CAREER: Semantic Code Search for End-User Programmers

PI: Kathryn Stolee

A. Project Summary

Overview

End-user programmers lack formal training in computer science, but still perform programming tasks. Languages used by end-user programmers vary substantially, and their programming tasks may involve using built-in functions in Excel, performing MySQL queries, writing scripts in VisualBasic, or processing data sets with Python. Recent studies have shown that end-user programmers face many of the same challenges as professionals, think in terms of examples when writing code, and could benefit from behavioral search.

Semantic code search finds code based on behavior, described using input/output examples. The engine behind the 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. The PI has implemented semantic code search for the Yahoo! Pipes web mashup language and subsets of Java, C, and MySQL. It has been applied to code reuse and program repair. In reuse, results show that Java code fragments returned by semantic search are more relevant than those found by other search approaches. In repair, C patches are higher quality when compared to patches from other approaches. End-user programmers, who already think in terms of examples when programming, could benefit from such example-based search.

While promising, in pursuing the development of semantic code search, some common challenges have emerged. Specifically, 1) *finding close matches* when an exact match for a specification does not exist, 2) *characterizing the differences* between two fragments of source code, and 3) *navigating a solution space* of fragments that all match a specification. The PI proposes to perform basic research in each area, exploring techniques to relax program encodings, determine when and how code fragments differ in behavior, and quickly converge on the desired solution from a potentially large solution space. The PI will evaluate these techniques in the context of end-user programmers and novice programmers, who work with smaller languages such as Excel, MySQL, VisualBasic, and Python, and with fewer development support tools. The PI's vision is to bring the benefits and power of semantic analysis to end-user programmers to facilitate more informed code changes, promote reuse and adaptation of high-quality artifacts, and reduce developer effort.

Intellectual Merit

This work will substantially improve the power of semantic code search. For end-user and novice programmers, it will facilitate reuse, cross-language clone detection, and refactoring verification. The proposed work has three 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 database sizes.
2. Techniques to characterize behavioral differences between code fragments, including proving equivalence independent of language, identifying differences between fragments, and characterizing similarity.
3. Approaches to navigate the solution space after semantic search or synthesis to quickly converge on the 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, allowing them to more effectively write and understand their programs.
2. The PI will integrate semantic reasoning into software testing courses at the undergraduate and graduate level as a way to show the importance of test suites that adequately exercise code.
3. The PI will continue to mentor women in computing by working with female undergraduate and graduate research assistants and attending the Grace Hopper conference with university students.

CAREER: Semantic Code Search for End-User Programmers

PI: Kathryn Stolee

C. Project Description

1 Introduction

As the quantity of source code increases, especially in the open source domain, finding existing code to reuse or learn from becomes more difficult. The most common approach to this problem is to treat it as a search problem [2]. Yet, for end-user programmers, mapping a behavior to a textual search query is a learning barrier, suggesting that behavioral search is ideal [43].

In code search, a specification is used to describe the desired results. *Syntactic* code search uses a textual query as a specification and matches based on syntactic features such as keywords and variable names. For example, a developer trying to find a Java method to compute a triangle's type given three side lengths might search for "`Type of Triangle in MYSQL`", the title of a question on StackOverflow[1]. In contrast, *semantic* search uses behavioral properties as the specification. For example, the developer gave the input `{20, 20, 23}`, and asked for an output table with the triangle type, "`isosceles`" (see Figure 1(a) for the full example). Semantic code search can be achieved through executing source code [69] or mapping source code into constraints representing its behavior and using a constraint solver to identify matches for a query [39, 83, 86, 89, 90]. To date, it has been used in three applications: code reuse [69, 83, 86, 89, 90], program repair [39], and finding API usage examples [61].

End-user programmers read, write, and execute code to support their jobs or hobbies, despite a lack of formal training in programming [6, 41]. As with professional programmers, end-user programmers choose which languages and APIs to use [43], design and test their code [64], maintain their code over time [87], and search online for code to reuse [43]. As the population of end-user programmers was estimated to have reached 90 million in 2012 [75], the potential for impact in supporting their development activities is large. Researchers have focused on improving software engineering support, including introducing version control for web mashups [48], using assertions to improve the reliability of web macros [44], creating smell detection and refactoring tools for spreadsheets [26, 27], and synthesizing string transformations [23].

