CAREER: On the Foundations of Semantic Code Search

PI: Kathryn Stolee

A. Project Summary

Overview: Programmers frequently search for code to support their development activities, including debugging, reuse, and design. Current search tools support textual queries with some filtering, but ignore a critical component of source code: its behavior. Semantic code search finds code based on behavior, often described using input/output examples. The engine behind one approach to semantic search is a constraint solver; code fragments (e.g., methods, blocks) are indexed using symbolic analysis to obtain a constraint representation of the code behavior. Given input/output examples and constraints for a code fragment, the solver returns `sat` if the code satisfies the specification. This approach combines search and synthesis, allowing the solver to modify code as needed to match the specification. Previous approaches to semantic code search either cannot find approximate matches or require the use of formal specifications.

The PI has evaluated semantic search via constraint solver in the contexts of code reuse in Java and program repair in C. While the results have been promising, some fundamental challenges have emerged. Specifically, 1) *finding close matches* when an exact match does not exist; 2) *using more expressive queries*; 3) *characterizing the differences* between two code fragments; and 4) *navigating a solution space* of search results. The PI proposes to perform basic research in each area, exploring techniques to relax program encodings, more precisely characterize desired code, determine how code fragments differ in behavior, and quickly navigate a potentially large solution space. To begin, the PI will evaluate these techniques with end-user programmers who work with languages such as Excel, VBA, MySQL, and Python, and with fewer development support tools. At the end, the PI will generalize the most successful techniques to languages for professional programmers, such as Java and C#.

This proposal lays the foundation for the PI's research and mentoring visions. Her research vision is to bring the benefits and power of semantic analysis to software engineering for all programmers. Instead of searching for code based on syntactic features, behavior-driven search will promote more effective code reuse. Her mentoring vision is to create an environment of inclusion and community for female students in computer science through small-scale but deeply personal mentoring interactions. She plans to scale the impact by sharing experiences with other faculty and running mentorship sessions at academic conferences.

Intellectual Merit: The proposed work has four expected intellectual merit contributions:

1. Models and approximate encodings for source code to amplify the search space during semantic code search, making semantic code search more powerful with limited repository sizes.
2. Expressive query formats to augment input/output examples and more precisely describe desired code.
3. Techniques to characterize behavioral differences between code fragments, including proving equivalence independent of language, identifying differences between fragments, and characterizing similarity.
4. Approaches to navigate the solution space after semantic search or synthesis to quickly converge on a desired solution, including human-in-the-loop and human-out-of-the-loop approaches.

Broader Impact: The proposed work has three expected broader impact contributions:

1. The resulting techniques will have a significant impact on the millions of end-user programmers and professional programmers, allowing them to more effectively reuse code.
2. The PI will create a graduate course, *semantic analysis for software engineering*, to disseminate and develop techniques in this proposal and train students as future research assistants and professionals.
3. The PI will lay the groundwork for a career-long mentoring program for women in computer science. Building on her track record in working with female undergraduate and graduate research assistants and attending the Grace Hopper conference with university students, the PI will develop a structure to support her mentoring of women in computer science. She plans to add 1-2 mentees per year, schedule activities to promote community, and scale by organizing mentoring sessions at academic conferences.

CAREER: On the Foundations of Semantic Code Search

PI: Kathryn Stolee

C. Project Description

1 Introduction

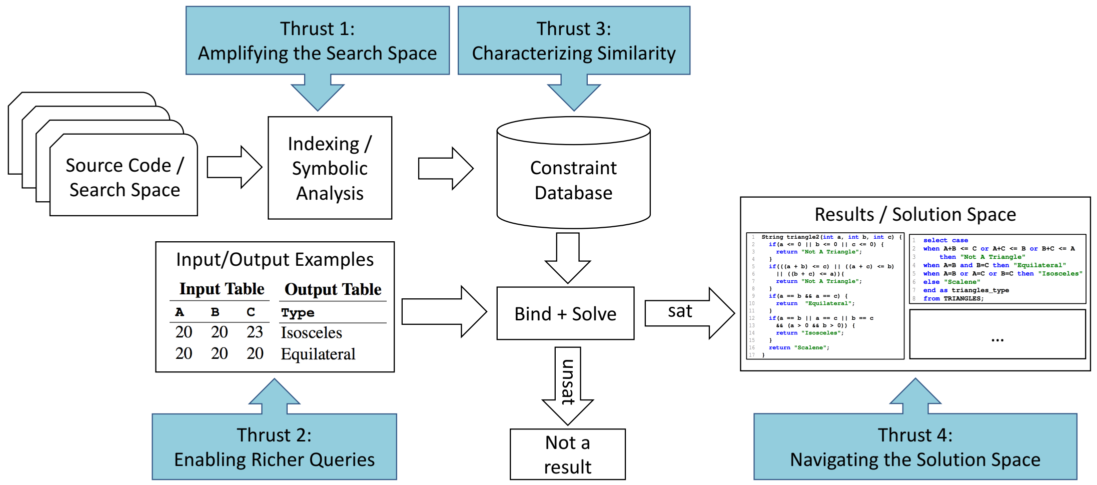
Programmers frequently search for code during software development activities, including during debugging, design, or reuse [2, 68]. In code search, a specification or query describes the desired code. *Syntactic* code search uses a textual query and matches based on syntactic features such as keywords and variable names. In contrast, *semantic* search uses behavioral properties as queries. For example, a syntactic query to find code that computes triangle types could be: "`Type of Triangle in MYSQL`", whereas a behavioral query could be an example input, `{20, 20, 23}` and output, "`isosceles`", to illustrate the desired behavior.

Semantic code search can be achieved through executing source code [65] or mapping source code into constraints representing its behavior and using a constraint solver to identify matches for a query [37, 82, 84, 87, 88]. Results are methods or code fragments that all behave as specified. Figure 1 illustrates a high-level architecture of semantic code search via constraint solver. A *constraint database* is built by *symbolic analysis* over *source code*; this is the *indexing* phase of search. Given a query in the form of *input/output examples*, the search algorithm *binds* the variables in a code fragment to the input/output examples. Then, an SMT *solver* will determine if the code satisfies the specification. A response of `sat` means it is a result, and `unsat` or `unknown` means it is not a result. One advantage of using the solver instead of executing code is the ability to return non-executable code fragments and code that has been modified to satisfy the query.

Semantic code search via constraint solver has been explored for two applications: code reuse [65, 82, 84, 87, 88] and program repair [37]. Semantic search has targeted end-user programming languages [84, 87] and professional programming languages [37, 65, 87, 88]. Even though end-user programmers lack formal training in programming [5, 39], they have similarities to professional programmers in their development activities, especially concerning search. All programmers: search for code to reuse [41, 68], choose which languages and APIs to use [41, 67], and maintain their code over time [58, 85]. As the population of end-user programmers was estimated to be 90 million in 2012 [72], together with the population of professional programmers, the potential for impact in supporting their development activities is large. As these populations generally use different languages, they are treated separately in this proposal.

I propose to build on my prior work in semantic code search via constraint solver [11, 37, 82, 84, 87, 88] and my expertise in end-user programming [27, 32, 42, 83, 85, 89–92]. In all applications of semantic code search, fundamental challenges arise when 1) the desired code does not exist; 2) the input/output query is not expressive enough; 3) it is difficult to differentiate between similar snippets; and 4) there are too many results to navigate efficiently. To address these challenges, **I propose to 1) find approximate solutions to semantic queries; 2) enable richer query models; 3) use the constraints to characterize the differences and similarities in behavior between code snippets; and 4) efficiently navigate the space of potential solutions**. This proposal describes four research thrusts that target these fundamental challenges in semantic code search. Without these techniques, semantic code search will often fail to find code that meets programmers' needs, either because the desired code does not exist (Thrust 1), is hard to describe precisely (Thrust 2), is hard to distinguish from other code (Thrust 3), or is hard to find (Thrust 4). By starting with end-user programmers who typically use smaller languages, we can quickly prototype the techniques (years 1–4) and lay the groundwork to generalize to professional programming languages (year 5).

Thrust 1: Amplifying the Search Space: For when there is not enough diversity in programs to find a match, abstractions and mutations can expand the search space. **I propose to develop techniques to amplify the search space by approximating code semantics, abstracting program behavior, and mutating programs.** Such innovations take advantage of the constraint-based representation of source code in semantic code search and use the solver to identify changes that can lead to desired code.



