce the number of query features the component contains, in essence finding partial component solutions for the problem, using automated reasoning to guide the relaxation.

Other approaches to semantic search use queries in the form of algebraic specifications. The component behavior is modeled using finite-state machines and a constraint solver is used to check the satisfiability of queries [17]. In this approach, the queries must be written directly in the specification language and the scope of applicability is small, finite programs. Additionally, the algebraic models are unable to identify components that might be close matches to a specification.

The previous approaches to semantic search require a complex query model, which comes at a high cost. The cost of writing specifications can be reduced by using incomplete behavioral specifications, such as those provided by test cases (a form of input/output) [39, 50, 53], but these approaches require that the code be executed to

find matches. Further, executing test cases against the code will return only exact matches, missing many relevant matches that may simply have a slightly different signature (e.g., extra parameter). Reordering the search parameters is an approach that can find relevant code with a slightly different signature from that specified originally by the programmer [53].

10.1.2 Studies on Repositories

Our research is dependent on the existence of large and diverse code repositories from which source code can be matched and returned during search. Repositories provide a mechanism for programmers to share code and learn from the experiences of others, and tend to attract many participants to the communities. For example, Yahoo! Pipes, one of our targeted domains, has over 90,000 users [32] and over 100,000 artifacts [49]. The number of lines of code from open source projects that are available and can be searched by code search engines, including code in Java and SQL, is over ten billion [47].

Recent work has aimed to capture the diversity that exists in large repositories of code. Clone detection (e.g., [30]) is a means of identifying locations with similar code, but are highly dependent on the structure and does not characterize repository diversity. Yet, it can be observed through clone detection studies that there are more clones when the size of the clone is small [30]. Studies of syntactic diversity confirm that there is low diversity among smaller code fragments. A study with over 420 million lines of source code in Java, C, and C++ revealed that at least 90% of code is syntactically redundant when considering code with 20 tokens or less (where six tokens is approximately one line) [14]. Our approach could leverage work such as this to do a smart search over code in a repository, starting with syntactically diverse code

early in the search. Not surprisingly, such syntactic redundancy is not limited to these languages; in our study of 32,000 Yahoo! Pipes programs, we found that nearly 60% of the programs were identical to another in the repository when the field values were abstracted away [71].

Mining software repositories is an active area of research with the goal of extracting and measuring properties of large data sets. The main objective of research in this area is to explore trends across different resources and aggregate information. An example of this is to extract temporal properties regarding interface usage from open source Linux projects for the purpose of anomaly detection [20]. Another example uses language modeling to uncover programming practices by mining projects hosted on GitHub [1]. Related to queries, other work has defined a query language for source control repositories that allows programmers to ask questions about the repositories, such as looking for an author who modified a file after another author [24]. Contrastingly, we are interested in the problem of extracting existing information from large repositories for the purpose of reuse, rather than data aggregation.

10.1.3 IDE Programming Support

As we are suggesting relevant source code for a programmer to use and possibly integrate into their development context, our approach has some similarity to code completion. Some approaches to this consider the program code that has previously been written to recommend likely APIs to use in the present [54]. Similar also to syntactic search, other approaches extract keywords from the code and also consider those when making code suggestions [42].

Our approach also has the potential to be useful for, and automate parts of, test-driven development [16]. Since the programmer is required to specify the behavior

of the code in terms of input and outputs, using these as tests could identify code that behaves as the developer needs to automate part of the development process.

10.2 Program Analysis

In our work, we leverage several program analysis techniques, including symbolic execution and constraint solving. Additionally, our work has implications for code reuse and program synthesis. In this section, we discuss related work in these areas.

10.2.1 Verification and Validation

In this work, we have talked about how symbolic analysis is used to generate constraints that represent the program behavior, and that this representation is used in the search process. Symbolic execution [7, 8, 36] is a technique that executes code with symbolic, rather than concrete, values, and can generate such symbolic summaries of source code. We leverage these existing tools, when available, as described in Chapter 6.

For Yahoo! Pipes, symbolic execution tools are not readily available, so we were responsible for all steps in the translation process [69]. For Java, however, tools like the symbolic execution extension [33, 34] to the Java PathFinder model checker [75] can generate symbolic summaries that we can use, but are limited in the data types and libraries that are supported. String processing support in particular is still limited, since the problem of path feasibility for strings is undecideable in the general case, but decidable for bounded strings [5]. String constant solvers built on top of SAT solvers have been used to automatically fix HTML generation errors [56]. In this work, only three string operations were considered for the repair task: insertion, deletion, and repair. These activities involve only *pure* string constraints. In JPF, inclusion of support for methods that have mixed integer and string constraints, such

as the $indexOf$ method that returns an integer based on a property of a string, has recently been investigated [52]. For path identification in multi-path programs, we are dependent on such support when such methods are involved in the path conditions.

In validation, constraint and SMT solvers have been used extensively for test case generation. Toward the goal of database generation for testing, reverse query processing takes a query and a result table as inputs and using a constraint solver, produces a database instance that could have produced the result [4]. Other work in test case generation for SQL queries has used SMT solvers to generate tables based on queries [74]. In our work, we do not generate database tables, but rather determine if a given query could have produced a specified result set (output) from specified input table(s).

Under the assumption that building symbolic execution engines is tedious for new languages, recent work has sought to automatically synthesize symbolic execution engines from input/output examples [18]. This work has shown promise for x86 instructions, which are small and side-effect-free, and deal only with primitive data types, so limitations abound. Such an approach would be extremely useful for the generalizability of our approach, where new domains could be added and encoded by simply generating a symbolic execution engine from examples.

10.2.2 Program Generation and Program Synthesis

Previous work in the area of automated program generation relates to our work in that the high level specifications are used as the basis to derive programs [2]. Some approaches represent the program code as algebraic equations [3], which is similar to our approach for representing program code as constraints. Generalizing these specifications can allow for the automatic generation of Java code, in this instance,

from the equational models [3]. The goals of this previous work and our approach are similar, to end up with code that matches specifications, but the approaches are different. Instead of generating a program, as is done in the previous approaches, we search for existing code that matches or approximately matches the desired behavior.

Closer to our work is program synthesis that uses solvers to derive a function that maps an input to an output (e.g., [21, 22, 23, 29, 60]). The domains of applicability include string processing [21], spreadsheet table transformations [23], a domain-specific language for geometric constructions [22], data structure manipulations [60], and loop-free bit manipulation programs [29]. Some of these approaches also utilize supplementary information like the code structure to help guide the solver [60]. The key difference between program synthesis and our approach is that we use the solver to find a match against real programs that have been encoded, while these synthesis efforts have to define a domain specific grammar that can be traversed exhaustively to generate a program that matches the programmers' constraints.

Tinker is a *programming by demonstration* tool built for the Lisp language [41] that also uses examples, but not solvers. The Tinker tool allows beginners to specify concrete data inputs in Lisp and specify how each example is handled using expressions. The tool records the manipulations and produces a program to automate the process.

Another means for program synthesis, as opposed to specifying input and outputs, utilizes partial programs called sketches (e.g., [51, 62, 63, 64]). These approaches typically operate over small, low-level and finite programs like bit vector manipulation [64] and hardware circuits [51], but more recently have targeted higher-level applications such as concurrent data structures [63] and optimizing code [62]. Part of our future work involves partial programs as a supplementary specification for our search approach, which makes this area most similar to our future work.

10.2.3 Code Reuse

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

For effective reuse, scope and dependencies must be understood for developers to effectively integrate code [15]. Some recent work assists programmers with integrating new code by matching it to structural properties in their development environment (e.g., method signature, return types) [9, 26]. These approaches guarantee structural matching, but the behavior of the integrated code may not be well understood. In our search, the user provides the input and output of their specified code, which contains some structural information in terms of types. After code is found to be a match, replacing the variables in a code snippet with variables from the developer's context is straightforward, since that mapping is a necessary part of the search process. This would be a step toward integration.

Chapter 11

Discussion

In this section, we describe the implications of this work and of the studies we have conducted, and how each relates to our future work.

This work involves the interlocking of two very different software engineering processes. One of them, the process of searching for source code, is a deeply human task, rooted in the context of a programmer looking for something, with success criteria judged by the programmer themselves. The other process, computing and using symbolic summaries of source code, is deep in the heart of state-of-the-art of program analysis techniques, leveraging symbolic execution and constraint solvers. While an unlikely combination, we have shown in this dissertation that these processes can be combined in such a way to benefit the programmer, and at the same time, push the state-of-the-art in code search.

From the human perspective, our approach leverages examples to describe desired source code. In programming, examples can be very powerful. Programmers use examples in a formal sense by writing functional test cases, in describing a situation to a colleague, or in posting a question online (Section 3.3). In this work, we have proposed the use of examples for another purpose, searching for source code to reuse.

From the program analysis perspective, our approach leverages symbolic execution and constraint solvers. Yet, it is also limited by the current tools. Specifically, in the process of collecting constraints for multi-path programs, the limitations of state-of-the-art symbolic execution engines with respect to string processing restrict the expressiveness of the programs we can encode (Section 7.2.2.1). Further, the constraint solver runtime in the presence of large specifications as was the case with Yahoo! Pipes, may make this search approach impractical in practice (Section 8.1.4.2). As program analysis techniques and solving approaches, theories, and tools improve in speed and expressiveness, so does our approach to code search.