I propose to build on my prior work in semantic code search via constraint solver [12, 39, 83, 86, 89, 90] and end-user programming [30, 35, 44, 85, 87, 91–94] to bring the benefits of behavioral code search to end-users. In semantic search, queries take the form of input/output examples and a search space is composed of code snippets translated into a constraint-based representation. A constraint solver finds solutions that satisfy the query. In addition to encoding code snippets, challenging situations arise in semantic code search when 1) the desired code does not exist, 2) it is difficult to differentiate between similar snippets, and 3) there are too many results to navigate efficiently. To address these challenges, **I propose to 1) find approximate or easily adaptable solutions to semantic queries, 2) use the constraints to characterize the differences and similarities in behavior between code snippets, and 3) efficiently navigate the space of potential solutions**. These techniques will be evaluated with end-user and novice programmers.

Across the board, one challenge with using input/output examples is that they are a weak specification. In part this is what makes them so attractive and accessible; users already think in terms of input/output when specifying issues on StackOverflow [89, 90, 100]. In addition to semantic search, other research uses input/output examples to describe desired program behavior. Program synthesis approaches depend on them to describe code to synthesize [23, 78, 100] and they are the cornerstone of programming-by-example systems that target notices or end-user programmers [11, 54]. Yet, there can be many solutions that satisfy weak input/output specifications, or in the case of a very precise specification, no solution may exist at all. This proposal describes three research thrusts that target the aforementioned challenges related to semantic code search via input/output example.

---

[1] https://stackoverflow.com/questions/38561938/type-of-triangle-in-mysql

Input Table			Output Table
A	B	C	Type
20	20	23	Isosceles
20	20	20	Equilateral
20	21	22	Scalene
13	14	30	Not A Triangle

(a) Example input/output

```
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;
```

(b) MySQL query in a StackOverflow answer

Figure 1: Example specification for a MySQL query that from StackOverflow

Thrust 1: Approximate Solutions: For when there is not enough diversity in programs to find a match as-is, abstractions and mutations can amplify the search space. For example, in Figure 1(a), if the output table contained a Spanish or French language translation of the triangle types, then the solution in Figure 1(b) would require changes to the string values for this to be a solution. To address this challenge, **I propose to develop techniques to amplify the search space by approximating code semantics, abstracting program behavior, and mutating programs.** Abstractions identify code that is behaviorally close, whereas mutations identify code that is syntactically close, to the desired code. Such innovations take advantage of the constraint-based representation of source code in semantic code search and use the constraint solver to identify changes to the program that can lead to desired code.

Thrust 2: Characterizing Semantic Similarity: One learning barrier for end-user programmers is the selection barrier [43], suggesting that assistance is needed in choosing between two solutions. For the example in Figure 1, the StackOverflow community offers six different identical solutions, but in a very similar question,[2] offers solutions that differ slightly in behavior. **I propose to use the constraint representations of code snippets to characterize program similarity and differences**, which will enable end-user programmers to understand behavioral differences between two solutions. It will also facilitate cross-language clone detection and refactoring verification.

Thrust 3: Navigating Solution Space: For when there are too many solutions for a specification, techniques are needed to prune and organize the solution space. Whether programs that satisfy a specification are identified using program synthesis or semantic search, navigating the space of potential solutions is a pervasive problem, as the solutions can be numerous [76]. **I propose to develop techniques to rapidly prune the solution space by identifying inputs that maximally fragment the space and to rank order programs within semantic clusters.** As an aside, techniques for this thrust are also applicable to program synthesis approaches based on input/output examples, which abound [23, 76, 78, 100].

This research will transform state-of-the-art constraint-based semantic code search for end-user programmers through techniques to better guide the search to converge on a desired program.

2 Preliminaries: Semantic Code Search

The goal of semantic code search is to identify code described by a behavioral specification [86, 89]. It uses input/output examples as queries, indexes code snippets using constraints to represent behavior, and matches code to queries using a constraint solver. In the indexing phase, symbolic execution pre-processes source code snippets (i.e., at the method level or below) into constraints representing their behavior. For example, consider the code and constraints in Figure 2 for a StackOverflow question about extracting the

---

[2] https://stackoverflow.com/questions/44897941/mysql-better-solution-to-nested-if/44898023