Input Table			Output Table
A	B	C	Type
20	20	23	Isosceles
20	20	20	Equilateral

```
String triangle2(int a, int b, int c) {
  if(a <= 0 || b <= 0 || c <= 0) {
    return "Not A Triangle";
  }
  if(((a + b) <= c) || ((a + c) <= b)
    || ((b + c) <= a)){
    return "Not A Triangle";
  }
  if(a == b && a == c) {
    return  "Equilateral";
  }
  if(a == b || a == c || b == c
    && (a > 0 && b > 0)) {
    return "Isosceles";
  }
  return "Scalene";
}
```

```
select case
when A+B <= C or A+C <= B or B+C <= A
    then "Not A Triangle"
when A=B and B=C then "Equilateral"
when A=B or A=C or B=C then "Isosceles"
else "Scalene"
end as triangles_type
from TRIANGLES;
```

Figure 1: The current state of semantic code search (white boxes). Shaded boxes with arrows represent the research thrusts in this proposal. Building the database for search involves indexing *source code* through *symbolic analysis*. Thrust 1 *amplifies the search space* through abstraction. Given a *constraint database* and *input/output example(s)*, an SMT *solver* determines which code satisfies the specification to form the *solution space*. Thrust 2 *enables richer specifications* than input/output examples. Thrust 3 involves *characterizing behavioral similarity* between code fragments in the database, which is used in search and other applications. Thrust 4 helps *navigate the solution space*. The example shown comes from StackOverflow.

Thrust 2: Enabling Richer Specifications: Input/output examples are a weak specification. Hence they are attractive and accessible: developers think in terms of input/output when specifying issues on StackOverflow [87, 88, 99]. Yet, there are many solutions that satisfy weak input/output specifications. **I propose to expand the query models supported by semantic code search**, enabling programmers to more precisely describe the code they want without requiring use of formal specifications [19, 62, 102].

Thrust 3: Characterizing Semantic Similarity: One barrier for programmers is the selection barrier [41], suggesting that assistance is needed in choosing between solutions. **I propose to use the constraint representations of code snippets to characterize program similarity and differences**, which will enable characterization of behavioral differences between two solutions. It will also facilitate refactoring verification and cross-language clone detection, which is of interest to my industrial partner, ABB.

Thrust 4: Navigating the Solution Space: When there are too many solutions that match a specification, techniques are needed to organize the solution space [75]. **I propose to develop techniques to rapidly prune the solution space by identifying inputs that maximally fragment the space.** These techniques are also applicable to research in program synthesis by input/output, which abounds [21, 75, 76, 99].

This proposal lays the foundation for semantic code search to be applied in broader contexts and to a wide range of developers. Addressing the fundamental challenges facing semantic code search will extend the applications that benefit from search to also include cross-language clone detection and refactoring verification (Thrust 3). It will allow more precise code specification (Thrust 2) and make semantic search more efficient (Thrust 4). It will enable semantic code search to operate in a constrained environment with low quantities of existing code (Thrust 1). Finally, it is aligned with my overall career goals of making programming easier and more accessible, and bringing the benefits of software engineering analyses and techniques to the millions of end-user programmers (years 1–4) and professional programmers (year 5).

This proposal also lays the foundation for a mentoring program for women in computer science. I currently mentor two female graduate students. Each semester, I also reach out to 1-2 women in the undergraduate and graduate programs at my university. We meet in person, discuss their goals, and check in over e-mail throughout their careers. I plan to define a structure around these efforts and include scheduled monthly meetings and a summer retreat. As this form of mentoring is deeply personal, I intend to keep the group small, adding 1-2 students each year. I will share my experiences with faculty at my university during department lunches. In the research community, I piloted an NSF-sponsored mentoring program at FSE 2016 and will continue such efforts.

```java
int at = full.indexOf('@');
int dot = full.indexOf('.', at);
String domain
  = full.substring(at+1, dot);
```

(a) Java snippet

```sql
select (SUBSTRING_INDEX(
  SUBSTR(email,
    INSTR(email, '@') + 1),'.',1))
```

(b) MySQL snippet from StackOverflow

```
 4  (assert (OR
 5    (full.charAt(at) = '@') (at = (-1))))
 6  (assert (OR
 7    (full.charAt(dot) = '.') (dot = (-1))))
 8  (assert (< (+ at 1) dot))
 9  (assert (forall ((i Int)) (=> (at < i < dot)
10    (and (not (full.charAt(i) = '@'))
11         (not (full.charAt(i) = '.'))))
12  (assert (forall ((i Int)) (=> (at < i < dot)
13    (domain.charAt(i) = full.charAt(i)))))
```

(c) Constraints for the snippets. Lines 4–7 identify the index of the characters of interest, `'@'` and `'.'`. Line 8 ensures the `'@'` comes first. Lines 9–13 ensure that the domain is the same in `domain` and `full`.

Figure 2: Example constraint-based encoding for extracting the domain from an email address

2 Preliminaries: Semantic Code Search

Semantic code search finds code described by a behavioral specification [84,87]. As illustrated in Figure 1, it currently uses input/output examples as queries, indexes code snippets using constraints to represent behavior, and matches code to queries using a constraint solver. For indexing, symbolic execution pre-processes source code snippets (i.e., at the method level or below) into behavioral constraints. For example, consider the code and constraints in Figure 2 for a StackOverflow question about extracting the domain from an email address[1]. Figure 2(a) presents a Java solution, Figure 2(b) presents a MySQL solution (from StackOverflow), and Figure 2(c) presents the constraints encoding the behavior. Since the snippets have the same behavior, in a constraint representation, they would be the same except for naming conventions. Each translated snippet is stored in a database (i.e., *Constraint Database* in Figure 1).

The database content depends on the application. For reuse [87, 88], we scraped GitHub projects for Java code snippets. For program repair [37], we built the database using the code for the program under repair. For Yahoo! Pipes, we scraped the community repository for publicly available programs [87,90,91]. Building a multi-lingual database is proposed to support cross-language clone detection (Thrust 3).

Once the database is built, given an input/output query, such as "`sporty@spice.uk`" and "`spice`", the search engine will first identify all snippets that take a string as input and a string as output; Figure 2(a) and Figure 2(b) would be among the set of snippets to check. It translates the input and output into constraints with assigned variables and binds those variables to the snippets' constraints (e.g., *bind* in Figure 1). With the example in Figure 2(c), the input could be bound to `full` and the output to `domain`. Together with the program encoding, the input/output constraints and bindings are given to a constraint solver; current implementations use the Z3 Satisfiability Module Theory (SMT) solver [12]. A result of `sat` indicates the code behaves as specified; `unsat` implies it does not. A response of `unknown` is assumed to not match.

When a method has multiple input values of the same type, the assignment of input values to parameters is decided by the solver [87]. Thus, parameter ordering in the specification is not an issue, as it

[1] https://stackoverflow.com/questions/41975445/get-domain-name-from-the-email-list

is with search that involves execution [65]. Consider an input {`5, 3, 4`}, output **`true`**, and a method **`def isSorted(a, b, c)`**, which returns **`true`** if $a \leq b \leq c$ and **`false`** otherwise. Semantic code search encodes all possible mappings of the input values to the method arguments (*bind* in Figure 1) and lets the solver select one that works. Six mappings are encoded: $(a = 5 \wedge b = 3 \wedge c = 4) \vee (a = 5 \wedge b = 4 \wedge c = 3) \vee (a = 3 \wedge b = 5 \wedge c = 4) \vee (a = 3 \wedge b = 4 \wedge c = 5) \vee (a = 4 \wedge b = 5 \wedge c = 3) \vee (a = 4 \wedge b = 3 \wedge c = 5)$. Only one mapping where $a = 3 \wedge b = 4 \wedge c = 5$ returns **`sat`**. Instead of executing the method up to six times to find a correct parameter ordering, we make one solver call to find a combination that works.