In the introduction of this work (Chapter 1), we presented the code search problem as having three stages, querying, indexing, and matching. We have explored various aspects of each of these stages, to various degrees, with various languages, as a means of validating the research ideas outlined in the general approach (Chapter 5) given the instantiations in two languages, for Yahoo! Pipes and for Java (Chapter 6). In the next sections, we discuss the impact of the work presented in this dissertation and point to open questions and future directions related to our exploration.

11.1 Empirical Studies

The studies presented in this work form a first step toward evaluating the effectiveness and utility of the search. More studies are needed, particularly related to *RQ2(c)* where we used only eight of the 22 problems in the final implementation. Additionally, each snippet from each search was evaluated by only one participant on Mechanical Turk, so it would be useful to have replication to get an average relevance per snippet. Evaluating the impact of abstraction and solver time on Satsy was evaluated only in the Yahoo! Pipes domain. Extending the implementation of Java to include abstraction

is part of future work, and will require an evaluation similar to the one in Yahoo! Pipes toward *RQ2*.

Additionally, we identify more opportunities for empirical studies related to code search, including characterizing code search behavior, evaluating further the feasibility of input/output as a query model, exploring other query models, and delivering the search to the programmer.

11.1.1 Characterizing Code Search Behavior

Searching for source code is a human process that is performed frequently during programming tasks [59], as we have shown in Section 3.1. We have proposed the use of input/output examples as a query model for code search. Two of the studies presented in this work, toward *RQ2(c)* and *RQ3*, in addition to the user survey toward *RQ1*, aim to characterize search behavior and evaluate the feasibility of using examples for search queries.

Our user survey showed that programmers frequently search for source code while programming (Section 3.1), corroborating the findings of previous work [59]. Of the 109 programmers surveyed, most reported that they search to find code that can be used for implementation ideas (67%) and/or to obtain code that can be reused with modification (48%, Table 3.3). With this high frequency of code search, it's important for researchers and tool builders to focus on getting the programmer what they need with minimal time and effort.

Understanding the real cost of a code search in the state-of-the-practice is important for researchers in code search. This would give a baseline to compare against when introducing a new technique. Observing programmers in practice while they search for source code could provide an indication of the cost and frequency of query reformulation,

the number of snippets explored, and the cost of exploring a snippet. This could give an overall measure of how much time and effort it takes to find relevant code.

In our second Java study (Section 7.2), we used human participants to judge the relevance of source code to a programming task. Another future direction is to interview programmers immediately after a code search task about why they stopped the search, what they learned, and what source code they may intend to reuse. Alternative studies could include think-aloud methods during code search tasks or instrument the search tools. From such studies we could also learn if programmers are open to using and learning from search results in different languages that could provide a similar solution (i.e., a C# solution to a Java question)? This would give us an idea of how much, and how many, languages need to be supported to make our approach to code search, and Satsy, useful in practice.

11.1.2 Feasibility of Input/Output Examples as Queries

In the study for *RQ2(c)*, we show, indirectly, that graduate students in computer science (i.e., the query generators) are rather adept at generating input/output queries to describe simple programming tasks, specifically those in Table 7.8. We know this by looking at the fact that the $P@10$ value for the Satsy search approach was 0.533 overall (Table 7.11), indicating that two things went right, 1) the queries were descriptive of the problem, and 2) Satsy had some relevant solutions in the repository. The former speaks to the ability of programmers to generate input/output queries in Java.

Yet, Java is only one of the languages targeted by our approach (Chapter 4) and implementation (Chapter 6). In a controlled quasi-experiment toward *RQ3*, we explored the feasibility of using input/output examples in Yahoo! Pipes and SQL. Presumably, the participants in this experiment were less familiar with these domain-

specific languages than the graduate students were with Java, yet, the participants were able to select output given an input and problem description with over 90% accuracy in just a couple minutes (Section 8.2.6). Performing a similar study in Java is left to future work.

11.1.3 Alternative Query Models

Although we have shown that programmers can compose queries as examples, and that they do this when asking questions online (Section 3.3), we would like to explore some alternative and some assistive specification models to find the one (or combination) that best meets programmer needs when searching for source code. Presently, we have several ideas for how to make the query model more natural and/or expressive. For example:

1. Negation in Examples. There may be a need for programmers to specify examples that throw exceptions, return some error condition, or specify output values that should not be allowed given an input. Being able to recognize error conditions from source code, and then model them as error conditions, would then allow programmers to include generic error conditions in their specifications, instead of guessing how another programmer may have implemented their error conditions. Allowing negation of the output given an input would address the latter need.

2. Partial Programs. Developers may know part of a solution, but not the whole solution, and may benefit from a search that allows them to provide what they have and receive a completed piece of source code as a result. This is similar to program synthesis by sketching [51, 62, 63, 64], except that in our approach, the programmer would not need to specify the "holes". Instead, we could find existing code that has the same pieces as the original and behaves

as specified with input/output examples. A natural extension of this approach would be, then, to let programmers specify variables or expressions, perhaps sharing variables, rather than concrete values, as part of the input/output pairs.

3. Inferring Additional Examples. This approach would be assistive in that it would help programmers create more general specifications. The idea is to infer structured data from a programmer's specification, possibly using a data abstraction like Topes [58], and then automatically generate other specifications with different formats but the same intention. For example, say a programmer gave the following input/output pair:

```
LS: {({"402−432−2345"}, {"402"})}
```

Recognizing the input as a phone number and the output as the area code, we could infer the following, additional specification:

```
LS′: {({"(913) 212−4545"}, {"913"}),({"+1 515 020 3113"}, {"515"})}
```

Thus, a new, stronger specification can be created on behalf of the programmer, yielding a more general solution from the search.

4. Inferring Patterns. Similar to the previous example, another assistive approach could infer patterns on the input. For example, given the following specification,

```
LS: {({" rizzo@ridell .org"}, {"rizzo"}),({"sandy@univ.edu"}, {"sandy"})}
```

We could infer a pattern of the following, where X and Y are symbolic values:

```
LS: {({X"@ridell.org"}, {X}),({Y"@univ.edu"}, {Y})}
```

5. Other Program Properties. Any property that can be encoded for a program can also be used as a specification, such as programming constructs used, number of parameters to a function, or even performance. In some cases, as with

the programming constructs used, this specification may act as a syntactic filter on the semantic results. For example, if a user gives the example input *susie@mail.com* and output *susie*, specifying also the use of the function `indexOf`, would prevent some coincidental matches, such as:

```
String scheme = uri.substring(0, 5);
```

which is coincidental unless all e-mail aliases have exactly five characters.

6. Conditional Specifications. Adding predicates to the lightweight specifications would allow the programmer to segment the input space and provide input/output pairs that should hold under various conditions. This means the programmer could provide symbolic inputs and outputs instead of specific test values. For example, consider a search for a program that returns the square-root of a number, or zero if the number is negative. The programmer could specify:

Predicate	**input**	**output**
$x \geq 0$	x	$sqrt(x)$
$x < 0$	x	0

7. Generating Additional Examples. For multi-path programs, there may be an opportunity to create additional input/output examples based on paths that are not matched by a specification. That is, additional input/output pairs could be generated for each path $q \in Q_P \setminus Q_{sat}$.

11.1.4 Delivering the Search to the Programmer

Delivery of search to the user is another dimension of this approach not yet explored. This has implications for HCI, end-user computing, and software engineering research, as we seek to answer the following questions: how to deliver the search to the

programmer, and when in the development process is search most appropriately made available?

We have built this approach on the principle that the user should not be exposed to the underlying representation used in the search (i.e., the constraints). This raises many questions and challenges, such as how to present abstractions to the programmer, how to explain why some snippets might be better than others, or how to guide the programmer to refine their query to find a suitable piece of code. These will all need to be addressed as this work progresses.

11.2 Extending the Approach

Our approach has been defined to be general enough to cover a broad set of languages, from Yahoo! Pipes to SQL to Java. Still, there are many directions that can be explored to extend the approach. Specifically, we describe how to use the solver to guide modifications to the specifications and how to stitch programs together in program composition. Other future directions include identifying additional abstractions for encoding, supporting additional languages, and improving matching, ranking, and performance.

11.2.1 Solver-Guided Specifications

When specifications are too strong, we may be able to provide some guidance to the programmer on how to appropriately tune their specifications. We propose to provide this guidance by extracting and using the *unsat core*. When an SMT solver returns *unsat*, this happens because a satisfying model cannot be found, and thus some constraints could not be satisfied. Extracting the *unsat core* from the solver identifies these constraints. The programmer is the owner of the specification, so

guidance could be provided on how to tune the specifications by suggesting to negate the constraints pertaining to the specifications in the *unsat core*.

When a specification is too weak, there may be many matches. The solver could also be used to generate sample input/output that would differentiate between matches to help programmers see how two pieces of source code differ.

11.2.2 Program Composition

Our current implementation returns single programs to the developer, yet there are some instances when no program matches *enough* of the specification. This could mean that the specification goes entirely unmatched, or that it goes partially unmatched.

Even after abstraction, it could be that no single program matches the user specifications. However, composition of programs, where the inputs and outputs of multiple programs are chained together, could create the desired behavior. This could be particularly useful for repositories that serve small communities, such as a biology lab that uses Perl. This would promote reuse first, but also help scientists to grow the repository by creating new source code. In this way the repository would better fit the current needs of the community.