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

(a) Java snippet

```
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

domain name from an email address[3]. Figure 2(a) presents a solution in Java, Figure 2(b) presents a solution in MySQL (from StackOverflow), and Figure 2(c) presents the constraints encoding the behavior of both. Since the snippets have the same behavior, in a constraint representation, they would look the same, except for naming conventions. Each translated snippet is stored in a database, which is searched to find results.

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 if snippets to check. From there, it will translate the input and output into constraints with assigned variables and bind those variables to each of the snippets' constraints. 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 [13]. A result of `sat` indicates the code behaves as specified; `unsat` implies it does not. A response of `unknown` is assumed to not be a match.

When a method has multiple input values of the same type, the assignment of input values to parameter values is decided by the solver [89]. Thus, parameter ordering in the specification is not an issue, as it can be with search that involves execution [69]. Consider an input {`5, 3, 4`}, an output *true*, and a method `def isPythagorean(a, b, c)`, which computes, $c == \sqrt{a^2 + b^2}$. Semantic code search encodes all possible mappings of the input values to the method arguments and lets the solver select one that works. That is, constraints are encoded for six mappings: $(a = 5 \wedge b = 3 \wedge c = 4) \vee (a = 5 \wedge b = 4 \wedge c = 3) \vee (a = 3 \wedge b = 5 \wedge c = 4) \vee (a = 3 \wedge b = 4 \wedge c = 5) \vee (a = 4 \wedge b = 5 \wedge c = 3) \vee (a = 4 \wedge b = 3 \wedge c = 5)$. This is important since only two mappings where $c = 5$ return `sat`. We make only one solver call to find a combination that works, instead of executing the method up to five times before finding a correct parameter ordering.

Originally implemented in Yahoo! Pipes, a (now deprecated) web mashup language, further efforts evolved semantic code search to subsets of Java [89, 90] and MySQL [89], and C [39]. 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 [90]. For program repair in C, SearchRepair [39] 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 [89]. As Yahoo! Pipes is a deprecated platform, it is omitted from this proposal, but language support included filtering, sorting, and combining RSS feeds [89].

---

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

The limitations with current semantic search rest largely on the limitations of symbolic execution. As symbolic execution gets more powerful, so does semantic code search. For Java, SPF [37, 65, 99] is the state-of-the-art tool, and recent efforts aim to improve, for example, heap summaries [31, 32] and efficiency through statistical probabilities [4, 9, 16, 57]. For C, KLEE [7] is the state-of-the-art and recent efforts have improved, for example, performance [70] and the underlying theories, such as arrays [14]. 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 VisualBasic, Python, or Excel (all yet to be explored). Using smaller 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; the PI wrote a translator to map Yahoo! Pipes data flow programs into constraints for an SMT solver, effectively creating a symbolic execution engine for a small language.

This proposal focuses on end-user and novice programmers as the clients of semantic code search. The targeted languages are MySQL, Python, VisualBasic, and Excel. A small subset of MySQL is supported for search already, but more indicative of its potential is the presence of synthesis engines for MySQL queries [100]; if it can be synthesized, it can be searched (though synthesis tends to overfit to a specification [79], a problem that does not plague semantic search [39, 90]). Excel formulas and VisualBasic functions for spreadsheets bring the concept of *spreadsheet cell location*, which provides unique opportunities for abstraction. Python provides a unique challenge in its use of dynamic typing, creating opportunities for natural abstraction in the type encoding (e.g., since a variable could be an integer or a string). For simplicity, we focus on low levels of granularity, such as methods or functions, though scaling to class, sub-system, and system levels is within the long-term vision of this work.

As we develop techniques to support semantic search for end-user programmers and other researchers enhance symbolic execution, innovations from the end-user domain can benefit professionals. For instance, ABB has an interest in this research, specifically cross-language clone detection. Their particular context, described in Section 3.2.4, requires support for C/C++, C# and Java. As a stretch goal, we would like to extend support for those languages, and have some evidence that it is realistic based on the application of semantic code search to program repair of C programs [39]. The primary focus of this proposal, however, is scaling semantic code search for end-user programmers and smaller languages.

3 Proposed Research

My broad research vision is to bring the benefits and power of semantic analysis to end-user programmers.

3.1 Thrust 1: Expanding the Solution Space

Searching for code to reuse in an open-source environment takes advantage of the quantity and variety of code available on platforms such as GitHub. However, for end-user programmers, large repositories of code in their language(s) may not exist at all [41]. 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 likely more limited, increasing the likelihood that the desired code does not exist. However, semantically close code may be available. In a preliminary exploration of semantic code search in C [39] applied to repairing bugs in the ManyBugs [51] benchmark dataset, 17/41 bugs were patched. One of the main reasons for the unpatched bugs was a lack of a match in the database. For these more constrained environments, such as end-user languages, academic settings, or industry code bases, it is reasonable to assume a similar situation could occur.

Finding approximate matches requires a notion of similarity between code and a specification to determine how well code meets a specification (which is notably different than similarity between two pieces of code, explored in Thrust 2). A *semantic match* means the code satisfies the specification. *Semantic similar-*

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

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

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

ity means that *with some modification*, the code satisfies the specification. This might mean that the code partially satisfes the specification already (e.g., four of five input/output examples are satisfied), or that it does not satisfy the specification at all (e.g., due to a return type mismatch).

