CAREER: Semantic Code Search for Software Testing and Maintenance

PI: Kathryn Stolee

A. Project Summary

Overview

Semantic code search is a technique developed to allow clients, whether it is a user or an algorithm, to find code based on behavior specified using input/output examples. The engine behind the search is a constraint solver; code snippets are indexed using symbolic execution to generate a constraint representation of the code's behavior. Given a specification and a snippet, the solver returns `sat` if the code satisfies the specification. Originally implemented in an end-user programming language called Yahoo! Pipes, further efforts involved extending it to subsets of Java, C, and MySQL. In recent research, the PI has successfully applied semantic code search to automatically repair C programs with patches that are higher quality in the sense that they overfit less to the specification, when compared to patches from other approaches. Additionally, for code reuse, the PI has shown that the snippets returned by semantic code search are significantly more relevant than those produced by state-of-the-art search based on structural components.

While promising, in pursuing the development of semantic code search for code reuse and program repair, some common issues have emerged. In particular, 1) finding *close matches* when an exact match for a specification does not exist, 2) characterizing the differences between two snippets of source code, and 3) navigating a solution space of programs that all match a specification, are common issues across these application areas. The PI proposes to perform basic research in each of those areas, exploring techniques to relax program encodings, determine when and how code snippets differ in behavior, and quickly converge on the desired program from a large space of solutions. Additionally, the PI will explore how semantic search can provide value in three new application areas within software engineering: cross-language clone detection, refactoring verification, and test-driven development. My vision is to integrate semantic reasoning into software testing and maintenance activities to facilitate more informed code changes, promote reuse and adaptation of high-quality artifacts, and, ideally, reduce developer effort.

Intellectual Merit

If successful, this work will substantially improve the power of semantic code search and by extensions, the domains that can benefit from semantic analysis, such as cross-language clone detection, refactoring verification, and test-driven development. 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 source code snippets, including proving equivalence of methods, identifying differences between snippets, and characterizing similarity.
3. Approaches to navigate the solution space after search or synthesis to quickly converge on the desired program, including human-in-the-loop and human-out-of-the-loop solutions.

Broader Impact

The proposed work has three expected broader impact contributions:

1. The resulting techniques will likely have a significant impact on industry by streamlining common software development processes, including cross-language clone detection, refactoring verification, and test-driven development, possibly resulting in fewer bugs due to increased understanding of code behavior.
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 their code.
3. The PI has a track record of mentoring women in computing by working with female undergraduate and graduate research assistants and attending the Grace Hopper Celebration of Women in Computing conference with university students.

CAREER: Semantic Code Search for Software Testing and Maintenance

PI: Kathryn Stolee

C. Project Description

1 Introduction

As the quantity of source code continues to increase, especially in the open source domain, finding existing code to reuse or learn from becomes more difficult. With so much code to sift through, it is difficult to know whether the desired code exists, or how to differentiate between two similar pieces of code. For the problem of finding code, a common approach to find code is through search [2].

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 "`Java compute triangle type`". In contrast, *semantic* search uses behavioral properties as the specification, for example, a developer could write an input/output example that demonstrates the desired behavior of code, such as the input `{2, 3, 4}` and output, "`scalene`". Semantic code search can be achieved through executing source code [53] or mapping source code into constraints representing its behavior and using a constraint solver to identify matches for a query [28,63,65,68,69]. It has been used in three applications: code reuse [53,63,65,68,69], program repair [28], and finding API usage examples [44].

I propose to build on my previous work in semantic code search via constraint solver [28,63,65,68,69]. In semantic code search, queries take the form of input/output examples, a search space is composed of code snippets translated into a constraint-based representation, and matching uses a constraint solver to determine if a code snippet behaves as specified. Beyond 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**. I also propose to explore three new applications for semantic code search: cross-language clone detection, refactoring verification, and test-driven development.

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 [68,69,76], they can be extracted from programs at runtime to provide specifications for program repair [28], program synthesis approaches depend on them to describe code to synthesize [20,59,76], in refactoring, input/output is used to verify behavior preservation [48], and input/output examples are the cornerstone of programming-by-example systems that target notices or end-user programmers [10,37]. 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 targets the aforementioned challenges related to semantic code search via input/output example, when 1) there are too few solutions, 2) it is difficult to differentiate between similar examples, and 3) there are too many examples:

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. Abstractions identify code that is semantically close, whereas mutations identify code that is syntactically close, to the desired code. To address this challenge, **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 constraint solver to identify changes to the program that can lead to desired code. These approaches will be evaluated in the context of test-driven development.