Another approach could be useful when a program matches some, but not all, of a specification. Counter-example guided inductive synthesis [40] is designed to use concrete input/output pairs that are *unmatched* to guide modification to the program P, where P has some "holes" that have been identified. We may be able to use similar principles in augmenting source code that partially matches a specification.

11.2.3 Abstraction

Our current implementation supports abstraction on data types only (Section 5.2.3). However, other abstractions or models may be helpful to allow the search to index richer code.

Section 5.2.3 describes possible abstraction strategies for the encoding, specifically using the data types. Finding appropriate abstraction levels (i.e., instance abstraction vs. abstracting all values) will require thorough experimentation and specialized heuristics. We also plan to explore other lattices that consider program properties, such as object and heap shape. These might be useful for finding relevant programs that are similar to a current implementation, and are part of our future work.

11.2.4 Language Support

Our current instantiation covers most of the Java String library, arithmetic operations, and list processing in Yahoo! Pipes (Chapter 6). We are able to provide solutions for many different programming tasks, as shown throughout the evaluation chapters. As shown in Chapter 7 and Chapter 8, we can use a combination of existing symbolic execution engines (e.g., JPF for Java) and our own implementations (i.e., for Yahoo! Pipes) to generate symbolic summaries of source code. Even with the current limitations of existing tools, we could create a repository with over 8,000 Java methods from open source projects (Section 7.2.2.1).

Extending the language support would allow us to encode richer programs, provide solutions for more complex query, and increase our potential to have an impact on the state-of-the-practice. We have several directions planned for this.

The next constructs in Java we plan to support are the *null* value, lists, loops, arrays, and file manipulation. Some of the query generators in the second Java study

(Section 7.2) used *null* and arrays as parts of their input/output queries, and since those are not supported by our approach, the specifications could not be used. We also plan to extend this approach to concurrent programs, where we would model the concurrency using, for example, the number of processes and shared variables. Loop processing remains a challenge in symbolic execution [72], and our current implementation does not support looping constructs. However, loops are frequently used, and our absence of support has limited the types of questions we can answer and the size of our constraint repository. Modeling object creation and heap manipulation are two additional challenges of the indexing phase not yet addressed. Tackling these challenge will be important for the practicality of this approach.

The current implementation in Java is limited to a single output. However, we can think of many situations when multiple outputs would be helpful. A simple example is decomposing a URL into domain and path. Extending the implementation to allow for multiple outputs is part of our future work.

As we have pointed out, the investment for encoding a new language is quite high, but happens only once (Section 6.5). We are interested in exploring techniques that would ease the burden of extending our approach to new languages. One possible direction is to automatically synthesize symbolic execution engines from input/output examples [18]. We are interested in identifying and extending support for our approach to new, promising domains. Specifically, we are interested in extending the approach to C, as it is the most common programming language today [73].

11.2.5 Performance

For our studies in Yahoo! Pipes, finding a relevant program from a specification could take on the order of minutes (Section 8.1.4). With the Java examples, which had

much smaller specifications, the solver returned in under a second (Section 7.1.1.4). As the complexity of programs and queries supported by our encodings increase, so does the solver time.

Encoding a language fragment requires the definition of many uninterpreted functions, axioms, and data sorts that are required for encoding an arbitrary piece of code in that language. However, not every piece of code requires all that information as part of the constraint system. Retaining only the functions and axioms required for encoding a piece of code could increase the performance of the solver.

For example, to support string encoding, there is a lot of set-up code that is required. Several axioms and interpreted functions are declared, forming a preamble for every encoded program (Section 6.2). However, not all of this information is required for every implementation. Specifically, we have a character set that is required for C_P, and for LS_{enc}. Each of these is a subset of the actual character set that is included in the header tacked on for solving. There may be a benefit to using a "minimum set" of character encodings by combining the required ones for C_P and LS_{enc} and pruning the duplicates. One drawback is that it would limit the solver when instantiating symbolic variables, and this could lead Satsy to over-reject. If the performance gains are big enough, it might be worth the risk. Evaluating this is left to future work.

11.2.6 Multiple I/O Satisfiability

Our current implementation supports the inclusion of multiple input/output pairs. This is useful so programmers can more clearly illustrate the behavior of their desired code.

When $\mid LS \mid > 1$, we currently consider the I/O pairs independently when solving. Future work is to consider them simultaneously and identify matching variables.

This requires bootstrapping the solver to identify the variable bindings that led to satisfiability, and then using that information in the linking phase for subsequent input/output pairs. For example, consider the code in Listing 11.1 and the following input/output pairs:

$LS_1 \leftarrow (\{2 : int, 3 : int\}, \text{false} : boolean)$, and

$LS_2 \leftarrow (\{3 : int, 2 : int\}, \text{false} : boolean)$.

Listing 11.1: Simultaneous Satisfiability Example

```
boolean between(int begin, int  pos) {
    return (begin <= pos);
}
```

Considered independently, $Solve(C_{ls_1} \wedge P_{11.1}) \rightarrow SAT$ and $Solve(C_{ls_2} \wedge P_{11.1}) \rightarrow SAT$. Using the satisfiable model and referring to I_1 as a list rather than a set, $begin \mapsto I_1[2]$ and $pos \mapsto I_1[1]$. For ls_2, this mapping is reversed, where $begin \mapsto I_2[1]$ and $pos \mapsto I_2[2]$. In order for these input/output pairs to be considered simultaneously, the binding of the satisfiable $model$ from $Solve(C_{ls_1} \wedge P_{11.1})$ needs to be retained and used as input when considering ls_2, for example, $Solve(C_{ls_2} \wedge P_{11.1} \wedge model_1)$. This will treat each index in the input specification as the same variables in the program encoding, and then simultaneous satisfiability may be possible.

It must also be considered that the initial binding for the first input/output pair may not work for the second input/output pair, but an alternative binding would work for both. Consider an alternative $ls_1 \leftarrow (\{2 : int, 2 : int\}, \text{true} : boolean)$. The binding $begin \mapsto I[2]$ and $pos \mapsto I[1]$ may result from $Solve(ls_1 \wedge P_{11.1})$, but $Solve(ls_2 \wedge P_{11.1} \wedge model_1) \rightarrow unsat$. However, swapping the bindings so $begin \mapsto I[1]$ and $pos \mapsto I[2]$ works for both input/output pairs. In the event that the first model does not work, negations of the initial bindings can be added as additional constraints

on the first solver invocation to force the second binding model, which then identifies Listing 11.1 as satisfying, simultaneously, LS.

11.2.7 Ranking

One of the most important factors in the effectiveness of a search approach, beyond having a rich pool of artifacts to search over and being able to identify results, is to return the most relevant results first. This process is called ranking, and it really matters. From the results of *RQ2(c)*, and specifically the exploration in Section 7.2.5.3, we found that the overall performance of Satsy may have been handicapped by an inadequate ranking algorithm. By creating a ranking algorithm based on participant responses to the Satsy snippets, and regenerating the top 10 search results based on the new ranking, we were able to predict that our hypothetical ranking for Satsy could increase the overall $P@10$ metric from $\bar{x}_{s\cdot} = 0.533$ to $\bar{x'}_{s\cdot} = 0.657$, putting it nearly on par with Google's performance with $\bar{x}_{g\cdot} = 0.675$.

Based on the findings in *RQ2(c)*, we are very interested in finding a good ranking algorithm that will make us competitive with the state-of-the-practice. Many factors likely play a role in the ideal ranking approach, such as the amount of program or specification that is matched, the complexity of the source code, who owns the code, and so forth. Identifying the important factors in ranking requires much more exploration.

11.3 Other Applications

We have shown how to use our approach to code search for the sake of code search. However, the approach we have defined and implemented may have applicability in

other areas of software engineering, such as tracking repository evolution, refactoring, automated matching, and test-driven development.

11.3.1 Repository Evolution

Our repository of encoded programs is built by scraping existing code, but ignores the fact that source code repositories grow as new code is committed and updated. One question we will explore in future work is how to handle changes in the source code used to build our repository. It may be useful to keep the old versions of the code, unless that code is buggy. This information may also be useful for ranking purposes, where code from more recent versions of projects would be reported first. It could additionally be used to summarize semantic changes to a repository over time.

11.3.2 Refactoring

In the Yahoo! Pipes implementation, part of the encoding step involves refactoring the programs for simplicity and size reduction. This support, however, has not been integrated into the encoder for Java. One future direction of work is to pre-process programs prior to encoding and remove smells like unreachable code, unused variables, and unnecessary abstraction.

11.3.3 Automated Patching

Another application of this approach could be to automate the generation of code patches. Say a program has a fault that has been isolated to a particular line (using, for example, fault-finding tools such as Tarantula [31]). Traces of passing test cases provide situations under which the code behaves correctly passes. The failing test case provides an input and an undesirable output. This information could be leveraged in

searching for a patch to the program that would behave as specified by all test cases. Exploring this research direction is left for future work.

11.3.4 Test-driven Development

Another application is for test-driven development, where the tests would define the behavior of a unit to be searched. It could therefore be possible, with a suitably rich repository, to compose a program from its test suite. While this could expedite the development process, if the specifications are not carefully crafted, code that over-approximates the behavior of a specification could have unintended consequences. A programmer would likely need to be involved at every step of the process to act as a "code reviewer."