For example, Figure 3 illustrates a StackOverflow question[4] 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 accepted answer was a work-around involving copy-paste to Notepad to remove formatting). 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 being built-in functions (solutions on lines 1 – 5) or written in VisualBasic (solution on lines 6 – 12). Given this specification, the code on lines 1 and 6 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 solution space to facilitate a match. Ultimately, the abstractions and mutations are hidden from the end-user programmers. The impact seen by them is in the search results. Abstraction and mutation create new code that was generated by modifying existing code to match the specification. Thus, the end-user programmer will receive search results that would not be there otherwise. This approach is similar to synthesis by sketching [20, 80, 81]. The main difference is that the code skeletons in this approach come from existing code written by others, whereas developers provide the templates in sketching. This is where semantic code search begins to blur the line between search and synthesis, as the solver synthesizes solutions to return `sat`.

3.1.1 Abstractions

Abstractions expand the search space and enable finding a match that is behaviorally similar to existing code. I propose to model relationships between abstracted representations using a lattice and subsumption. That is, if a specification is satisfied at one level of the lattice, it is also satisfied at all levels below it. Lower levels of the lattice will be more likely to match but require the most adaption and possible slower solver times [89]. Higher levels are less likely to match, but reuse code that is closer to what a developer

---

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

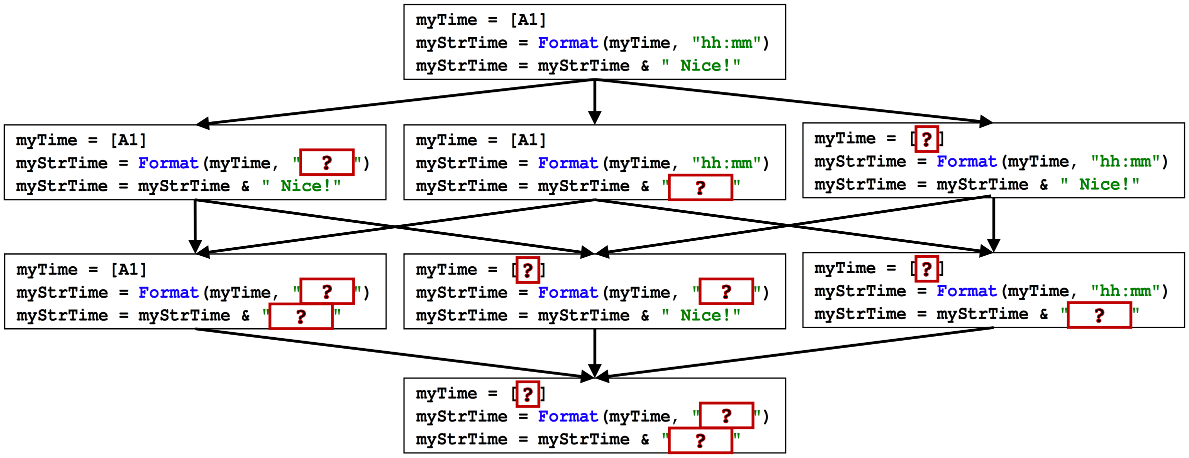

Figure 4: Abstraction lattice over strings and spreadsheet cells for VisualBasic code in Figure 3 on lines 6 – 12. The boxes with question marks represent abstracted values.

wrote. Since subsumption is a strong criteria for connections in the lattice, I also plan to explore the various specification matches outlined by Zaremski and Wing to model precondition and postcondition relationships between code behaviors [103]. I propose to explore the potential of building abstraction lattices by relaxing variable values, applying behavioral subtyping [55] and program slicing [101] to the program arguments, and abstracting the relational operators [97].

Variable Values: A first step involves weakening hard-coded primitive values, such as integers, characters, or floats, so the solver can identify appropriate values for the satisfiable model. Suppose an end-user programmer wants a solution in VisualBasic to the input/output example in Figure 3(a). The code in Figure 3(b) on lines 6 –12 does not behave as specified. The lattice in Figure 4 represents possible abstractions on the cell reference (i.e., `[A1]` on line 8) and the hard-coded string values in lines 9 and 10. 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. 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 to be part of the solution space.

Interface: Type signature abstraction involves deleting, modifying, or adding input parameters.
Deleting could involve a forward slice on a variable [101] 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 [55] 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.
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 [56, 59, 60, 63, 102]. The code after adding would appear lower in the lattice.

Relational Operators: Recent work in mutation testing using relational operators has shown that some mutations are harder to kill than others [97]. For example, changing `<=` to `<` impacts behavior only when the values being compared are equivalent. These hard-to-kill mutations create programs that are more behaviorally similar to the original, possibly forming an abstraction lattice (note that if an abstraction lattice cannot be formed, we can treat this as a mutation, described in Section 3.1.2). As with prior work [97], model counting will identify hard-to-kill mutants.

3.1.2 Mutations

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, an abstraction over arithmetic operators could replace the + operator with a selection from the group: {`-, +, *, /, %`}; the solver would select 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 relational operator. To ensure the mutants indeed expand the search space we will verify that they are reachable and not equivalent. I propose to explore three mutations to expand the search space.

Standard Mutations: This is inspired by mutation testing [34]. We use standard mutation operators over arithmetic operators, array indices, and other standard constructs [36], giving the solver the option of which to use to satisfy a specification.

Library Calls: 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 interface and could be treated as a mutation, where the solver can pick which is appropriate.

Variable Replacement: This mutation involves replacing one or more variables in a code fragment with another in-scope variable [50]. For example, if there are multiple global variables available, the solver could select a global variable that will satisfy the specification. Swapping the parameter order is built-in to the search process, as described (Section 2).

3.1.3 Evaluation

Once the lattices are built for each snippet, I propose to connect the lattices by proving equivalence between representations (Section 3.2.1). There are several dimensions to explore in abstracting encodings, 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. In preliminary work, I have explored abstraction for Yahoo! Pipes [83, 89] considering string and integer values, applying them globally. Solution space size and time were measured, revealing up-to 89x increase in the solution space and a significant slowdown in solver speed, especially when string values were abstracted.

Evaluating these techniques requires obtaining queries and repositories of code to search over, such that a solution does not exist. Input/output examples for MySQL queries will come from a publicly available dataset [100]. For Excel, we can use the EUSES spreadsheet corpus [17] to compose a database of snippets and input/output examples can come from StackOverflow. 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, accuracy compared to the intended programs, and the syntactic delta between an original program and a modified program (for parts 3). For MySQL, we will compare against query synthesis [100].

To support the studies, stand-alone tools or plugins will be implemented, depending on the language and environment. We will build a large repository of programs to be hosted on a secure server at North Carolina State University, which is dedicated to this project. This will allows us to update the repository of indexed programs without impacting the plugin users. Initially, the Neo4J graph-based database will be explored to explicitly model relationships among the encodings.