Thrust 2: Characterizing Semantic Similarity: For when the differences between programs are unclear, code snippets must be comparable based on their behavior. This way, users can understand how two pieces of code differ or programs can be ranked based on common behaviors. **I propose to use the constraint representations of code snippets to characterize program similarity and differences**, which will enable cross-language clone detection, a challenge of particular interest to my industrial partner, ABB (see letter of support). It will also facilitate 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 synthesis or semantic search, navigating the space of potential solutions is a pervasive problem, as the solutions can be numerous [57]. **I propose to develop techniques to rapidly prune the solution space by identifying inputs that maximally fragment the space of solutions 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 [20, 57, 59, 76].

If successful, this research will transform state-of-the-art constraint-based semantic code search search, and its applications, through techniques to better guide the search to converge on a desired program. It will also expand the applications that benefit from semantic code search to include cross-language clone detection, refactoring verification, and test-driven development.

2 Preliminaries: Semantic Code Search

Originally implemented in an end-user programming language called Yahoo! Pipes, the goal of semantic code search was to identify existing programs that behave as specified given input/output examples [65, 68]. Further efforts evolved semantic code search to subsets of Java [68, 69] and MySQL [68]. For applications in program repair, together with my collaborators, we extended the support to a subset of the C language [28].

Semantic code search 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, line 1 of the source code in Figure 1(a) maps to lines 4–7 of the constraints in Figure 1(b). The first constraint on lines 4–5 defines `up` as the location of `` `@' `` in `full` or as `-1`, and lines 6–7 assert that `up` is the first index of `` `@' `` in `full`, per the semantics of the `indexOf` method in `java.lang.String`. The constraint on lines 8–9 represent lines 2–3 of source code. It asserts that `alias` matches `full` within bounds of `0` and `up`, per the semantics of the `substring` method in `java.lang.String`. 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 "`sporty`", the search engine will first identify all methods that take a string as input and a string as output. From there, it will translate the input and output into constraints with assigned variables, and bind those variables to program constructs. With the

Figure 1: Example constraint-based encoding of Java snippet

```
1  int up = full.indexOf('@'});
2  String alias
3    = full.substring(0, up);
```

(a) Java snippet

```
4  (assert (OR
5    (full.charAt(up) = '@') (upper = (-1))))
6  (assert (forall ((i Int)) (=> (0 <= i < up)
7    (not (full.charAt(i) = '@'))))
8  (assert (forall ((i Int)) (=> (0 <= i < up)
9    (alias.charAt(i) = full.charAt(i)))))
```

(b) Constraints for the Java snippet. Lines 4–7 map to `indexOf`. Lines 8–9 map to `substring`.

example in Figure 1, the input could be bound to `full` and the output to `alias`. Together with the program encoding, the input/output constraints and bindings are given to an SMT solver. The result `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 left up to the solver [68]. In this way, parameter ordering is not an issue, as it can be with search that involves execution [53]. For example, if we have an specification input with {`5, 3, 4`} and a method `isPythagorean(int a, int b, int c)`, semantic code search encodes all possible mappings and lets the solver select one that works, if it exists. That is, constraints are encoded for all 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, for the pythagorean theorem that $c=\sqrt{a^2+b^2}$, only two mappings where $c=5$ lead to a satisfiable solution. We make only one solver call to find a combination that work, instead of executing the method up to five times before finding a correct parameter ordering.

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. For program repair, the SearchRepair implementation targets C code, supports all C primitives and constructs from the Java implementation, console output, and additionally encodes `char*` variables, the modulo operator, and the string library functions: `isdigit`, `islower`, `isupper`, `strcmp`, and `strncmp`. The limitations with the current semantic search rest largely on the limitations of symbolic execution, but also, as symbolic execution gets more powerful, so does semantic code search. The C and Java implementations are based on KLEE [7] and SPF [26, 49, 75], respectively. For C, KLEE is the state-of-the-art in symbolic execution, and recent efforts have improved, for example, performance [54] and the underlying theories [13]. For Java, SPF is the state-of-the-art tool, and recent efforts aim to improve, for example, heap summaries [23, 24] and efficiency through statistical probabilities [5, 8, 14, 40].

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 and MySQL (already explored), or languages such as R, MatLab, or Excel (yet to be explored). Using smaller languages often removes the convenience associated with using existing tools such as KLEE and SPF. However, writing code that transforms programs into constraints for smaller languages is certainly achievable. In fact, this is where semantic code search started; I 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. Different parts of this proposal explore different languages, or subsets of languages to demonstrate simpler versions of the techniques before scaling.