11.4 Conclusion

We have presented an approach to source code search that uses input/output examples as queries and searches a repository for source code that matches the defined behavior. The novelty of the approach lies in the use of an SMT solver for matching input/output examples to source code that meets the specification. This necessitates a transformation process on the source code and the specifications into first-order-logic so the solver can identify matches.

In this work, we have motivated the need for better code search tools and techniques, formalized the definition of this approach, described its implementation in a tool, Satsy, and evaluated the approach across several dimensions.

To motivate the need for better code search, we surveyed over 100 programmers about their search habits, finding that code search is a common task and that current search tools are often inadequate. To justify the choice of an input/output model for

queries, we explored questions asked by the community on `stackoverflow.com` and found that questions are frequently accompanied by input/output examples, indicating that programmers already think in this way when looking for help online. Additionally, we conducted a quasi-experiment that showed programmers can quickly and accurately compose specifications in this model.

Given that the input/output query model is reasonable, we also evaluated the effectiveness of our search using Satsy applied to two languages, Java and Yahoo! Pipes. In Java, we conducted a series of studies comparing Satsy to state-of-the-practice search engines, Google and Merobase, a code-specific search engine. Our results show that Satsy outperforms Merobase and is competitive with Google in terms of returning relevant results, where relevance was judged by 30 programmers. Further, we show that Satsy can operate on top of a syntactic search engine by filtering the results based on desired behavior. Doing so can reduce the amount of source code that must be evaluated by a programmer by 34%. In Yahoo! Pipes, we showed the impact of abstraction and the power of the approach to identify approximate matches when exact matches do not exist in a repository.

While this example-driven approach to semantic code search seems promising in the domains we studied, generality remains a goal that needs to be addressed in the context of richer programs and possibly with alternative specification models. Exploring these directions, and others, are outlined as part of the future work.

We have presented the first work in applying SMT solvers to the problem of code search. This is just one step toward our ultimate goal of leveraging existing resources, such as source code repositories, to positively impact programmer productivity.

Appendix A

Encoding Details

The following provides the encoding details for the Java and Yahoo! Pipes implementations (see Chapter 6)

A.1 Java

The encoding logic for each of the support string library methods is shown in Table A.1, Table A.2, Table A.3, Table A.4, Table A.5, are Table A.6. It is organized by method name, alphabetically. For each method call, the constraint purpose and definition are presented. When types for the variables are not stated, we used the following scheme: i, j, k and $from$ are integers, s, sub, t, and u are strings, c and d are characters, and b is a boolean. Each row in the tables is implemented as an *assert* statement in SMTLIB2 format, with one exception to handle disjunction.

For the indexOf and lastIndexOf operators, the constraint for *ensure s[i] = c* needs a little more processing to protect against interference in the values. That is, for $s.\text{indexOf}(c) = i$ in Table A.2, we want to force i to be the index of c in s when c is at index i in s, and otherwise let it $i = -1$. This would protect against the case

where $(s.\,\text{charAt}(c) = i) \wedge (i = -1)$ could evaluate to true from the first and third constraints. So, the constraint $(s.\,\text{charAt}(i) = c) \vee (i = -1)$ is implemented as follows:

```
(assert
    (ite
        (exists
            ((j Int)) (and (>= j 0) (< j (length s)) (= (charAt s j) c)))
        (>= i 0)
        (= i (- 1)) ))
```

This means that if character c exists at some index j within the bounds of $[0, length(s))$, then i must be at least zero, otherwise it is set to -1. This presents a forcing condition that ensures $i = -1$ is not the default behavior.

The processing is a little more complicated when dealing with strings as the arguments, as in $s.\,\text{indexOf}(sub) = i$ in Table A.2. The following ensures that i is the index of sub is s if sub appears in s:

```
(assert
    (ite
        (exists ((start Int))
            (and
                (>= start 0)
                (< start (length s))
                (forall ((check Int))
                    (=>
                        (and (< check (+ start (length sub))) (>= check start))
                        (= (charOf s check) (charOf sub (- check start))))))))
        (>= i 0)
        (= i (- 1))))
```

In this case, we are looking for a start index such that the substring sub exists in the receiving object s. If such index exists, then $i \geq 0$, else $i = -1$.

Table A.1: Java Implementation Details (part 1)

$s.\text{charAt}(s, i) = c$

definition	$(\text{charAt}(s, i) = c)$

$s.concat(t) = u$

ensures the lengths match	$(\text{length}(u) = (\text{length}(s) + \text{length}(t)))$
ensures that the content of $u = s + t$	$(\forall i$ $(((i \geq 0) \wedge (i < \text{length}(s)))$ $\rightarrow (\text{charAt}(u, i) = \text{charAt}(s, i)))$ $\wedge((i \geq \text{length}(s)) \wedge (i < (\text{length}(s) + \text{length}(t))))$ $\rightarrow (\text{charAt}(u, i) = \text{charAt}(t, (i - \text{length}(s)))))$

$s.\text{contains}(sub) = true$

ensure size constraints of strings	$(\text{length}(s) \geq \text{length}(sub))$
ensures sub is a subset of s	$(\exists j$ $((j \geq 0)$ $\wedge(j < (\text{length}(s) - \text{length}(sub)))$ $\wedge(\forall i$ $(((i < \text{length}(sub)) \wedge (i \geq 0))$ $\rightarrow (\text{charAt}(s, (i + j)) = \text{charAt}(sub, i))))))$

$s.\text{endsWith}(sub) = true$

ensure length constraints of strings	$(\text{length}(s) \geq \text{length}(sub))$
ensures s ends with sub	$(\exists j$ $((j \geq (\text{length}(s) - \text{length}(sub)))$ $\wedge(j < \text{length}(s))$ $\wedge(\forall i$ $(((i < \text{length}(s)) \wedge (i \geq j))$ $\rightarrow (\text{charAt}(s, i) = \text{charAt}(sub, (i - j)))))))$

$s.\text{equals}(t) = b$

ensures s and t are the same object within the solver	$((s = t) \iff b)$
axiom for equality	$((\forall i$ $((i \geq 0)$ $\wedge(i < \text{length}(s)))$ $\rightarrow (\text{charAt}(s, i) = (\text{charAt}(t, i))))$ $\wedge(\text{length}(s) = \text{length}(t)))$ $\iff b)$

Table A.2: Java Implementation Details (part 2)

$s.\,\text{equalsIgnoreCase}(t) = true$

ensure length constraints of strings	$(\text{length}(s) = \text{length}(t))$
converts to lowercase and checks content	$(\forall i$ $((i \geq 0) \wedge (i < \text{length}(s)))$ $\rightarrow (\text{toLower}(\text{charAt}(s, i)) = \text{toLower}(\text{charAt}(t, i))))$

$s.\,\text{indexOf}(c) = i$

ensures $s[i] = c$	$(s.\,\text{charAt}(i) = c)$ $\vee(i = -1)$
ensures i is the first occurrence of c	$(\forall j$ $(((j \geq 0) \wedge (j < i))$ $\rightarrow (\neg(\text{charAt}(s, j) = c)))$
sets bounds on i	$((i \geq 0) \wedge (i < \text{length}(s)))$ $\vee(i = -1)$

$s.\,\text{indexOf}(c, from) = i$

ensures $s[i] = c$	$(s.\,\text{charAt}(i) = c)$ $\vee(i = -1)$
ensures i is the first occurrence of c after $from$	$(\forall j$ $(((j \geq from) \wedge (j < i))$ $\rightarrow (\neg(\text{charAt}(s, j) = c)))$
sets bounds on i	$((i \geq 0) \wedge (i < \text{length}(s)))$ $\vee(i = -1)$
sets bounds on $from$	$((from \geq 0) \wedge (from < \text{length}(s)))$
relationship between bounds	$(i \geq from)$ $\vee(i = -1)$

$s.\,\text{indexOf}(sub) = i$

ensure length constraints	$(\text{length}(s) \geq (\text{length}(sub) + i))$ $\vee(i = -1)$
ensure sub matches s starting at i	$(\forall j$ $(((j \geq i) \wedge (j < (i + \text{length}(sub))))$ $\rightarrow (\text{charAt}(s, j) = \text{charAt}(sub, (j - i)))))$ $\vee(i = -1)$
ensure i is first occurrence of sub	$(\forall j$ $(((j \geq 0) \wedge (j < i))$ $\rightarrow (\exists k$ $((k \geq j)$ $\wedge(k < (j + \text{length}(sub)))$ $\wedge(\neg(\text{charAt}(s, k) = \text{charAt}(sub, (k - j))))))))$
sets bounds on i	$((i \geq 0) \wedge (i < (\text{length}(s) - \text{length}(sub))))$ $\vee(i = -1)$

Table A.3: Java Implementation Details (part 3)

$s.\,\text{indexOf}(sub, from) = i$

ensure length constraints	$(\text{length}(s) \geq (\text{length}(sub) + i))$ $\vee(i = -1)$
ensures sub matches s starting at i	$(\forall j$ $\quad(((j \geq i) \wedge (j < (i + \text{length}(sub))))$ $\quad\quad\rightarrow (\text{charAt}(s, j) = \text{charAt}(sub, (j - i)))))$ $\vee(i = -1)$
ensures i is first occurrence of sub from $from$	$(\forall j$ $\quad(((j \geq from) \wedge (j < i))$ $\quad\quad\rightarrow (\exists k$ $\quad\quad\quad((k \geq j)$ $\quad\quad\quad\wedge(k < (j + \text{length}(sub)))$ $\quad\quad\quad\wedge(\neg(\text{charAt}(s, k) = \text{charAt}(sub, (k - j))))))))$
sets bounds on i	$((i \geq 0) \wedge (i \leq (\text{length}(s) - \text{length}(sub))))$ $\vee(i = -1)$
sets bounds on $from$	$(from \geq 0) \wedge (from < (\text{length}(s) - \text{length}(sub)))$
relationship between bounds	$(from \leq i)$ $\vee(i = -1)$