```
 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 5: Example programs for proving equivalence between VisualBasic and Python

3.2 Thrust 2: Differentiating Between Solutions

In a preliminary study on code search at Google, we found that developers frequently search for examples [71]. 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 [89]. During search, when many code examples are returned, differentiating between them is an important task, currently performed by inspection or by executing code. This thrust addresses the challenge of determining similarity and differences between methods or code fragments. Beyond the benefit to users, this is an important problem to address in support of Thrust 1 (Section 3.1); determining the similarity between abstracted and mutated code allows abstraction lattices for various fragments to be combined.

Consider the example from Excel in Figure 3. A difference between solutions 2 and 3, or between 2 and 4, appears when the hour is a single-digit. A difference between 3 and 4 appears when the time is between midnight and 1am. For example, with the time `0:59:59 AM` as input, the solution 3 returns `12:59:59 AM` while solution 4 returns `00:59:59 AM`. Characterizing when and how these solutions behave differently, especially when provided in different languages, is important for selecting which to use.

Thrust 2 proposes techniques for characterizing similarities and differences, such as these, 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, refactoring verification, and cross-language clone detection.

3.2.1 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 work exploits the common AST [46]). Representing code as semantic constraints provides a common abstraction for proving equivalence.

For example, consider the two methods in Figure 5. Method `VB` is in VisualBasic 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 [13], the outcome is `sat`, indicating these are functionally equivalent.[5] 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 [38, 104], for simplicity, we use an integer example.

In the event that the solver returns `unknown` due to constraint complexity, a back-up plan uses fuzz testing [22, 58] to determine with some empirical certainty if the methods are the same or behaviorally close; prior work has used fuzzing and semantic analysis for plagiarism detection [58]. Using fuzzing 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".