3 Proposed Research

My broad research vision is to bring the benefits and power of semantic analysis into all stages of the software development lifecycle. Challenges face all applications of semantic search, from the previously explored applications of program repair and reuse to the new applications of cross-language clone detection, refactoring verification, and test-driven development. In all cases, techniques are needed to handle scenarios when 1) there are too few solutions, 2) it is difficult to understand how solutions differ, and 3) there are too many solutions. To tackle these challenges, I propose the following:

Thrust 1: When there are too few solutions, techniques are needed to expand the scope of code that can be modeled, and also find approximate solutions when an exact one does not exist.

Thrust 2: When it is difficult to determine the differences between two snippets, techniques are needed to describe similarities and differences in the semantics, which can be derived from the constraints.

Thrust 3: When there are too many solutions, techniques to navigate a large space of possible solutions are needed, including selecting inputs that maximally divide the solution space and ranking the solutions.

Figure 2: Example specification and close match

Test	Input	Output
1	0 0 0	Not a triangle
2	1 1 1	Equilateral
3	2 2 3	Isosceles
4	2 3 4	Scalene
5	2 3 5	Not a triangle

```
int sides(int a, int b, int c) {
  if(a <= 0 || b <= 0 || c <= 0)
    return 0; // invalid, negative
  if(((a + b) <= c) || ((a + c) <= b) || ((b + c) <= a))
    return 0; // triangle inequality
  if(a == b && a == c)
    return 1; // equilateral
  if(a == b || a == c || b == c)
    return 2; // isosceles
  return 3; // then it's scalene
}
```

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, in an industrial setting, it may be desirable to limit code reuse to within an organization. In an academic setting, it may be desirable for a student to limit reuse to just code written by them. In these scenarios, the quantity and variety of code to reuse is likely more limited, increasing the likelihood that the desired code does not exist, but something close might. For example, in a preliminary exploration of semantic code search in C [28] applied to repairing bugs in the ManyBugs [34] benchmark dataset, 17/41 bugs were patched. The patch database was built per-project. One of the main reasons for the unpatched bugs was that there was no match in the database. This motivates a need for finding matches that are behaviorally or structurally close to snippets in the search database.

To find approximate matches, we need a notion of similarity between code and a specification to determine how well code meets a specification (which is notably different that similarity between two pieces of code, explored in Thrust 2 in Section 3.2). A *semantic match* means the code satisfies the specification. Semantic *similarity* 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 type mismatch), but the code can be easily modified to be a match through a form of interface adaptation [52].

For example, consider the specification table with five test cases in Figure 2. The desired solution contains three integers as input and one string as output, and computes the type of a triangle given the lengths of the three sides, returning "not a triangle", "equilateral", "isosceles", or "scalene". The method, `sides`, in Figure 2 provides a solution that comes close to the intended program, except for the return type and values. In this case, behavioral subtyping [38] cannot be exploited, as it might between floats and integers. Instead of returning `0`, if the method returned, "`Not a triangle`", it would be a perfect match for tests 1 and 5 (and the others, assuming appropriate return values). However, for current approaches to semantic code search, `sides` would not be a candidate program solution based on the type signature mismatch alone. However, with slight modifications on the return type and return values, it behaves exactly as specified. Thrust 1 explores how to make this possible, exploring the abstractions and mutations necessary to facilitate a match.

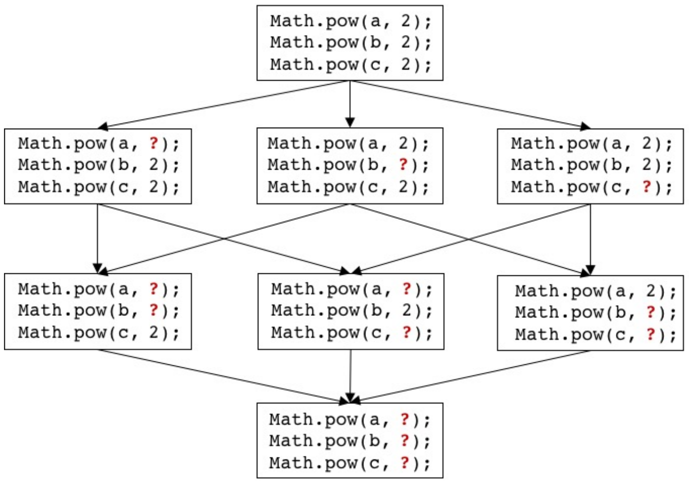
3.1.1 Abstractions

Abstractions systematically weaken a snippet's encoding to broaden the set of specifications it can potentially match. A straightforward approach to abstraction involves weakening hard-coded primitive values, such as integers, characters, or floats. For example, consider a method that returns the the character `'a'` under one abstraction could allow the method to return any vowel, or another level could allow it to return any character on the standard English keyboard, or even any unicode character. The SMT solver will identify appropriate values for the weakened variables in the satisfiable model, providing guidance on how to modify the program to fit the specification.