$s.\,\text{isEmpty}() = b$

length of s is zero	$((\text{length}(s) = 0) \iff b)$

$s.\,\text{lastIndexOf}(c) = i$

ensures $s[i] = c$	$(s.\,\text{charAt}(i) = c)$ $\vee(i = -1)$
ensures i is the last occurrence of c	$(\forall j$ $\quad(((j > i) \wedge (j < \text{length}(s)))$ $\quad\quad\rightarrow (\neg(\text{charAt}(s, j) = c))))$
sets bounds on i	$((i \geq 0) \wedge (i < \text{length}(s)))$ $\vee(i = -1)$

$s.\,\text{lastIndexOf}(c, from) = i$

ensures $s[i] = c$	$(s.\,\text{charAt}(i) = c)$ $\vee(i = -1)$
ensures i is the last occurrence of c before $from$	$(\forall j$ $\quad(((j > i) \wedge (j < \text{length}(s)))$ $\quad\quad\rightarrow (\neg(\text{charAt}(s, j) = c))))$
sets bounds on i	$((i \geq 0) \wedge (i < \text{length}(s)))$ $\vee(i = -1)$
sets bounds on $from$	$(from \geq 0) \wedge (from < \text{length}(s))$
relationship between bounds	$(from \geq i)$

Table A.4: Java Implementation Details (part 4)

$s.\,\text{lastIndexOf}(sub) = i$

ensure i is the last index of sub in s	$(\forall j$ $\quad(((j \geq i) \wedge (j < (i + \text{length}(sub))))$ $\quad\quad\rightarrow (\text{charAt}(s, j) = \text{charAt}(sub, (j - i)))))$ $\vee (i = -1)$
ensure substring is last instance	$(\forall j$ $\quad(((j > i) \wedge (j < \text{length}(s) - 1))$ $\quad\quad\rightarrow (\exists k$ $\quad\quad\quad((k \geq j) \wedge (k < (j + \text{length}(sub)))$ $\quad\quad\quad\wedge(\neg(\text{charAt}(s, k) = \text{charAt}(sub, (k - j))))))))$
sets bounds on i	$((i \geq 0) \wedge (i < (\text{length}(s) - \text{length}(sub))))$ $\vee (i = -1)$

$s.\,\text{lastIndexOf}(sub, from) = i$

ensures i is the last index of sub in s ending before at $from$	$(\forall j$ $\quad(((j \geq i) \wedge (j < (i + \text{length}(sub))))$ $\quad\quad\rightarrow (\text{charAt}(s, j) = \text{charAt}(sub, (j - i)))))$ $\vee (i = -1)$
ensures substring is last instance from $from$	$(\forall j$ $\quad(((j > i) \wedge (j \leq from))$ $\quad\quad\rightarrow (\exists k$ $\quad\quad\quad((k \geq j) \wedge (k < (j + \text{length}(sub)))$ $\quad\quad\quad\wedge(\neg(\text{charAt}(s, k) = \text{charAt}(sub, (k - j))))))))$
sets bounds on i	$((i \geq 0) \wedge (i < (\text{length}(s) - \text{length}(sub))))$ $\vee (i = -1)$
sets bounds on $from$	$(from \geq 0) \wedge (from < \text{length}(s))$
relationship between bounds	$(from \geq i)$

$s.\,\text{length}() = i$

assert length	$(\text{length}(s) = i)$
every index from 0 to i has a character	$(\forall j$ $\quad(((j \geq 0) \wedge (j < i))$ $\quad\quad\rightarrow (\exists c(\text{charAt}(s, j) = c))))$

$s.\,\text{replace}(c, d) = t$

ensure length constraints	$(\text{length}(s) = \text{length}(t))$
ensure all instances of c in s are replaced by d in t	$(\forall i$ $\quad((i \geq 0) \wedge (i < \text{length}(s)))$ $\quad\quad\rightarrow ((\text{charAt}(s, i) = c) \wedge (\text{charAt}(t, i) = d))$ $\quad\quad\vee(\neg(\text{charAt}(s, i) = c)$ $\quad\quad\quad\wedge((\text{charAt}(s, i) = (\text{charAt}(t, i))))$

Table A.5: Java Implementation Details (part 5)

$s.\,\text{startsWith}(sub) = true$

ensure size constraints of strings	$(\text{length}(s) \geq \text{length}(sub))$
ensures s starts with sub	$(\forall i$ $(((i \geq 0) \wedge (i < \text{length}(sub)))$ $\rightarrow (\text{charAt}(s, i) = \text{charAt}(sub, i))))$

$s.\,\text{substring}(i) = sub$

ensures s ends with sub starting at i	$(\forall k$ $(((k \geq i) \wedge (k < (i + \text{length}(sub))))$ $\rightarrow (\text{charAt}(s, k) = \text{charAt}(sub, (k - i)))))$
sets bounds on i	$(i \geq 0) \wedge (i < \text{length}(sub))$
sets constraints on string sizes	$(\text{length}(s) \geq \text{length}(sub))$

$s.\,\text{substring}(i, j) = sub$

ensures s between i and j equals sub	$(\forall k$ $(((k \geq i) \wedge (k < j))$ $\rightarrow (\text{charAt}(s, k) = \text{charAt}(sub, (k - i))))$
sets bounds on i	$(i \geq 0) \wedge (i < j) \wedge (j < \text{length}(s))$
sets constraints on string sizes	$(\text{length}(s) \geq \text{length}(sub)) \wedge ((j - i) = \text{length}(sub))$

$s.\,\text{toUpperCase}() = t$

ensures the lengths match	$(\text{length}(s) = \text{length}(t))$
Make sure that all lower case letters in s are upper case in t	$(\forall i$ $((i \geq 0) \wedge (i < \text{length}(s)))$ $\rightarrow ((((\text{charAt}(s, i) = \text{`a'}) \wedge (\text{charAt}(t, i) = \text{`A'}))$ $\vee((\text{charAt}(s, i) = \text{`b'}) \wedge (\text{charAt}(t, i) = \text{`B'}))$ $\vee((\text{charAt}(s, i) = \text{`c'}) \wedge (\text{charAt}(t, i) = \text{`C'}))$ $\vee \ldots$ $\vee((\text{charAt}(s, i) = \text{`z'}) \wedge (\text{charAt}(t, i) = \text{`Z'})))$ $\vee((\neg(\text{charAt}(s, i) = \text{`a'}))$ $\wedge(\neg(\text{charAt}(s, i) = \text{`b'}))$ $\wedge(\neg(\text{charAt}(s, i) = \text{`c'}))$ $\wedge \ldots$ $\wedge(\neg(\text{charAt}(s, i) = \text{`z'}))$ $\wedge(\text{charAt}(s, i) = \text{charAt}(t, i)))))$

Table A.6: Java Implementation Details (part 6)

$s.\,\text{toLowerCase}() = t$

ensures the lengths match	$(\text{length}(s) = \text{length}(t))$
Make sure that all lower case letters in s are lower case in t	$(\forall i$ $((i \geq 0) \wedge (i < \text{length}(s)))$ $\rightarrow ((((\text{charAt}(s,i) = \text{`A'}) \wedge (\text{charAt}(t,i) = \text{`a'}))$ $\vee((\text{charAt}(s,i) = \text{`B'}) \wedge (\text{charAt}(t,i) = \text{`a'}))$ $\vee((\text{charAt}(s,i) = \text{`C'}) \wedge (\text{charAt}(t,i) = \text{`c'}))$ $\vee \ldots$ $\vee((\text{charAt}(s,i) = \text{`Z'}) \wedge (\text{charAt}(t,i) = \text{`z'})))$ $\vee((\neg(\text{charAt}(s,i) = \text{`A'}))$ $\wedge(\neg(\text{charAt}(s,i) = \text{`B'}))$ $\wedge(\neg(\text{charAt}(s,i) = \text{`C'}))$ $\wedge \ldots$ $\wedge(\neg(\text{charAt}(s,i) = \text{`Z'}))$ $\wedge(\text{charAt}(s,i) = \text{charAt}(t,i)))))$

$s.\,\text{trim}() = t$

ensure length requirement	$(\text{length}(t) \leq \text{length}(s))$
s and t are identical between the whitespaces. j is an offset, where t is identical to s between j and $j + \text{length}(t)$.	$(\exists j$ $(j \geq 0)$ $\wedge(j < (\text{length}(s) - \text{length}(t)))$ $\wedge(\forall i$ $((i \geq 0) \wedge (i < \text{length}(t)))$ $\rightarrow ((\text{charAt}(s,(i+j)) = \text{charAt}(t,i))))$ $\wedge(\forall k$ $(((k \geq 0) \wedge (k < j))$ $\rightarrow (\text{charAt}(s,k) = \text{`~'}))$ $\wedge(((k \geq (j + \text{length}(t))) \wedge (k < \text{length}(s)))$ $\rightarrow (\text{charAt}(s,k) = \text{`~'}))))$
t has no leading or trailing whitespaces	$((\text{length}(t) > 0)$ $\rightarrow (\neg(\text{charAt}(t,0) = \text{`~'})$ $\wedge\neg(\text{charAt}(t,(\text{length}(t)-1)) = \text{`~'})))$

A.2 Yahoo! Pipes

The specifics of the module encodings are shown in Table A.7, Table A.8, and Table A.9 for the supported constructs, listed alphabetically. The $hasRec(L, r)$ and $recordOf(L, i) = r)$ functions are defined at the end of Table A.8. For each module, the constraint type (inclusion, exclusion, order, size) are shown followed by the constraint definition. When types for the variables are not stated, we used the following scheme: i, j, k, and l are integers, s is a string, r, r_1, and r_2 are records, L, in, in_1, in_2, in_3, in_4, in_5, out, out_1, and out_2 are lists. Each row in the tables is implemented as an *assert* statement in SMTLIB2 format.