3.2.2 Differentiating Input

When methods are not identical, there exists at least one input on which the methods behave differently. End-user programmers may want answers to the question, "how are these methods *different*?" [43]. Identifying one, or a class of inputs, can be informative 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 5 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. Contrastingly, 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 3.2.3).

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 [74] to recognize categorical data and exploit known boundary values for a domain.

3.2.3 Similarity

When programmers are learning to use a language, they often make simplifying and invalid assumptions about programming language structures [42]. Research has shown that certain language constructs are more prone to invalid assumptions, specifically conditionals, Boolean operators, loops and data structures [43]. Demonstrating behavioral similarity between fragments within the same language can illustrate alternate implementations to aid comprehension.

Consider, for example, the methods in Figure 6(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 or empirical techniques) 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 be used to determine classes of similarity. For example, we could assert that two methods are equivalent for all integers $x|x < 0$, or for all strings smaller than five characters, or for all lower-case letters in the English alphabet. Model counting [15] could help 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 constraints representation provides several options in describing similarity. If proving equivalence returns `unsat`, the unsat core identifies the set of assertions that are mutually unsatisfiable.

---

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

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

Similarity between two methods could be measured based on the size of the unsatisfiable core. Another approach could use MaxSAT [53] 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. For example, in Figure 6, 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` $\subset$ `original` and `sliced` $\equiv$ `addAB`. Further, using a canonicalization on the constraint representations and longest common sequence of constraints could identify a notion of similarity. Constraint reuse may also provide a notion of similarity [98]. 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 dataflow and control-flow graphs [45,47,92]. My prior work in code similarity analysis [92] considers end-user programs in Yahoo! Pipes 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 [10] may be a practical formalism to describe code similarity when mutation describes the differences between programs. For the code that differs in small structural ways, such as an arithmetic operator, those methods form a product-line, where there is the variation is the operator to select. Re-framing Figure 6 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 add-on to the method. Paired with identifying a differentiating input per Section 3.2.2, it could illustrate an instance of how the code differs.

Empirical Similarity: If semantic and structural analyses fail, fuzz testing is a well-studied approach to generating random inputs for source code. Comparing output values quantifies similarity.

3.2.4 Evaluation

When the solver says two methods are equivalent based on their constraint representations, as a sanity check, we will assess the validity of that claim. The techniques in Thrust 2 will be evaluated in three contexts: cross-language clone detection, refactoring validation, and understanding code changes.

Refactoring Validation: Refactoring is a semantics preserving transformation over source code [18], and there is substantial evidence that end-user programmers maintain and restructure their code [29,85,92]. 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 prior work on smells in spreadsheets [26–29] 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 refactorings 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 would help them understand the impact of their code changes. This will be evaluated using open source Python and VisualBasic repositories from GitHub. An evaluation would use an A/B test with and without the similarity analysis, and 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. In this scenario, a software architect must identify the relevant components of the acquired product, often in a different language from the existing codebase, and then integrate the new code into the architecture of the existing product. A key challenge in the identification step is understanding how the two systems implement their common features, which differ on language, naming conventions, and structure. This is a scenario frequently faced by my industrial partner, ABB. Of particular interest to them are the approaches 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 access to developers for interviews, observation, and evaluation.

3.3 Thrust 3: Reducing and 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*. 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 end-user programmers for additional examples to refine the solution space is rapidly becoming the next bottleneck in semantic code search as well as program synthesis [76]. Thrust 3 proposes techniques to help clients of semantic code search (and synthesis) navigate the solution space, exploring approaches with the human-in-the-loop, such as identifying oracles for given inputs, and the human-out-of-the-loop, such as ranking. From the perspective of the end-user programmer, the impact will making fewer decisions before converging on a desired solution.

3.3.1 Input Selection

In semantic code search or synthesis via input/output example, it is often implied in the literature that a specification can be strengthened to prune the solution space by simply adding another example [78, 89]. However, not 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. The problem is traversing the space of solutions to find the desired solution. To address this challenge, 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, with my collaborators, we tackled this challenge in the domain of data wrangling using Python [76]. 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[6]. In this use case, a fictional end-user programmer wants to analyze San Francisco restaurant inspection data to understand

---

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

(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

the “cleanliness of the city”. The data is available from the city of San Francisco’s OpenData project. The challenge for the scientist 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 to this 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. In our case study with the input/output 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. Further, each program needs to be executed on each input, though the results can be cached. 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 2 (Section 3.2.2) instead of randomly. Regarding clusters, we select the input leading to the number of clusters, but should also consider uniformity in cluster sizes. A cost-benefit analysis will be conducted to determine the impact of the input generation approach and the number of inputs on the effectiveness of the approach.