Originally implemented for Yahoo! Pipes, a (now deprecated) web mashup language, further efforts evolved semantic code search to subsets of Java [87, 88] and MySQL [87], and C [37]. Current language support in Java encodes methods with the following datatypes: **`int`**, **`float`**, **`String`**, **`char`**, and **`boolean`**; library: **`java.lang.String`**; and constructs: if-statements, arithmetic and multiplicative operators [88]. For program repair in C, SearchRepair [37] supports all C primitives, structs, console output, **`char*`** variables, and the string library functions: **`isdigit`**, **`islower`**, **`isupper`**, **`strcmp`**, and **`strncmp`**. For MySQL, the current implementation supports simple **`SELECT`** statements [87]. As Yahoo! Pipes is a deprecated platform, it is omitted from this proposal, but language support included filtering, sorting, and combining RSS feeds [87].

Semantic search is dependent on symbolic analysis and benefits directly from advances in symbolic execution. For Java, SPF [34, 61, 98] is the state-of-the-art tool, and recent efforts aim to improve, for example, heap summaries [28, 29] and efficiency through statistical probabilities [4, 8, 15, 54]. For C, KLEE [6] is the state-of-the-art and recent efforts have improved, for example, performance [66] and the underlying theories, such as arrays [13]. While symbolic execution continues to mature and semantic code search benefits from these efforts, approaches to dealing with language analysis limitations are needed. Current solutions include the use of only subsets of a language or to select languages that are smaller and less expressive than Java and C, such as Yahoo! Pipes (already explored) and MySQL (lightly explored), or languages such as Excel, Visual Basic for Applications (VBA), or Python (all yet to be explored). Using smaller or less supported languages removes the convenience associated with having existing analysis tools. However, writing code to transform programs into constraints for smaller languages is achievable. In fact, this is where semantic code search started, with a custom translator to map Yahoo! Pipes data flow programs into constraints for an SMT solver, effectively creating a symbolic execution engine for a small language.

3 Proposed Research

My broad research vision is to bring the benefits of semantic analysis to all programmers to facilitate informed code changes, promote reuse and adaptation of high-quality artifacts, and reduce developer effort.

This proposal focuses on all programmers. For simplicity, we begin with low levels of granularity, such as methods or functions, and end-user programmers (years 1–4). Scaling to class, sub-system, and system levels, and to professional programming languages, is within the longer-term vision of this work (year 5).

The targeted end-user programming languages are Excel, VBA, MySQL and Python. A small subset of MySQL is supported already, but more indicative of its potential is recent research in program synthesis of MySQL queries [99]. Semantic search can return any code that can be synthesized, however, synthesis tends to overfit to a specification [77], a problem that does not plague semantic search [37, 88]. Excel formulas and VBA functions for spreadsheets reference cell location, providing unique opportunities for abstraction. Python's dynamic typing creates opportunities for abstraction in the type encoding (e.g., since a variable could be an integer or a string). In year 5, we will extend support for semantic code search to professional programmers using languages such as C/C++, C#, and Java, and have some evidence that it is realistic based on the application of semantic code search to program repair of C programs [37].

3.1 Thrust 1: Amplifying the Search Space

Searching for code to reuse in an open-source environment takes advantage of the quantity and variety of code available. However, for end-user programming languages, large repositories of code may not exist at

Input (formatted as Time): 11:00:15   Output (formatted as Text): 11:00:15 AM

(a) Input from the question; output based on highest voted answer. Input is in spreadsheet cell `[A1]`.

```
1  =TEXT(A1, "h:mm:ss")

2  =TEXT(A1, "h:mm:ss AM/PM") //

3  =TEXT(A1, "hh:mm:ss AM/PM")

4  =CONCATENATE(TEXT(A1,"hh:mm:ss"),
5      IF(A1>="12:00:00"," PM", " AM")) //
```

```
6  Function GetMyTimeField()
7      Dim myTime As Date
8      Dim myStrTime As String
9      myTime = [A1]
10     myStrTime = Format(myTime, "hh:mm")
11     myStrTime = myStrTime & " Nice!"
12     GetMyTimeField = myStrTime
13 End Function
```

(b) Three solutions from StackOverflow answers and two solutions added for illustration, marked with //. Four use built-in functions and one uses VBA

Figure 3: Excel Example for Type Conversion with Formatting, Inspired by a StackOverflow post

all [39]. In an academic setting, it may be desirable for a student to limit reuse to code written by them. In an industrial setting, it may be desirable to limit code reuse to within an organization. In these scenarios, the quantity and variety of code is more limited, decreasing the size of the constraint database, and increasing the likelihood that the desired code does not exist. However, semantically close code may be available.

Finding approximate matches requires a notion of similarity between code and specification to determine how well code meets a specification (which is notably different than similarity between two pieces of code, explored in Thrust 3). A *semantic match* means the code satisfies the specification. *Semantic similarity* means that *with some modification*, the code satisfies the specification.

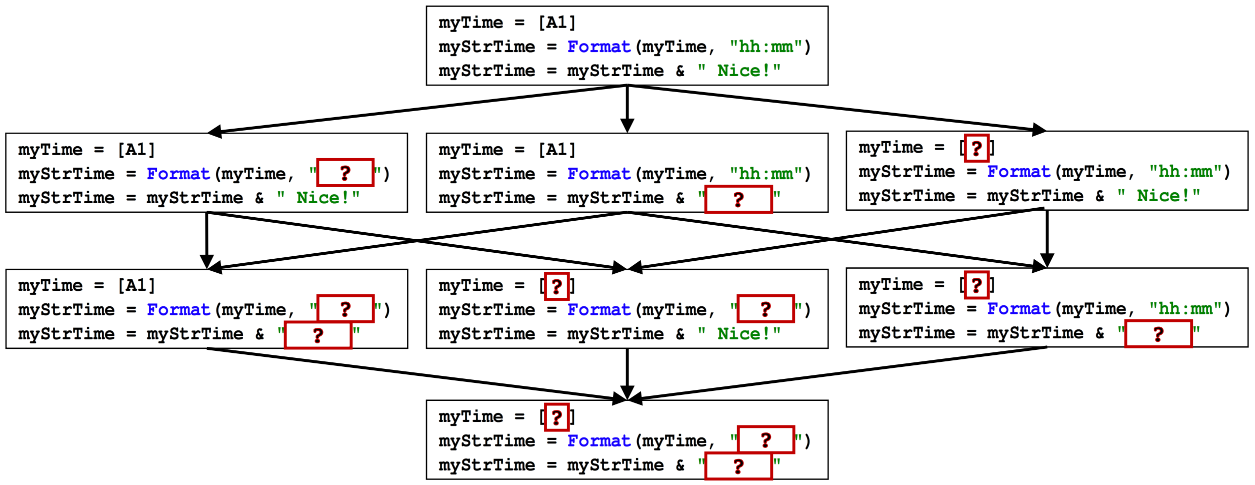
For example, Figure 3 illustrates a StackOverflow question[2] about converting the contents of a spreadsheet cell, formatted as time, to text. The input provided is shown in Figure 3(a). Since an output was not provided, it was generated based on the highest-voted answer. The community proposed the answers starting on lines 1, 3, and 6 in Figure 3. The other two, starting on lines 2 and 4, were added for illustration. These solutions span languages, either using built-in Excel functions (solutions on lines 1 – 5) or VBA (solution on lines 6 – 13). Given this specification, the code on line 1 and lines 6–13 do not behave as specified; they omit the `:ss` and/or the `AM/PM` components. This leaves three possible answers that would be returned by a semantic code search engine. However, an abstraction on the string values would allow the solver to identify string values that could make the two non-solutions become solutions.

Thrust 1 explores how to make this possible, exploring abstractions and mutations to amplify the search space. Ultimately, the abstractions and mutations are hidden from the programmers; the impact is seen in the search results. Abstraction and mutation create new code that was generated by modifying existing code, creating new search results that would not be there otherwise.