Table A.7: Yahoo! Pipes Implementation Details (part 1)

fetch*(in)* = *out*

inclusion	$(\forall r$ $(\text{hasRec}(in, r) \rightarrow \text{hasRec}(out, r)))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall i$ $(((i \geq 0) \wedge (i < \text{size}(out)))$ $\rightarrow (\text{recordOf}(in, i) = \text{recordOf}(out, i))))$
size	$(\text{size}(in) = \text{size}(out))$

filter*(in, title, equals, s)* = *out*

inclusion	$(\forall i($ $((i \geq 0) \wedge (i < \text{size}(in)))$ $((getTitle(\text{recordOf}(in, i)) = s)$ $\rightarrow \text{hasRec}(out, r))$ $\wedge((\neg(getTitle(\text{recordOf}(in, i)) = s))$ $\rightarrow \neg(\text{hasRec}(out, r)))))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall r_1, r_2$ $(\text{hasRec}(out, r_1) \wedge \text{hasRec}(out, r_2)$ $\wedge(\exists i, j$ $((i < j)$ $\wedge(\text{recordOf}(out, i) = r_1) \wedge (\text{recordOf}(out, j) = r_2))))$ $\rightarrow (\exists k, l$ $((k < l)$ $\wedge(\text{recordOf}(in, k) = r_1) \wedge (\text{recordOf}(in, l) = r_2))))$
size	$(\text{size}(in) \geq \text{size}(out))$

sort*(in)* = *out*

inclusion	$(\forall r$ $(\text{hasRec}(in, r) \rightarrow \text{hasRec}(out, r)))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall i, j$ $((i < j)$ $\wedge(i \geq 0)$ $\wedge(j < \text{size}(out)))$ $\rightarrow getPubdate(\text{recordOf}(out, i))$ $\leq getPubdate(\text{recordOf}(out, j)))$
size	$(\text{size}(in) = \text{size}(out))$

Table A.8: Yahoo! Pipes Implementation Details (part 2)

tail*(in, n) = out*

inclusion	$(\forall i$ $(((i \geq max(0, \text{size}(in) - n)) \wedge (i < \text{size}(in)))$ $\rightarrow \text{recordOf}(in, i) = \text{recordOf}(out, i)))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall i$ $(((i \geq 0) \wedge (i < \text{size}(out)))$ $\rightarrow \text{recordOf}(in, i + max(0, \text{size}(in) - n)) = \text{recordOf}(out, i)))$
size	$(\text{size}(in) \geq \text{size}(out))$

truncate*(in, n) = out*

inclusion	$(\forall i$ $((i \geq 0) \wedge (i < min(n, \text{size}(in))))$ $\rightarrow \text{recordOf}(in, i) = \text{recordOf}(out, i)))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall i$ $((i \geq 0) \wedge (i < \text{size}(out)))$ $\rightarrow \text{recordOf}(in, i) = \text{recordOf}(out, i)))$
size	$(\text{size}(in) \geq \text{size}(out))$

union$(in_1, in_2, in_3, in_4, in_5) = out$

inclusion	$(\forall r$ $(\text{hasRec}(in_1, r) \vee \text{hasRec}(in_2, r)$ $\vee\ \text{hasRec}(in_3, r) \vee \text{hasRec}(in_4, r) \vee \text{hasRec}(in_5, r)))$ $\rightarrow \text{hasRec}(out, r))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r)$ $\rightarrow \text{hasRec}(in_1, r) \vee \text{hasRec}(in_2, r)$ $\vee \text{hasRec}(in_3, r) \vee \text{hasRec}(in_4, r) \vee \text{hasRec}(in_5, r)))$
order	$(\forall i$ $((i \geq 0) \wedge (i < \text{size}(in_1)))$ $\rightarrow (\text{recordOf}(in_1, i) = \text{recordOf}(out, i))$ $\wedge((i \geq \text{size}(in_1)) \wedge (i < \text{size}(in_1) + \text{size}(in_2))$ $\rightarrow (\text{recordOf}(in_2, i - \text{size}(in_1)) = \text{recordOf}(out, i))$ $\wedge \ldots$
size	$((\text{size}(in_1) + \text{size}(in_2) + \text{size}(in_3) + \text{size}(in_4) + \text{size}(in_5))$ $= \text{size}(out))$

Table A.9: Yahoo! Pipes Implementation Details (part 3)

split$(in) = \{out_1, out_2\}$

inclusion	$(\forall r$ $(\text{hasRec}(in, r)$ $\rightarrow \text{hasRec}(out_1, r) \wedge \text{hasRec}(out_2, r)))$
exclusion	$(\forall r$ $((\text{hasRec}(out_1, r) \vee \text{hasRec}(out_2, r))$ $\rightarrow \text{hasRec}(in, r)))$
order	$(\forall i$ $((i \geq 0) \wedge (i < \text{size}(out_1)))$ $\rightarrow (\text{recordOf}(in, i) = \text{recordOf}(out_1, i))$ $\wedge(\text{recordOf}(in, i) = \text{recordOf}(out_2, i)))$
size	$((\text{size}(in) = \text{size}(out_1)) \wedge (\text{size}(in) = \text{size}(out_2)))$

output$(in) = out$

inclusion	$(\forall r$ $(\text{hasRec}(in, r) \rightarrow \text{hasRec}(out, r)))$
exclusion	$(\forall r$ $(\text{hasRec}(out, r) \rightarrow \text{hasRec}(in, r)))$
order	$(\forall i$ $((i \geq 0) \wedge (i < \text{size}(out)))$ $\rightarrow (\text{recordOf}(in, i) = \text{recordOf}(out, i)))$
size	$(\text{size}(in) = \text{size}(out))$

wire

equality	$(source = destination)$
size	$(\text{size}(\text{destination}(in))) = \text{size}(\text{source}(out)))$

hasRec(L, r)

definition	$(\text{hasRec}(L, r) = true)$ $\iff (\exists i$ $((i \geq 0) \wedge (i < \text{size}(L)) \wedge (\text{recordOf}(L, i) = r)))$

recordOf$(L, i) = r$

definition	$(\text{recordOf}(L, i) = r)$

Appendix B

User Quasi-Experiment and Study on Search Habits

This section provides the user survey, quasi-experiment related to input/output specification, and IRB informed consent for a user study.

B.1 User Survey

Table B.1 reports the survey questions and answers used in the user survey described in Section 3.1. The *Question* column states the question as it appeared to the participants. The *Format* column describes the question format, indicating if it used radio buttons (R, single-selection), check-boxes (C, multiiple-selection), or was open-ended (O). For example, Question 3 asks about the programming experience of participants, providing four options in a radio-button format.

Table B.1: Survey Questions

	Question	Format	Answer
1.	Are you at least 19 years old?	R	(Yes, No)
2.	What is your gender?	R	(Male, Female, Prefer not to answer)
3.	How long have you been programming?	R	(0 years, less than 2 years, 2-5 years, 5+ years)
4.	How often do you program?	R	(never, daily, weekly, monthly)
5.	What are your primary programming languages?	O	
6.	How often do you look for example source code by searching (e.g., using the web, and IDE search feature, grep, etc.)	R	(never, monthly, weekly, daily)
7.	Where do you usually search for sample code?	O	
8.	How do you search for code?	C, O	(function name, keyword, variable name, libraries used, other please specify)
9.	How many snippets of code do you typically examine before finding something useful?	R	(1, 2, 3, 4, 5, 6+)
10.	What do you do with useful code once found?	C,O	(Copy/paste as is, Copy/paste and modify, Just get ideas for implementation, Link to found code, Other please specify)

Format Key: R = radio button, C = checkbox, O = open-ended

B.2 Additional Survey Results

The results for some survey questions in Table B.1 were not reported in Section 3.1. Results for Question 5 are shown in Table B.2. Table 3.4 shows results for Question 7 and Tables B.3 and B.4 shows results for Question 8.