3.3.2 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, and possibly others.

Ranking by Semantic Popularity: Regardless of whether the solution space was built by finding existing code or abstraction/mutation were used, there are likely to exist behavioral similarities among the solutions. I propose to exploit behavioral similarity, as measured in Thrust 2, 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, 19], 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. For code reuse, it was determined 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) [90]. Users often provided input/output specifications that under-approximated the desired behavior, so those solutions with more behavior are more likely to satisfy the intended specification. 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 [73]. We replicated that result on two different end-user language domains, web mashups, and GreaseMonkey scripts [35]; such program characteristics could be used for ranking based on reuse probability.

3.3.3 Evaluation

Navigating the solution space via input selection depends on user involvement. The assumption that the cost of creating/evaluating an output is lower than the cost of providing a differentiating input/output example from scratch requires exploration. To begin, we will design user studies that use real data sets, such as the San Francisco restaurant data used in Section 3.3.1. Success will be determined by measuring the time and effort to the user in selecting outputs for an input. This cost 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 [88], but this was not evaluated against the cost of generating examples from scratch.

Success in ranking will use the input/examples examples from recent work in synthesizing SQL queries from input/output examples [100], and compare against semantic search for SQL queries [89]. Comparing the synthesized SQL queries to the searched-for SQL queries in terms of understandability, generalizability, and relevance will require human evaluation. In terms of study subjects, North Carolina State University has a Computer Science Masters program with a track in software engineering for which research or participation in research studies is a requirement. This free pool of qualified participants are currently employed full time in industry or looking to obtain an industry job, and thus are representative of people who would be performing code search tasks, thought not end-user programmers. If successful with trained programmers, we will evaluate these on students in entry-level computer science courses at North Carolina State University when they are first exposed to MySQL.

4 Related Work

Specifications used in previous semantic code search work include formal specifications [21,66,103] and test cases [67, 69]. Formal specifications allow precise and sound matching but must be written by hand, which is difficult and error-prone. Test cases are more lightweight but require the code to be executed to identify a match, and thus cannot identify *close* or *partial* matches. Recent work in keyword-based search has begun to incorporate semantic information for finding working code examples from the Web [40] or reformulating queries for concept localization [24]. Several code search engines or example recommendations tools exploit structural information, such as Strathcona [33], Sourcerer [2], and XSnippet [72]

Perhaps most similar to my proposed approach is CodeGenie [49, 52], which is a test-driven approach to search based on Sourcerer. Queries are generated based on code features (e.g., naming conventions of missing method). The difference is in the query format, where semantic code search requires behavioral examples rather than textual queries.

This work builds on existing work in symbolic execution [7, 37, 65, 99]. For code that has behavioral interface specifications, those could be exploited to compose constraint-based summaries of behavior [5, 25]. In the absence of code specifications, we characterize the behavior of code statically using symbolic execution.

(more to come)

5 Educational Benefits, Outreach, and Broader Impacts

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

Undergraduate Education: My focus on undergraduate education related to this grant targets newcomers to programming, senior undergraduates, and REU students.

Newcomers: Code search that identifies multiple semantically identical implementations of reusable code could be valuable in education, especially for newcomers or end-user programmers. Characterizing the dissimilarities between a developer's code and an oracle (e.g., as in a programming class) would be useful as well to illustrate when and how a developer's code diverges from the intended code. Test cases provide a sampling for showing when code behavior diverges, but the techniques to more precisely characterize code similarities and differences (Section 3.2) provide more useful information. The techniques will be piloted in entry-level programming classes at NCSU for non-majors, which teach Python.

Senior Undergraduates: I am designing a special topics course on software testing for undergraduates to be piloted in Spring 2018. This course will involve hands-on exercises in performance testing, unit testing, usability testing, integration testing, and regression testing of actual software projects. To demonstrate the importance of building complete test suites, I will design activities that use semantic search to expose students to the variety of code that can satisfy weak input/output specifications.

REUs: Each summer, the software engineering group at NCSU runs an NSF-funded REU (research experience for undergraduates) on the "Science of Software". At this program, 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. While some of the concepts of this grant would be too advanced for that group, the notion of using code search to support software maintenance and testing tasks would be suitable for lectures and REU projects.