3.1.1 Abstractions

Abstractions expand the search space and enable finding an existing match that is behaviorally similar. I propose to model relationships between abstracted code representations using a lattice with constraints in the nodes and subsumption relationships on the edges. If a specification is satisfied at one level, it is also satisfied at levels below it. Lower levels of the lattice are more likely to match but require more adaptation [87]. Higher levels are less likely to match but return code closer to what a developer wrote. Since subsumption is a strong criteria, I will also explore precondition and postcondition matches outlined by Zaremski and Wing [102]. I propose the following four abstractions:

[2]https://stackoverflow.com/questions/220672/convert-time-fields-to-strings-in-excel



```
myTime = [A1]
myStrTime = Format(myTime, "hh:mm")
myStrTime = myStrTime & " Nice!"
```

```
myTime = [A1]
myStrTime = Format(myTime, "?")
myStrTime = myStrTime & " Nice!"
```

```
myTime = [A1]
myStrTime = Format(myTime, "hh:mm")
myStrTime = myStrTime & "?"
```

```
myTime = [?]
myStrTime = Format(myTime, "hh:mm")
myStrTime = myStrTime & " Nice!"
```

```
myTime = [A1]
myStrTime = Format(myTime, "?")
myStrTime = myStrTime & "?"
```

```
myTime = [?]
myStrTime = Format(myTime, "?")
myStrTime = myStrTime & " Nice!"
```

```
myTime = [?]
myStrTime = Format(myTime, "hh:mm")
myStrTime = myStrTime & "?"
```

```
myTime = [?]
myStrTime = Format(myTime, "?")
myStrTime = myStrTime & "?"
```

Figure 4: Abstraction lattice over strings and spreadsheet cells for VBA code in Figure 3 on lines 6 – 13. The boxes with question marks represent abstracted values.

Variable Values: A first step involves weakening hard-coded primitive values, such as integers, characters, or floats, so the solver can identify values for the satisfiable model. Suppose an end-user programmer wants a solution in VBA to the input/output example in Figure 3(a). The code in Figure 3(b) on lines 6 –13 does not behave as specified. The lattice in Figure 4 represents possible abstractions on the cell reference (i.e., **`[A1]`** on line 9) and the hard-coded string values in lines 10 and 11. Written as-is, the code at the top level of the lattice does not satisfy the specification. As values are relaxed, we move down the lattice, allowing the solver more liberty in modifying the code to meet a specification. The input/output example is satisfied only when the two strings are abstracted, which is true of the bottom level as well as moving up and to the left. In generating the satisfiable model, the solver would identify that the string values of "**`hh:mm:ss AM/PM`**" and "" (empty string) would satisfy the specification. For the bottom level, the solver would identify **`[A1]`** as the spreadsheet cell. These abstractions allow this code, after modification, to be part of the solution space.

Backwards Slicing Program slicing decomposes programs based on their control and data flow [100]. Given a statement of interest, backwards slicing identifies and retains all statements above it that impact the statement. For example, in Figure 5, the methods in (a) and (c) are equivalent after backward slicing on **`z`** in **`original`** to create **`sliced`**. Thus, it is possible to conclude that the behavior of **`sliced`** $\subseteq$ **`original`** and **`sliced`** $\equiv$ **`addAB`**. The sliced code would appear at lower levels of the lattice compared to the original code. Proving equivalence between **`sliced`** and **`addAB`** would connect those lattices (Thrust 3).

```
1  int original() {
2    int x = 1;
3    int y = 2;
4    int z = y-2;
5    int r = x;
6    z = x + y;
7    return z;
8  }
```

(a) Method to illustrate backward slicing.

```
1  int sliced() {
2    int x = 1;
3    int y = 2;
4
5
6    z = x + y;
7    return z;
8  }
```

(b) Backward slice on variable **`z`** on line 6.

```
1  int addAB() {
2    int a = 2;
3    int b = 1;
4    int c =  a + b;
5    return c;
6  }
```

(c) Method equivalent to the slice in part (b).

Figure 5: Example of backward slicing in parts (a) and (b), and a method in part (c) equivalent to the slice.

Interface: Type signature abstraction involves deleting, modifying, or adding input parameters. *Adding* could be useful when two pieces of code satisfy different parts of a specification; the snippets can be composed together to create a full match, borrowing techniques from conditional synthesis [53, 56, 57, 60, 101]. The code after adding would appear higher in the lattice.
Deleting could involve a forward slice on a variable [100]; this would identify all the code that depends on the variable and could be removed. This is appropriate when the type signature of the input/output examples is a subset of the type signature of a method under consideration. The original code would appear at a higher level in the lattice, the code after forward slicing would appear lower.
Modifying could take advantage of behavioral subtyping [52] and change the types of existing parameters for when the specification and candidate snippet have different but compatible type signatures (e.g., changing a `float` to an `int`). The subtyped representation would appear higher in the lattice.

Relational Operators: Recent work in mutation testing using relational operators showed that some mutations are harder to kill than others [96]. For example, changing `<=` to `<` is killed only when the values being compared are equivalent. These hard-to-kill mutations create behaviorally similar programs to the original, possibly facilitating an abstraction lattice (if an abstraction lattice cannot be formed, this can be a mutation, as in Section 3.1.2). As with prior work [96], model counting will identify hard-to-kill mutants.

3.1.2 Mutations

Mutation testing [31] is used to evaluate test suite quality by inserting small modifications into code and observing the impact on the test suite behavior. In semantic code search, mutations expand the search space and enable finding a match that is structurally similar to existing code. The implementation is similar to abstraction in that the solver will identify how to modify the code. For example, a mutation on arithmetic operators could replace + with a selection from the group: {`-, +, *, /, %`}. The solver selects which to use in the satisfying model for a specification. In essence, this creates a lattice with two levels, one with original operator and another below it with a relaxed operator. I propose to explore three mutations:

Standard Mutations: We will use standard mutations over arithmetic operators, array indices, and other constructs [33]. Given a specification, the solver decides which operator to use to create matching code.

Method Replacement: Abstracting the method itself would increase the search space. For example, in the `calendar.py` library, functions, `itermonthdates(month, days)`, `itermonthdays(months, days)` and `itermonthdays2(month, days)` have the same method signature and could be treated as a mutation.

Variable Replacement: Abstracting variables in code means replacing them with another in-scope variable [47]. For example, abstracting a global variable means the solver selects another global variable to satisfy the specification. Swapping the parameter order is a built-in process, as described in Section 2.

3.1.3 Evaluation

We will implement stand-alone tools or plugins to support the studies. We will host large repositories of programs on a secure server at North Carolina State University (NCSU), which is dedicated to this project and will evolve to support all the thrusts in this proposal.

After building lattices for each snippet, there are several dimensions to explore, including:

1. where to abstract/mutate (e.g., return statements, if-conditionals, cell references, operators);
2. how much to abstract (e.g., globally, instances, combinations of instances);
3. how many choices to offer (e.g., the solver can choose any spreadsheet cell, or only in column `A`);
4. what to abstract (e.g., values, types, interfaces, relational operators); and
5. where in the lattice to start the search (e.g., at the bottom, middle, or top).

Tradeoffs in speed, search space size, and solution space size will be explored. Studies will use three scenarios: 1) writing a program from scratch, 2) searching without abstraction, and 3) searching with abstraction.

Success will be measured in time and accuracy. In preliminary work, I explored abstraction for Yahoo! Pipes [82, 87] considering string and integer values, applied globally. Results showed an up-to-89x increase in the solution space and a significant slowdown in solver speed, especially when abstracting strings.

3.2 Thrust 2: Expressive Query Models

Input/output examples are a weak specification. When a programmer is writing examples by hand, enumerating all possibilities is laborious. We have observed that programmers under-specify their code when asked to provide examples: they prefer code that has additional control flow paths not exercised by their examples [88]. This demonstrates a need for more precise behavioral queries.

Suppose a programmer is looking for code that returns `true` for values in the range `[1, 10]` and `false` otherwise. To illustrate this behavior using examples and adhering to a boundary value testing approach [1] would require eight inputs, where the input is `MIN_INT`, `0`, `1`, `2`, `9`, `10`, `11`, and `MAX_INT`. A more precise way would use input ranges, such as, `[MIN_INT, 0]` $\Rightarrow$ `false` $\wedge$ `[1, 10]` $\Rightarrow$ `true` $\wedge$ `[11, MAX_INT]` $\Rightarrow$ `false`. The particular query representation may need to change depending on the client (e.g., not all end-user programmers may be comfortable with propositional logic), but incorporating some formalism into query models may be advantageous to enhance the precision of the search.

Thrust 2 proposes techniques to increase the expressiveness of specifications supported by semantic code search through the use of conditional specifications, partial programs, and negation. We will create an interactive system to support specification composition by predicting boundary values or edge cases.

Conditional Specifications: Abstracting concrete values in input/output examples will yield a lightweight yet stronger method of specifying behavior. 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. A conditional specification could say that, for all input `x`, if `x`$\geq$`0`, the output is `sqrt(x)`, else it is `0`. This approach would reduce the over-specificity associated with input/output examples and reduce overfitting since the examples would cover a broader range of behaviors.