Table B.2: Programming Languages Used by Participants (Question 5)

Language	**Count**	**Percent**
Java	63	57.8%
C	52	47.7%
C++	43	39.4%
SQL	20	18.3%
Excel	18	16.5%
Javascript	17	15.6%
C#	16	14.7%
29 others	≤ 9	< 10 %

Table B.3: Search Methods Used by Participants (Question 8)

Search Method	**Count**	**Percent**
Function Name	49	45%
Keyword	79	72%
Variable Name	18	17%
Libraries Used	38	35%
Other	25	23%

Table B.4: Search Methods Used by Participants - Other (Question 8)

Search Method	**Count**	**Percent**
Functionality or problem description	14	
Language	3	
Algorithm	3	
Error	2	
5 others	1	

B.3 Qualification Test

To control for quality among the Mechanical Turk participants, we included a qualification test that contained competency questions related to Yahoo! Pipes and SQL. Participants were required to answer the competency questions with 50% or greater accuracy, meet the age requirement, and agree to the informed consent (Section B.4) in order to participate in the study.

The following two questions were used to determine participant competency (scoring was performed as described in Section 8.2.5, and the answer is indicated with an x next to the relevant rows):

1. Select the rows **with a price per unit equal to $0.50 from the table below**, using the checkboxes next to each row.

	Id	**Fruit**	**Vendor**	**Price**
x	1	Apple	John's Produce	$0.50
	2	Apple	John's Produce	$0.84
	3	Apple	Fran's Fruit	$1.50
	4	Pear	John's Produce	$0.98
	5	Pear	Pearly Pears	$2.50
x	6	Grapes	Eduardo's Uvas	$0.50
x	7	Bananas	Pato's Plantains	$0.50

2. Select **all items with "mountains" or "Mountains" in the title.**

	Title:	Cool, dry after light rain; wet weekend?
	Description:	After a light weekend rain, Orange County should dry out for the work week, though we could be looking at more rain for the ...
	Link:	http://sciencedude.ocregister.com/2012/01/16/cool-dry-after-light-rain-more-ahead/166663/
	Date:	Mon Jan 16 08:51:00 CST 2012
	Title:	As space junk falls, Russia hints at sabotage
	Description:	A space probe stuck in orbit could fall back to Earth as soon as Sunday or Monday, though most experts say the chance that ...
	Link:	http://sciencedude.ocregister.com/2012/01/13/as-space-junk-falls-russia-hints-at-sabotage/166579/
	Date:	Fri Jan 13 11:49:00 CST 2012
x	Title:	Warm today; red-flag warning for mountains
	Description:	Warm, dry winds will raise wildfire risk in the Santa Ana Mountains Friday, prompting a red-flag warning from the National Weather ...
	Link:	http://sciencedude.blog.ocregister.com/2012/01/13/warm-today-red-flag-warning-for-mountains/166607/
	Date:	Fri Jan 13 08:49:00 CST 2012
x	Title:	Red-flag warning for Santa Ana Mountains
	Description:	The National Weather Service has issued a red-flag wildfire warning for the Santa Ana Mountains from midnight Thursday to 2 p.m. Friday ...
	Link:	http://sciencedude.blog.ocregister.com/2012/01/12/red-flag-fire-warning-for-santa-ana-mountains/166585/
	Date:	Thu Jan 12 18:02:00 CST 2012
	Title:	Ocean Institute to celebrate Marine Protected Areas
	Description:	The Ocean Institute at Dana Point Harbor will join the celebration of the fourth Underwater Parks Day from 10 a.m. to 3 p.m. Jan. 21 ...
	Link:	http://www.ocregister.com/news/marine-335239-aquarium-institute.html
	Date:	Thu Jan 12 15:00:00 CST 2012

.

B.4 Informed Consent IRB# 20120212388 EX

The user survey was conducted with two groups of participants, students and Mechanical Turk participants. As such, two separate informed consent forms were required, but the deviations were small. The IRB protocol for this study includes the following terms and conditions, with differences marked:

Purpose of the Research

The goal of this research is to assess the feasibility of using input/output specifications for demonstrate desired program behavior.

Procedures

(Mechanical Turk) You must be at least 19 years of age to participate in this study. Participation in this study will involve the completion of one or more HITs that will ask you to answer a question regarding a list of items or a table. You will be asked multiple-choice questions about the pipes. This study will be conducted from your personal computer via the Mechanical Turk website.

Each HIT should take no more than 1-5 minutes to complete, though the allotted time by Mechanical Turk allows 10 minutes per HIT. To complete all 10 of the HITs available to you should take no longer than one hour in total.

(Students) You have been asked to participate based on your enrollment in this course, and you must be at least 19 years of age to participate. The assessment will be administered here, today. Participation in this study is voluntary and will involve the completion of a survey regarding your programming practices and the completion of

up to 10 tasks in which you will determine the results of a query on a table or list of data. This assessment will take no more than 15 minutes to complete.

Benefits and Risks

This assessment will provide you with practice in analyzing queries on tables and/or lists. There are no known risks or discomforts.

Compensation

(Mechanical Turk only) For each HIT completed (up to 10), you will receive compensation of $0.26, for a potential total compensation of $2.60. If you choose not to complete any HITs, you will not receive any monetary compensation.

Confidentiality

(Mechanical Turk) Your answers will be strictly confidential and will not be connected to your name, email, IP address, or any other identifying information.

(Students) Your answers will be strictly confidential and will not be connected to your name or made available to your instructor.

Use of Data

The data collected will be aggregated and used for research in the area of software engineering, so if you decide to participate we would appreciate your effort in completing the assigned task to the best of your abilities. The data will be summarized across all participants (i.e., we will not be able to identify you from your data). Results will be reported in conference proceedings, journals, and presentations at software engineering conferences and workshops.

Opportunity to Ask Question

You may ask any questions concerning this research and have those questions answered before agreeing to participate in or during the study. The investigators for this research are Dr. Elbaum and Ms. Stolee; their respective email addresses are 'elbaum at cse.unl.edu' and 'kstolee at cse.unl.edu'. If you have questions concerning your rights as a research subject that have not been answered by the investigator or to report any concerns about the study, you many contact the University of Nebraska-Lincoln Institutional Review Board, telephone (402) 472-6965 or e-mail 'irb at unl.edu'.

Freedom to Withdraw

(Mechanical Turk) You are free to decide not to participate in this study or to withdraw at any time without adversely affecting your relationship with the investigators or the University of Nebraska. Your decision will not result in any loss or benefits to which you are otherwise entitled, but compensation is only provided once HITs have been completed.

(Students) You are free to opt-out at any time without adversely affecting your relationship with the investigators, your instructor, or UNL.

Consent, Right to Receive a Copy

Your acceptance certifies that you have decided to participate having read and understood the information presented. You may save this page for your records.

Affiliations

We are researchers in the ESQuaReD lab at the University of Nebraska-Lincoln.

Appendix C

Search Results Study

C.1 Qualification Test

The comprehensive section of the qualification test used two code snippets and asked multiple-choice questions about the behavior of the source code. The questions were as follows:

Answer the next two questions considering the following Java method:

```
public static boolean foo(String s, char c) {
    if (s.indexOf(c) != -1) {
        return true;
    } else {
        return false;
    }
}
```

1. Identify the output (return value) for the following method invocation:

   ```
   foo("hello", 'x');
   ```

   a) True

   b) False

c) Null

2. Identify the method invocation that yields the following output (return value):

```
true
```

a) `foo("I drive", 'a');`

b) `foo("I drive a car", 'a');`

c) `foo("running is better", 'a');`

Answer the next two questions considering the following Java method:

```
public static String bar(String s, int i){
    if(s.length() > i) {
        return s.substring(0, i);
    }else {
        int diff = i - s.length();
        for(int j = 0; j < diff; j++) {
            s = s + '*';
        }
    }
    return s;
}
```

1. Identify the output (return value) for the following method invocation:

```
bar("hello", 7);
```

a) hello

b) hello**

c) he

d) Null

2. Identify the method invocation that yields the following output (return value):

```
he
```

a) `bar("hello", 7);`

b) `bar("help", 7);`

c) `bar("hello", 3);`

d) `bar("help", 2);`

C.2 Informed Consent IRB# 20130513551 EX

The IRB protocol for this study includes the following terms and conditions:

Purpose of the Research

The goal of this research is to measure the relevance of search results from three search techniques for program source code: descriptive, signature, and semantic.

Procedures

You must be at least 19 years of age to participate in this study. Participation in this study will involve the completion of one or more HITs that will ask you to answer a question about the relevance of source code to a programming task. You will be asked to answer multiple-choice questions about the source code and also to justify your answers. This study will be conducted from your personal computer via the Mechanical Turk website.

Each HIT should take no more than 3-5 minutes to complete, though the allotted time by Mechanical Turk is 20 minutes per HIT. To complete all 22 of the HITs available to you should take no longer than two hours in total.

Risks and/or Discomforts

There are no known risks or discomforts

Benefits

This assessment will provide you with practice in assessing the relevance of source code for a particular goal.

Compensation

For each HIT satisfactorily completed (up to 22), you will receive compensation of $0.20, for a potential total compensation of $4.40. If you choose not to complete any HITs, you will not receive any monetary compensation.

Confidentiality

Your answers will be strictly confidential and will not be connected to your name, email, IP address, or any other identifying information. The data will be summarized across all participants and results will be reported in conference proceedings, journals, dissertations, and presentations at software engineering conferences and workshops.

Opportunity to Ask Questions

You may ask any questions concerning this research and have those questions answered before agreeing to participate in or during the study. The investigators for this research are Dr. Elbaum and Ms. Stolee; their respective email addresses are 'elbaum at cse.unl.edu' and 'kstolee at cse.unl.edu'. If you have questions concerning your rights as a research subject that have not been answered by the investigator or to report any concerns about the study, you many contact the University of Nebraska-Lincoln Institutional Review Board, telephone (402) 472-6965 or e-mail 'irb at unl.edu'.