Graduate Education: In Fall 2017, I piloted a graduate course, *Automated Program Repair*, for Masters and PhD students. The course was successful and led to a publication based on a student project [77]. A particularly successful unit in the course was on synthesis [59, 60, 63] and semantic code search [39] approaches to program repair. For Fall 2018, I plan to spin off and expand that unit into a graduate course on *Semantic Analysis in Software Engineering*, which will focus on how to use code search and synthesis for software engineering support. A primary focus of the course will be on semantic code search, and I will recruit research assistants from the enrolled students.

Mentor at the Grace Hopper Conference: In 2013, I traveled to the Grace Hopper Conference (GHC) with ≈20 female students from the Computer Science and Electrical and Computer Engineering departments at Iowa State University (though participation is open to all students, regardless of gender identity). In post-conference surveys, students report increased feelings of belonging and enthusiasm for their degree programs. I plan to continue involvement with GHC, support students in presenting their work in the poster sessions, and focus mentoring discussions on balancing an academic career and a family.

Technology Transfer: ABB has written a letter of support for this research, with a particular interest in its applications to cross-language clone detection. While the languages used by ABB are not the primary target of this proposal, extending support for these languages is a stretch goal. ABB has offered access to industrial systems to test techniques and access to developers for interviews, observation, and evaluation.

Research Artifacts: Reproducibility is a key component of the scientific process. 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 [85, 87–90], I will make all artifacts available on a central project website for the project for other researchers to use in replications or comparison studies.

Student Mentorship: The PI is a junior faculty and is establishing her research program. This project contributes to mentoring and supporting graduate and undergraduate students. The PI has, and will continue to engage undergraduate (including in REU settings) students, minority, and female students. Of her three PhD students at the time of submission, two are female and one is a veteran.

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 [8] 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 adapting semantic code search to fix faults in professional programming languages [39]. understanding how professionals search for code [71], crowdsourcing [95,96], and automated repair [3,39,51,62,79]. **Broader impact:** The benchmark datasets developed for automated program repair [51] will advance research in the automated repair field by standardizing evaluations and enabling comparison among techniques (e.g. [79]). *The proposed work in semantic code search for end-user 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 on professional programming languages. This CAREER proposal targets different languages, including MySQL, Excel, VisualBasic, and Python. The advances in this proposal, specifically the abstractions/mutations for end-user languages, distinguishing between matches, navigating the solution space and ranking, are new.*

7 Research Plan

(TODO: finish me) Figure 8 maps out the breakdown of work per year. Two graduate students and the PI will perform the proposed research under the following schedule:

	Year 1	**Year 2**	**Year 3**	**Year 4**	**Year 5**
Abstractions (3.1.1)	x				
Mutations (3.1.2)		x			
Equivalence (3.2.1)		x			
Differencing (3.2.2)			x		
Similarity (3.2.3)			x		
Input Selection (3.3.1)				x	
Ranking (3.3.2)					x

Figure 8: Breakdown of proposed work per year

Year 1: The focus will be on building abstractions and lattices to amplify the search space (Thrust 1).

Year 2: Similarity metrics and characterizing similarity (Thrust 2) will improve the expressiveness of the lattices.

Year 3: Solution space navigation will begin (Thrust 3).

Year 4: Continue to add abstractions, mutations, and techniques for building effective lattices. Evaluation will begin with ABB for cross-language clone detection.

Year 5: Continue to explore provable equivalence and evaluation in refactoring verification.

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] A. Borgida and P. Devanbu. Adding more ?dl? to idl: towards more knowledgeable component inter-operability. In *Proceedings of the 21st international conference on Software engineering*, pages 378–387. ACM, 1999.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. Fisher and G. Rothermel. The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

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

[27] 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.

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

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

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] Symbolic PathFinder, December 2012.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] S. K. Kuttal, A. Sarma, and G. Rothermel. On the benefits of providing versioning support for end users: an empirical study. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(2):9, 2014.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, New York, NY, USA, 2012. ACM.

[62] 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.

[63] 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.

[64] R. R. Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 10(2):15–21, 1998.

[65] 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.

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

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

[68] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *Software: Practice and Experience*, 21(6):539–556, 1991.

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

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] D. Singh, V. Sekar, B. Johnson, and K. Stolee. Evaluating how static analysis tools can reduce code reviewer effort. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[78] 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.

[79] 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.

[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. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

[82] K. T. Stolee. Analysis and Transformation of Pipe-like Web Mashups for End User Programmers. Master's Thesis, University of Nebraska–Lincoln, June 2010.

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

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

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

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

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

[95] 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.

[96] 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.

[97] 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.

[98] 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.

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

[100] 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.

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

[102] 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.

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

[104] 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.