Any function that can be synthesized can be used in a specification. Challenges include how to intuitively deliver this capability to the programmer and how to manage solver performance. For example, the range of values `[0, MAX_INT]` is quite large and may not be efficient. Trimming to `[0, 9]` may improve the solver time and could be used as a first pass in the search before trying the whole value range.

Negation: I propose to explore specifications that support exceptions, error conditions, or negated values. As an exceptions example, given an input of `null`, an output could be that `NullPointerException` should (or should not) be thrown. For error conditions, given an input of `null`, an output could specify that the program should terminate. For negated values, given an input of `0`, the output should anything except `0`. Allowing such specifications will enable developers to specify undesirable or disallowed behavior.

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 partial code and receive a completed piece of code as a result. This is similar to program synthesis by sketching [18, 64, 78, 79, 81], except that in our approach, the programmer would not need to specify the "holes"; that is done automatically by abstraction. Instead, search would find existing code that behaves as specified and has (at least some of) the same pieces as the original.

3.2.1 Evaluation

Evaluating the richer specifications requires putting them in the hands of users and determining if users can effectively use the new models to specify desired code. Success will be measured in time, effort, and accuracy. This will be evaluated for end-user programmers and professional programmers; it is possible that these groups may benefit from different models.

```
 1  ' VBA method
 2  Function VB(ByVal x As Integer)
 3    As Integer
 4
 5    Dim a As Integer, b As Integer
 6    Dim c As Integer
 7    a = x * 2
 8    b = x * 3
 9    c = a * b
10    VB = c
11  End Function

12  # Python method
13  def Py(x):
14      d = x * x
15      e = d * 6
16      return int(e)
```

```
17  (set-logic UFNIA)
18  (declare-fun VB (Int) Int)
19  (declare-fun Py (Int) Int)
20
21  (assert (forall ((x Int))
22    (exists ((a Int)(b Int)(c Int)(output Int))
23      (and
24        (= (VB x) output) (= a (* x 2))
25        (= b (* x 3)) (= c (* a b))
26        (= output c)))))
27  (assert (forall ((x Int))
28    (exists ((d Int)(e Int)(output2 Int))
29      (and
30        (= (Py x) output2) (= d (* x x))
31        (= e (* d 6))     (= output2 e)))))
32  (assert (forall ((x Int)) (= (VB x)(Py x))))
33  (check-sat)
```

Figure 6: Example programs and SMT constraints for proving equivalence between VBA and Python

3.3 Thrust 3: Characterizing Semantic Similarity

In a preliminary study on code search at Google, we found that developers frequently search for examples [68]. In MySQL, of 100 StackOverflow questions evaluated, we found that 74 were looking for examples of how to perform tasks; 53 of those contained input/output examples [87]. During search, when many code examples are returned, differentiating between them currently involves inspection or execution.

For example, in Figure 3, Solution 2 has one `"h"` in the string `"h:mm:ss"` and solutions 3 and 4 have two, as in `"hh:mm:ss"`. A behavioral difference is seen when the hour is a single-digit, such as `9:03:32`. Jumping from the syntactic difference to the behavioral difference is non-trivial [41], especially in a multilingual environment.

Thrust 3 proposes techniques for characterizing similarities and differences between code fragments by proving equivalence, identifying differentiating inputs, or characterizing similarity using constraints. Anticipated benefits include explaining similarities and differences to the user, identifying connections between abstraction lattices (Thrust 1), refactoring verification, and cross-language clone detection.

3.3.1 Proving Equivalence

We can use state-of-the-art SMT solvers to prove equivalence of some code snippets. This is useful for cross-language clone detection that is not dependent on the languages having a common intermediate language (e.g., VB.NET and C#.NET clone detection exploits the common AST [44]). Representing code as semantic constraints provides a common abstraction for proving equivalence.

For example, consider the two methods in Figure 6. Method `VB` is in VBA and `Py` is in Python. While structurally different, the return value for both is `x*x*6`. The constraints on lines 21–26 represent method `VB` and lines 27–33 represent method `Py`. Line 32 asserts that for all integers, the two methods behave equivalently. When provided to Z3 [12], the outcome is `sat`, indicating these are functionally equivalent.[3] While a simple and contrived example, this shows the potential of using Z3 for cross-language clone detection. A more complex example appears in Figure 2 with snippets in Java and MySQL. As strings are more complex to represent as constraints for solvers [36, 103], for simplicity, we show an integer example.

In the event that the solver returns `unknown` due to constraint complexity, a back-up plan uses fuzz testing [20, 55] or code relatives [94] to determine with some empirical certainty if the methods are the

[3]For some fun, copy-paste the constraints into the online Z3 interpreter and see for yourself: http://rise4fun.com/z3/tutorial

same or behaviorally close. Using fuzzing or dynamic behavior (code relatives) also allows us to measure similarity between compilable code fragments unsupported by the semantic encoding, though it is not an option for abstracted and mutated programs in which the solver fills in the "holes".

3.3.2 Differentiating Input

When methods are not identical, there exists at least one input on which the methods behave differently. Programmers may want answers to the question, "how are these methods *different*?" [41]. Identifying one, or a class of inputs, can be informative for describing how and when the methods' behaviors differ. For example, using the solver, a differentiating input can be identified by changing line 32 in Figure 6 to `(assert (exists ((x Int)) (not(= (VB x)(Py x))))).` Instead of asserting the methods are identical for all integers, this revised line asks for an integer on which the methods differ. If `sat` is returned, the satisfiable model returned by the solver identifies a value on which `VB` and `Py` behave differently. Removing the `not()` operator will identify an input on which the methods `VB` and `Py` are the same, if one exists, leading to describing similarity (Section 3.3.3).

If the solver fails to identify a differentiating input (i.e., returning `unknown`), random input generation or fuzzing may be useful, though its not guaranteed to find an answer. We can guide the selection of inputs based on "magic" values in the code, such as the time `12:00:00` in solution 4 of Figure 3. Another idea uses Topes [71] to recognize categorical data and exploit known boundary values for a domain.

3.3.3 Similarity

Programmers often make simplifying and invalid assumptions about programming language structures [40]; some language constructs are more prone to invalid assumptions [41]. Demonstrating behavioral similarity between fragments within the same language can illustrate alternate implementations to aid comprehension.

Consider, for example, the methods in Figure 5(a)–(c). Part (a) and (c) show two methods from a hypothetical repository. Part (b) shows a backward slice on the variable `z` on line 6 from `original`. The backward slice is equivalent to the method in (c). I propose to quantify similarity using strong measures (e.g., via constraint representation, backward slicing) and weaker measures (e.g., via structural techniques or fuzzing) to characterize the differences between two pieces of code.

Solver-based Similarity: Rather than finding a single input on which the code snippets are the same, the solver could help determine classes of similarity. For example, it could determine two methods are equivalent for all negative integers $x \mid x < 0$, for all strings with fewer than five characters, or for all lower-case letters in the English alphabet. Model counting [14] could identify all the models on which two code fragments are similar. Patterns can be formed based on the models' contents to characterize the conditions under which the methods are alike, inspired by equivalence class testing [1]. Challenges include identifying meaningful equivalence classes and division points between the classes for exploration. We will start with manual analysis and then identify the division points automatically based on values in `if` and loop conditionals.

Working from the constraint representation provides several options in describing similarity. If proving equivalence returns `unsat`, the unsat core identifies a set of assertions that are mutually unsatisfiable. Similarity between two methods could be measured based on the size of the unsatisfiable core. Another approach could use MaxSAT [50] to measure the maximum number of clauses, or the percent of clauses, that are satisfiable. Similarity could be measured using abstractions, as described in Section 3.1. Using a canonicalization on the constraint representations and longest common sequence of constraints could measure similarity. Constraint reuse may also provide a notion of similarity [97]. In all these approaches, the challenge is ensuring that the similarity in constraints is reflective of the similarity in actual code behavior, especially when the constraints have been abstracted.