Freedom to Withdraw

You are free to decide not to participate in this study or to withdraw at any time without adversely affecting your relationship with the investigators or the University of Nebraska. Your decision will not result in any loss or benefits to which you are otherwise entitled, but compensation is only provided once HITs have been completed.

Consent, Right to Receive a Copy

Your acceptance certifies that you have decided to participate having read and understood the information presented. You may save this page for your records.

Affiliations

We are researchers in the ESQuaReD lab at the University of Nebraska-Lincoln.

C.3 R Analysis Details

In the R analysis, we read in a file that contains summary information, with a column for each of the factors, Problem and Search (both categorical), a column for the Generators (also categorical, these are the 'subjects' in the design), and the P@10 and Q@10 metrics in the P and Q columns, respectively.

```
dataagg <-
    read.csv(file="phdthesis/dissertation/mturkResultsForRagg.csv",head=TRUE,sep=",")
summary(dataagg)
     Problem     Generator   Search      Snippet         P                Q
 eight   : 9   g      : 8   goog:24   Min.   :1   Min.   :0.1000   Min.   :0.0000
 eleven  : 9   e      : 7   mero:24   1st Qu.:1   1st Qu.:0.3000   1st Qu.:0.0000
 five    : 9   j      : 7   mine:24   Median :1   Median :0.5000   Median :0.2000
 seven   : 9   a      : 6             Mean   :1   Mean   :0.5278   Mean   :0.2153
 six     : 9   b      : 6             3rd Qu.:1   3rd Qu.:0.8000   3rd Qu.:0.3000
 thirteen: 9   c      : 6             Max.   :1   Max.   :1.0000   Max.   :0.8000
 (Other) :18   (Other):32
> summary(Generator)
a b c d e f g h i j k l
6 6 6 6 7 6 8 6 6 7 5 3
> summary(Problem)
    eight    eleven      five     seven       six  thirteen    twenty twentyone
        9         9         9         9         9         9         9         9
```

We observe a strong correlation between P@10 and Q@10, so in the ANOVA we consider P@10 as the response variable. Search and Problem are the factors in this 2x2 factorial design.

```
> cor(P, Q, method="spearman")
[1] 0.718138
> summary(aov(P~Search*Problem, data=dataagg))
               Df  Sum Sq Mean Sq F value  Pr(>F)
Search          2 1.08111 0.54056 11.6527 7.496e-05 ***
Problem         7 0.52444 0.07492  1.6151     0.1540
Search:Problem 14 0.99222 0.07087  1.5278     0.1373
Residuals      48 2.22667 0.04639
---
Signif. codes:  0  ***  0.001  **  0.01  *  0.05  .  0.1     1
```

The hypothesis tests were conducted using the Mann-Whitney non-parametric test of means:

```
> wilcox.test(P[Search=="goog"], P[Search=="mine"])
```

```
        Wilcoxon rank sum test with continuity correction

data:  P[Search == "goog"] and P[Search == "mine"]
W = 383.5, p-value = 0.04751
alternative hypothesis: true location shift is not equal to 0

> wilcox.test(P[Search=="goog"], P[Search=="mero"])

        Wilcoxon rank sum test with continuity correction

data:  P[Search == "goog"] and P[Search == "mero"]
W = 470.5, p-value = 0.0001498
alternative hypothesis: true location shift is not equal to 0

> wilcox.test(P[Search=="mine"], P[Search=="mero"])

        Wilcoxon rank sum test with continuity correction

data:  P[Search == "mine"] and P[Search == "mero"]
W = 391, p-value = 0.0326
alternative hypothesis: true location shift is not equal to 0
```

Took the average number of 'yeses' that were answered over time for each of the basic tasks. Turns out the slope coefficient is very low, 0.01739, indicating no linear relationship between a participants' likeness to answer 'yes' and the order of the basic tasks.

```
> order=1:24
> rel = c (13,15,15,19,15,18,18,12,12,17,13,19,21,16,15,15,18,16,17,16,12,17,17,14)
> plot(order, rel)
> r = lm(formula=rel~order)
> r

Call:
lm(formula = rel ~ order)

Coefficients:
(Intercept)        order
   15.61594      0.01739
```

Bibliography

[1] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[2] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, November 1985.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

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

[5] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 307–321, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[7] Lori A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, SE-2(3):215 – 222, sept. 1976.

[8] Lori A. Clarke and Debra J. Richardson. Applications of symbolic evaluation. *J. Syst. Softw.*, 5(1):15–35, February 1985.

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

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

[11] CVC3. http://www.cs.nyu.edu/acsys/cvc3/, June 2013.

[12] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on*

Foundations of software engineering, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[15] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12:17–26, November 1995.

[16] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 1135–1139, New York, NY, USA, 2003. ACM.

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

[18] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM.

[19] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. Exemplar: Executable examples archive. In *International Conference on Software Engineering*, pages 259–262, 2010.

[20] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 119–130, New York, NY, USA, 2010. ACM.

[21] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium*

on Principles of programming languages, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[22] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 50–61, New York, NY, USA, 2011. ACM.

[23] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. *SIGPLAN Not.*, 46(6):317–328, June 2011.

[24] Abram Hindle and Daniel M. German. Scql: a formal model and a query language for source control repositories. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[25] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[26] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, December 2006.

[27] Jeff Huang and Efthimis N. Efthimiadis. Analyzing and evaluating query reformulation strategies in web search logs. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 77–86, New York, NY, USA, 2009. ACM.

[28] Java Language and Virtual Machine Specifications. http://docs.oracle.com/javase/specs/, June 2013.

[29] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[30] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[31] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

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

[33] Symbolic PathFinder, December 2012.

[34] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[35] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *International symposium on Software testing and analysis*, ISSTA '09, pages 105–116, 2009.

[36] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

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

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

[39] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 917–918, New York, NY, USA, 2007. ACM.

[40] A Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, PhD thesis, EECS Department, University of California, Berkeley, 2008.

[41] Henry Lieberman. Watch what i do. chapter Tinker: a programming by demonstration system for beginning programmers, pages 49–64. MIT Press, Cambridge, MA, USA, 1993.

[42] Greg Little and Robert C. Miller. Keyword programming in java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 84–93, New York, NY, USA, 2007. ACM.

[43] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

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

[45] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, 2012. ACM.

[46] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2004.

[47] Ohloh Code Search. http://code.ohloh.net/, September 2012.

[48] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6, April 1999.

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

[50] Andy Podgurski and Lynn Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering Methodology*, 2, July 1993.

[51] Andreas Raabe and Rastislav Bodik. Synthesizing hardware from sketches. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 623–624, New York, NY, USA, 2009. ACM.

[52] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 139–148, New York, NY, USA, 2012. ACM.

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

[54] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, June 2010.

[55] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. *SIGPLAN Not.*, 41(10):413–430, October 2006.

[56] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[57] Nicholas Sawadsky, Gail C. Murphy, and Rahul Jiresal. Reverb: recommending code-related web pages. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 812–821, Piscataway, NJ, USA, 2013. IEEE Press.

[58] Christopher Scaffidi, Brad Myers, and Mary Shaw. Topes: reusable abstractions for validating data. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[59] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering Methodology*, 21(1):4:1–4:25, December 2011.

[60] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 289–299, New York, NY, USA, 2011. ACM.

[61] SMT-LIB, December 2012.

[62] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. *SIGPLAN Not.*, 42(6):167–178, June 2007.

[63] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 136–148, New York, NY, USA, 2008. ACM.

[64] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 404–415, New York, NY, USA, 2006. ACM.

[65] SQL Language Reference. http://docs.oracle.com/cd/B28359_-01/server.111/b28286/toc.htm, June 2013.

[66] Kathryn T. Stolee. Analysis and Transformation of Pipe-like Web Mashups for End User Programmers. Master's Thesis, University of Nebraska–Lincoln, June 2010.

[67] Kathryn T. Stolee. Finding suitable programs: Semantic search with incomplete and lightweight specifications. In *International conference on Software engineering Doctoral Symposium*, pages 1571–1574, 2012.

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

[69] Kathryn T. Stolee and Sebastian Elbaum. Toward semantic search via smt solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 25:1–25:4, New York, NY, USA, 2012. ACM.

[70] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. Technical report, University of Nebraska-Lincoln, November 2012.

[71] Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma. Discovering how end-user programmers and their communities use public repositories: a study on yahoo! pipes. *Information and Software Technology*, 2012.

[72] Jan Strejček and Marek Trtík. Abstracting path conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 155–165, New York, NY, USA, 2012. ACM.

[73] TIOBE Programming Community Index for June 2013. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, June 2013.

[74] 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.

[75] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

[76] Ian H. Witten and Dan Mo. Watch what i do. chapter TELS: learning text editing tasks from examples, pages 183–203. MIT Press, Cambridge, MA, USA, 1993.

[77] Jeffrey Wong and Jason Hong. What Do We "Mashup" When We Make Mashups? In *International Workshop on End-User Software Engineering*, 2008.

[78] The Yices SMT Solver. http://yices.csl.sri.com/, June 2013.

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

[80] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146–170, April 1995.

[81] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering Methodology*, 6, October 1997.