Structural Similarity: A common way to explore structural similarity is to look for isomorphisms in data-flow and control-flow graphs [43, 45, 90]. My prior work in code similarity [90] considers end-user

Yahoo! Pipes programs and looked for structural similarity using several levels of structural abstraction that could be adapted to the control flow or data flow graph for the targeted languages. Product lines [9] may be a practical formalism to describe code similarity when mutation describes the differences between programs. Re-framing Figure 5 as a product line, the code in part (b) or (c) could form the base program and lines 4–5 in part (a), which are removed during slicing, could be an optional variation for the method. Paired with identifying a differentiating input per Section 3.3.2, it could illustrate an instance of how the code differs.

3.3.4 Evaluation

Evaluating Thrust 3 involves: cross-language clone detection, refactoring, and code change comprehension.

Refactoring Validation: Refactoring is a semantics preserving transformation over source code [16], and there is substantial evidence that all programmers maintain and restructure their code [26, 83, 90]. Best practices indicate a passing test suite should exist prior to refactoring, and that the code should still pass after refactoring. That is, as long as the code after refactoring passes, it is deemed to be a successful refactoring. However, as tests are often a weak specification for intended code behavior, soundness and completeness are not guaranteed, and not all end-user programming environments make testing easy. Using existing refactoring datasets from spreadsheets [23–26] and professional languages [35], we will determine if the proposed techniques are sufficient for proving equivalence or identifying differentiating inputs for refactored code. Success will be measured by how many refactoring patterns can be validated by these techniques.

Understanding Code Changes: Given a method before and after a change, the similarity analysis reveals how and when the behavior is different from the original method. In the hands of users, this could aid code changes comprehension. The evaluation will use Python repositories from GitHub. In an A/B test with and without the similarity analysis, we will ask participants to describe the impact of code changes. Success will be measured by accuracy in describing changes in program behavior.

Cross-language clone detection: Suppose a company acquires a competitor and needs to integrate features from a new product into an existing offering. A software architect must identify the relevant components of the acquired product and integrate the new code into the existing product. A key challenge is understanding how the two systems implement common features, which differ on language, naming conventions, and structure. This scenario faces my industrial partner, ABB. Of particular benefit would be techniques that characterize similarity based on constraints, as this common representation would facilitate cross-language clone detection. ABB has offered access to industrial systems to test the techniques and to developers for interviews, observation, and evaluation. A byproduct of this research could be language migration libraries for the supported languages.

3.4 Thrust 4: Navigating the Solution Space

As semantic code search can handle larger and more complex code and abstractions are applied to amplify the search space, the space of potential solutions for a weak specification will tend to increase. The collection of potential solutions is called the *solution space* (Figure 1). Navigating this space becomes a burden for the human or algorithm using the search. Reducing this burden requires a more complete specification (e.g. more input/output examples). Asking programmers for additional examples to refine the solution space is rapidly becoming the next bottleneck in semantic code search as well as program synthesis [75].

Thrust 4 proposes techniques to help clients of semantic code search (and synthesis) navigate the solution space, exploring human-in-the-loop and human-out-of-the-loop approaches. From the perspective of the programmer, the impact will making fewer decisions before converging on a desired solution.

3.4.1 Input Selection

In semantic code search or synthesis via input/output example, the literature often implies that a specification can be strengthened to prune the solution space by simply adding another example [76, 87]. However, not

(a) Business Information Table (input 1)

bus_id	name	address	city	state	latitude	longitude
16441	“HAWAIIAN DRIVE"	“2600 SAN BRUNO AVE"	SFO	CA	NA	-122.404101
61073	“FAT ANGEL"	“1740 O’ FARRELL ST "	SFO	CA	0.0	-122.433243
66793	“CP - ROOM"	“CANDLE PARK"	SFO	CA	37.712613	-122.387477
1747	“NARA SUSHI"	“1515 POLK ST "	SFO	CA	37.790716	NA
509	“CAFE BAKERY"	“1365 NORIEGA ST "	SFO	CA	37.754090	0.0

(b) Inspection Table (input 2)

bus_id	Score	date
509	85	20130506
1747	93	20121204
16441	94	20130424
61073	98	20130422
66793	100	20130112

(c) Output Table

bus_id	name	Score	date	latitude	longitude
66793	“CP - ROOM"	100	20130112	37.712613	-122.387477

Figure 7: Inputs and Output Example

all examples prune the solution space effectively. For example, consider the specification in Figure 3 and the set of solutions. The solutions on lines 2, 3, and 4 would be returned by a semantic code search engine given the specification, but it is not entirely clear which should be the winner. Adding another test may or may not reduce the solution space. For example, an input of “`11:01:02 PM`” would retain solutions 2, 3, and 4. The challenge is identifying inputs that divide the solution space. I propose to automatically find an input that maximally fractures the solution space; when a user provides an output, the solution space is reduced as much as possible.

In preliminary work, we tackled this challenge in the domain of data wrangling using Python [75]. Consider the two input and one output tables in Figure 7. The example is from the Zipfian Academy, a group that teaches Python novices how to analyze large data sets[4]. Here, a fictional end-user programmer wants to analyze San Francisco restaurant inspection data to understand the “cleanliness of the city”. The data is available from the city of San Francisco’s OpenData project. The challenge is that the data needs some wrangling (e.g., merging, formatting, filtering) before it can be analyzed. For example, the scientist must join Figure 7(a), containing business information, and Figure 7(b), containing inspection data, and then filter rows with invalid latitude and longitude values to obtain the output shown in Figure 7(c). Given this example, our synthesis engine produces 4,418 Python programs that perform the transformation.

We explored an approach to input selection that permutes the input tables, creating up to 100 different inputs. Each of the programs in the solution space is executed against the inputs. Based on the output values, the programs are clustered. The input that creates *the most* clusters is selected as the one to present to the user, for which they provide an output. In the example in Figure 7 and using the program provided with the data set as the oracle, three more inputs reduces the solution space from 4,418 to two.

Efficiency of this approach is a challenge. In the worst case, given k programs in the solution space, the user will evaluate $k-1$ inputs (i.e., each input creates two clusters, one of size $k-1$, and one of size 1). In the best case, the user will evaluate just one input, where the desired program is in its own cluster. If this approach does not meet reasonable performance standards, there are several mitigation strategies. Concerning the number of inputs to generate, a larger number improves the probability that the “best" input can be identified to divide the space, but a smaller number of inputs reduces the time costs. Inputs could be generated using the differentiating input approach in Thrust 3 (Section 3.3.2) instead of randomly. Regarding clusters, we select the input leading to the highest number of clusters, but should also consider uniformity in cluster sizes. A cost-benefit analysis will determine the impact of the input generation approach and the number of inputs on the effectiveness of the approach.

[4]http://nbviewer.ipython.org/github/Jay-Oh-eN/happy-healthy-hungry/blob/master/h3.ipynb

3.4.2 Ranking

For some applications of semantic code search, a human is not available to provide feedback on which solution is the best. Ranking may help identify the program with the highest probability of being correct. I propose to develop ranking techniques based on 1) behavioral similarity, 2) conformance to the specification, and 3) elements familiar to the developers. Ideal ranking may include some or all of these approaches.

Ranking by Semantic Popularity: The solution space contains code with some behavioral similarities. I propose to exploit behavioral similarity, as measured in Thrust 3, to create clusters. Sorting the clusters by size from largest to smallest and selecting a member from each of the top *n* clusters forms the top *n* results. Given the high redundancy in open source code [3, 17], it would make sense that someone has written the intended code, and possibly many people. Thus, semantic commonality could be useful for reuse.

Conformance to the Specification: Given a specification and a solution, how well that specification covers the paths or statements in the solution could provide insights for ranking. We already have some evidence that coverage matters in determining the relevance of Java code snippets to questions asked on StackOverflow. Results show that code which provides *more* behavior than the specification in terms of program paths was the most relevant (i.e., the code had paths that were uncovered by the specification) [88]. Initially this means that code with higher complexity is more relevant, but there is likely to be an upper bound on the ratio of covered to uncovered program paths. One goal is to discover this bound during evaluation.

Familiar Programming Constructs: In end-user programming, some characteristics of programs, such as counts of comments, code length, and parameters, are predictive of reuse for web macros [70]. We replicated that result on two different end-user language domains, web mashups, and GreaseMonkey scripts [32]. Such program characteristics, or code smells, could be used for ranking based on reuse probability.

3.4.3 Evaluation

Navigating the solution space via input selection requires user involvement. The assumption that the cost of creating/evaluating an output is lower than the cost of providing a differentiating input/output example requires exploration. To begin, we will design user studies that use real data sets, such as the San Francisco restaurant data used in Section 3.4.1. Success will be determined by time and effort in selecting output for an input. This will be balanced with search space reduction compared to non-guided input selection. In preliminary work, we have some evidence that developers can correctly identify an output for a given input for MySQL tables [86], but cost was not evaluated against generating examples from scratch.

Success in ranking will use the input/output examples from recent work in synthesizing SQL queries from input/output examples [99]. Comparing the synthesized SQL queries to searched SQL queries in terms of understandability, generalizability, and relevance will require human evaluation. For study subjects, NCSU has a Computer Science Masters program with a track in software engineering for which research or participation in research studies is a graduation requirement. This free pool of qualified participants are currently employed full time in industry or looking for an industry job, and thus are representative of people who perform code search tasks.

4 Related Work

Specifications used in previous semantic code search work include formal specifications [19, 62, 102] and test cases [46, 49, 63, 65]. Formal specifications allow precise and sound matching but must be written by hand, which is difficult and error-prone. Recent work in keyword-based search has begun to incorporate semantic information for finding working code examples from the Web [38] or reformulating queries for concept localization [22]. Several code search engines or example recommendations tools exploit structural information, such as Strathcona [30], Sourcerer [2], and XSnippet [69]. This proposal builds on the idea of searching based on approximate behavior, rather than structure or execution.

Program synthesis approaches depend on input/output examples to describe code to synthesize [21, 76, 99] and input/output examples are the cornerstone of programming-by-example systems that target notices or end-user programmers [10, 51]. Abstraction is similar to synthesis by sketching [18, 80, 81], except the code skeletons come from existing code whereas developers provide the skeletons in sketching.

5 Educational Benefits, Outreach, and Broader Impacts

This section describes proposed broader impact efforts in mentoring, education, and technology transfer.

Mentorship: To reduce some of the barriers women face in computer science, such as conflicts between professional and family life [73] and feelings of isolation [74], I plan to develop a structure around my mentorship activities. Currently, I mentor two female PhD students and one veteran. I also reach out to 1-2 additional female undergraduate and graduate students per year, talk with them about their goals, and check-in periodically over e-mail. My goal is to synthesize my mentoring activities and bring together women at various stages of their careers, adding 1-2 more each year, expanding to mentees outside my university, and creating cohorts. This small-scale community building can be more personal than larger initiatives, allowing us to do monthly or weekly check-ins, a summer retreat, and possibly attend the Grace Hopper Conference, as I have done with students in the past. Selection will happen informally, but if demand increases, I will consider outsourcing some mentoring to senior graduate students, postdocs, or faculty.

I plan to share my experiences during weekly faculty lunches within my department and extend my experiences to mentoring programs at academic conferences. At FSE 2016, I piloted a successful NSF-sponsored Mentorship Sessions program (PI:Emerson Murphy-Hill). Students, postdocs, or junior faculty requested to meet with senior faculty. I matched students with mentors and facilitated lunch meetings; 23 mentees met with 14 mentors, totaling 46 meetings. Nearly 90% of the mentees reported that the sessions were valuable and 75% of the mentors plan to keep in touch with their mentees. I designed these sessions because it is often difficult for junior researchers to start a professional mentoring relationship with people outside their university, but these relationships are very valuable in building and maintaining community.

Undergraduate Education: Activities related to this grant target newcomers and REU students.

Newcomers: Code search that identifies multiple semantically identical implementations of code could be valuable in education, especially for newcomers or end-user programmers. Test cases provide examples when code behavior diverges, but the techniques to more precisely characterize code similarities and differences (Section 3.3) provide more useful information. The techniques will be piloted in entry-level programming classes at NCSU for non-majors, which teach Python.

REUs: Each summer, the software engineering group at NCSU runs an NSF-funded REU (Research Experience for Undergraduates) on the "Science of Software." Places are reserved for students from traditionally under-represented areas (e.g. economically challenged regions of the state of North Carolina) and/or students from universities lacking advanced research facilities. Over the past two summers, I have worked with seven students; two have been involved with code search projects.

Graduate Education: In Fall 2018, I will develop a graduate course on *Semantic Analysis in Software Engineering*, focusing on how to use code search and synthesis for software engineering support. I will recruit research assistants from the enrolled students and pilot the empirical studies.

Technology Transfer: Semantic code search in Java has piqued the interest of NASA's JavaPathfinder team (JPF). In summer 2017, they hosted a Google Summer of Code, for which the PI has a student exploring code similarity in Java (Thrust 3). In future summers, the PI will pursue future opportunities for technology transfer from this project to industry, and specifically the JPF team, through this program.

ABB has written a letter of support for this research, with a particular interest in its applications to cross-language clone detection for C/C++, C#, and Java. ABB has offered access to industrial systems to test techniques and access to developers for interviews, observation, and evaluation.

Research Artifacts: I will make my tools available on an open source license. I will submit and disseminate techniques, data, results, and other artifacts associated with the research to top software engineering research venues. As I have done for my past experiments [83, 85–88], I will make all artifacts available on a central project website for other researchers to use in replications or comparison studies.

6 Results of Prior NSF Support

PI Stolee has an NSF grant that has been recommended for funding, titled, *"SHF: Small: Supporting Regular Expression Testing, Search, Repair, Comprehension, and Maintenance"* (total: $499,996, July 1, 2017 – June 30, 2020). **Intellectual merit:** Advances in regular expression analysis [7] are proposed, including test criteria, similarity metrics, code smells and refactoring. **Broader Impact:** This work will fund two female PhD students. It has the potential to improve software quality and comprehension. *Code search for regular expressions is a small focus of the SHF-small grant; adapting code search to regular expressions for end-user programmers is absent from (and complementary to) this CAREER proposal.*

PI Stolee has a joint NSF grant titled, *"SHF: Medium: Collaborative Research: Semi and Fully Automated Program Repair and Synthesis via Semantic Code Search"* (total: $1,200,000, PI Stolee's portion NSF CCF-1563726 $387,661, July 1, 2016 – June 30, 2020). This work extends another joint NSF grant titled, *"SHF: EAGER: Collaborative Research: Demonstrating the Feasibility of Automatic Program Repair Guided by Semantic Code Search"* (total: $239,927, PI Stolee's portion NSF CCF-1446932 $87,539, July 1, 2014 – December 31, 2016). **Intellectual merit:** This funding has resulted in advances in semantic code search for fault fixing in professional programming languages [37], understanding how professionals search for code [68], crowdsourcing [93,95], and automated repair [3,37,48,59,77]. **Broader impact:** The benchmark datasets developed for automated program repair [48] will advance research in the automated repair field by standardizing evaluations and enabling comparison among techniques (e.g. [77]). *The proposed work in semantic code search for all programmers in this CAREER proposal is complementary to my efforts to adapt semantic code search to program repair in the SHF-medium grant, which focuses exclusively on repair. This CAREER proposal primarily targets different languages, including Excel, VBA, MySQL and Python, with a long-term vision of supporting professional languages. The advances in this proposal, specifically the abstractions/mutations for end-user languages, distinguishing between matches, navigating the solution space and ranking, and applications to cross-language clone detection and refactoring verification, are new.*

7 Research Plan

Figure 8 maps the breakdown of work per year. Two graduate research assistants are budgeted. In Year 1, both will be assigned to developing abstractions and mutations (Thrust 1). In Year 2, one student will focus on similarity between programs (Thrust 3) while the other will focus on intuitive query models (Thrust 2). As the query models may aid in measuring similarity, both students will work on Thrust 3 in Year 3. In Year 4, we will begin to transition to professional languages for the abstraction in Thrust 1 while working on solution space navigation for *EUP* languages. In Year 5, we will focus on generalizing from the end-user domain to professional languages. This schedule recognizes dependencies between: the abstractions in § 3.1.1 and proving equivalence in § 3.3.1; mutations in § 3.1.2 and similarity in § 3.3.3; differentiating inputs in § 3.3.2; and input selection in § 3.4.1; similarity in § 3.3.3 and ranking in § 3.4.2. In most cases, we have fail-safe options for replacing constraint-based analysis with empirical similarity analysis.

	Year 1	**Year 2**	**Year 3**	**Year 4**	**Year 5**
Thrust 1 (§ 3.1)	EUP			Prof	
Thrust 2 (§ 3.2)		EUP			Prof
Thrust 3 (§ 3.3)		EUP	EUP		Prof
Thrust 4 (§ 3.4)				EUP	Prof

Figure 8: Breakdown of proposed work per year. *EUP* means end-user languages, *Prof* means professional languages.

References

[1] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[3] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China, Nov. 2014.

[4] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. *SIGPLAN Not.*, 49(6):123–132, June 2014.

[5] M. Burnett, C. Cook, and G. Rothermel. End-user software engineering. *Commun. ACM*, 47(9):53–58, Sept. 2004.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pages 209–224, 2008.

[7] C. Chapman and K. T. Stolee. Exploring regular expression usage and context in python. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[8] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 49–60. IEEE, 2016.

[9] P. Clements and L. Northrop. *Software product lines*. Addison-Wesley,, 2002.

[10] A. Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 33–39. ACM, 1991.

[11] R. M. de Mello, K. T. Stolee, and G. H. Travassos. Investigating samples representativeness for online experiments in java code search. In *International Symposium. on Empirical Soft. Eng. and Measurement*, 2015.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] S. Falke, F. Merz, and C. Sinz. Extending the theory of arrays. In *Verified Software: Theorie, Tools, Experiments: 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164, page 108. Springer, 2014.

[14] A. Filieri, M. F. Frias, C. S. Păsăreanu, and W. Visser. Model counting for complex data structures. In *Model Checking Software*, pages 222–241. Springer, 2015.

[15] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical symbolic execution with informed sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 437–448. ACM, 2014.

[16] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 147–156, Santa Fe, NM, USA, 2010.

[18] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, New York, NY, USA, 2014. ACM.

[19] C. Ghezzi and A. Mocci. Behavior model based component search: An initial assessment. In *Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*, pages 9–12, Cape Town, South Africa, 2010.

[20] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[21] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 317–330. ACM, 2011.

[22] S. Haiduc, G. D. Rosa, G. Bavota, R. Oliveto, A. D. Lucia, and A. Marcus. Query quality prediction and reformulation for source code search: The refoqus tool. In *International Conference on Software Engineering (ICSE) Formal Demonstrations Track*, pages 1307–1310, San Francisco, CA, USA, 2013.

[23] F. Hermans and D. Dig. Bumblebee: a refactoring environment for spreadsheet formulas. In *Proc. of ISFSE '14*, pages 747–750, 2014.

[24] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proc. of ICSE '12*, pages 441–451, 2012.

[25] F. Hermans, M. Pinzger, and A. van Deursen. Detecting code smells in spreadsheet formulas. In *Proc. of ICSM '12*, 2012.

[26] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, pages 1–27, 2014.

[27] F. Hermans, K. T. Stolee, and D. Hoepelman. Smells in block-based programming languages. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pages 68–72. IEEE, 2016.

[28] B. Hillery, E. Mercer, N. Rungta, and S. Person. Towards a lazier symbolic pathfinder. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[29] B. Hillery, E. Mercer, N. Rungta, and S. Person. Exact heap summaries for symbolic execution. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation-Volume 9583*, pages 206–225. Springer-Verlag New York, Inc., 2016.

[30] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 237–240, New York, NY, USA, 2005. ACM.

[31] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[32] J. Jackson, C. Scaffidi, and K. T. Stolee. Digging for diamonds: Identifying valuable web automation programs in repositories. In *Information Science and Applications (ICISA), 2011 International Conference on*, pages 1–10. IEEE, 2011.

[33] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[34] Symbolic PathFinder, December 2012.

[35] I. Kádár, P. Hegedűs, R. Ferenc, and T. Gyimóthy. A manually validated code refactoring dataset and its assessment regarding software maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 10:1–10:4, New York, NY, USA, 2016. ACM.

[36] S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 259–270, New York, NY, USA, 2014. ACM.

[37] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering*, 2015. to appear.

[38] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 664–675, Hyderabad, India, 2014.

[39] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.

[40] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005.

[41] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

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

[43] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.

[44] N. A. Kraft, B. W. Bonds, and R. K. Smith. Cross-language clone detection. In *SEKE*, pages 54–59, 2008.

[45] J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

[46] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53(4):294–306, Apr. 2011.

[47] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.

[48] C. Le Goues, N. Holtschulte, E. K. Smith, Y. B. P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE), in press, 22 pages*, 2015.

[49] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526. ACM, 2007.

[50] C. M. Li and F. Manya. Maxsat, hard and soft constraints. *Handbook of satisfiability*, 185:613–631, 2009.

[51] H. Lieberman. Programming by example. *Communications of the ACM*, 43(3):72–72, 2000.

[52] B. H. Liskov and J. M. Wing. Behavioral subtyping using invariants and constraints. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1999.

[53] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[54] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 575–586, New York, NY, USA, 2014. ACM.

[55] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM, 2014.

[56] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the International Conference on Software Engineering - Volume 1*, pages 448–458. IEEE, 2015.

[57] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[58] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[59] K. Muşlu, Y. Brun, and A. Meliou. Preventing data errors with continuous testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, MD, USA, July 2015.

[60] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[61] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[62] J. Penix and P. Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6:139–170, Apr. 1999.

[63] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2:286–303, July 1993.

[64] A. Raabe and R. 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.

[65] S. P. Reiss. Semantics-based code search. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 243–253, Vancouver, BC, Canada, 2009.

[66] E. F. Rizzi, S. Elbaum, and M. B. Dwyer. On the techniques we create, the tools we build, and their misalignments: A study of klee. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 132–143, New York, NY, USA, 2016. ACM.

[67] M. P. Robillard and R. Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[68] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[69] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. *SIGPLAN Not.*, 41(10):413–430, Oct. 2006.

[70] C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Predicting reuse of end-user web macro scripts. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 93–100. IEEE, 2009.

[71] C. Scaffidi, B. Myers, and M. Shaw. Topes. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 1–10. IEEE, 2008.

[72] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Symposium on Visual Languages and Human Centric Computing*, 2005.

[73] G. Scragg and J. Smith. A study of barriers to women in undergraduate computer science. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '98, pages 82–86, New York, NY, USA, 1998. ACM.

[74] A. Settle, J. Doyle, and T. Steinbach. Participating in a computer science linked-courses learning community reduces isolation. *arXiv preprint arXiv:1704.07898*, 2017.

[75] D. Shriver, S. Elbaum, and K. T. Stolee. At the end of synthesis: Narrowing program candidates. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ICSE-NIER '17, pages 19–22, Piscataway, NJ, USA, 2017. IEEE Press.

[76] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

[77] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[78] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. *SIGPLAN Not.*, 42(6):167–178, June 2007.

[79] A. Solar-Lezama, C. G. Jones, and R. 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.

[80] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. *SIGPLAN Not.*, 40(6):281–294, June 2005.

[81] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. 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.

[82] K. T. Stolee. Solving the Search for Source Code. PhD Thesis, University of Nebraska–Lincoln, August 2013.

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

[84] K. T. Stolee and S. Elbaum. Toward semantic search via SMT solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 25:1–25:4, 2012.

[85] K. T. Stolee and S. Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Softw. Eng.*, 39(12):1654–1679, Dec. 2013.

[86] K. T. Stolee and S. Elbaum. On the use of input/output queries for code search. In *International Symposium. on Empirical Soft. Eng. and Measurement*, October 2013.

[87] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 23(3):26:1–26:45, May 2014.

[88] K. T. Stolee, S. Elbaum, and M. B. Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software*, 2015.

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

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

[91] K. T. Stolee, S. Elbaum, and A. Sarma. Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes. *Information and Software Technology*, 2012.

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

[93] K. T. Stolee, J. Saylor, and T. Lund. Exploring the benefits of using redundant responses in crowd-sourced evaluations. In *Proceedings of the Second International Workshop on CrowdSourcing in Software Engineering*, pages 38–44. IEEE Press, 2015.

[94] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara. Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 702–714, New York, NY, USA, 2016. ACM.

[95] P. Sun and K. T. Stolee. Exploring crowd consistency in a mechanical turk survey. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*, pages 8–14. ACM, 2016.

[96] W. Visser. What makes killing a mutant hard. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 39–44, New York, NY, USA, 2016. ACM.

[97] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 58. ACM, 2012.

[98] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[99] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. ACM.

[100] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[101] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[102] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering Methodology*, 6, October 1997.

[103] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.