Figure 3: Abstraction Lattice and Example

```
1 boolean isPyth(int a, int b, int c){
2   int a2 = (int)Math.pow(a, 2);
3   int b2 = (int)Math.pow(b, 2);
4   int c2 = (int)Math.pow(c, 2);
5   if(c2 == (a2+b2)){
6     return true;
7   }
8   return false;
9 }
```

(a): Example Code



Math.pow(a, 2); Math.pow(b, 2); Math.pow(c, 2);
Math.pow(a, ?); Math.pow(b, 2); Math.pow(c, 2);
Math.pow(a, 2); Math.pow(b, ?); Math.pow(c, 2);
Math.pow(a, 2); Math.pow(b, 2); Math.pow(c, ?);
Math.pow(a, ?); Math.pow(b, ?); Math.pow(c, 2);
Math.pow(a, ?); Math.pow(b, 2); Math.pow(c, ?);
Math.pow(a, 2); Math.pow(b, ?); Math.pow(c, ?);
Math.pow(a, ?); Math.pow(b, ?); Math.pow(c, ?);

(b): Abstraction lattice for (a) over integers

In preliminary work, I have explored this for Yahoo! Pipes [63, 68] with abstractions on the string and integer values. This amplified the search space, but at the cost of speed. Abstractions can be applied globally (i.e., to all strings, integers) or to an instance (i.e., one integer or string at a time)s. Abstractions can be applied at random or strategically, possibly targeting return statements, if-conditions, or loop-conditions.

I propose to model relationships between abstracted representations using a lattice in which the behavior in one level is subsumed by another. That is, if a specification is satisfied at one level of the lattice, it is also satisfied at all levels below it. Since subsumption is a strong criteria, I plan to explore the various specification matches outlined by Zaremski and Wing to model relationships between code behaviors [78]. For example, consider the code example in Figure 3(a) and lattice in Figure 3(b). Written as-is, the code represents the top level of the lattice. As values are relaxed, we move down the lattice. Relaxing the hard-coded value `2` on line 2 of the method is represented as `Math.pow(a, ?);` in the lattice. The top of the lattice has the strongest constraints and the bottom has the weakest. If the code satisfies a specification at one abstraction level, it also satisfies the specification at lower levels. That is, with input `a=3, b=4, c=5` and the output is `true`, all encodings match the specification. If instead `a=1, b=2, c=3` and output is `true`, only those levels in which `Math.pow(a, ?);` and `Math.pow(b, ?);` are abstracted will satisfy the specification, since that would check if $1^3 + 2^3 == 3^2$, replacing each of the abstracted integers with the number three.

There are several dimensions to explore in abstracting encodings, including 1) where to abstract (e.g., return statements, if-conditionals), 2) how much to abstract (e.g., globally, instances), and 3) what to abstract (e.g., variable values). Once the lattices are built, other challenges concern how to connect the lattices by proving equivalence between representations (Section 3.2.1) and where in the lattice to begin when searching for a match. While lower levels of the lattice will be more likely to match, those require the most adaption. Higher levels are less likely to have a match, but allow reuse of code that is closer to what a developer wrote. Beyond abstracting variable values, I propose to explore the potential of building abstraction lattices using program slicing [77], relational operators, and behavioral subtyping [38]. The tradeoffs in tweaking each of these dimensions will be explored.

3.1.2 Mutations

As opposed to abstractions, when mutating a program, the relationship of the original program to the modified program is not entirely straightforward. For example, replacing the + operator on line 5 of Figure 3(a) with the – operator will results in different semantics, but the relationship is not entirely clear. Still, mutations expand the search space and enable finding a match that is syntactically and structurally similar to existing code. To ensure the mutants indeed expand the search space we will verify that they are reachable and not equivalent [73].

In amplifying the search space via mutation, the idea is similar to abstraction of hard-coded value where parts of the program are replaced by "holes" that the SMT solver fills in when finding a satisfiable model for the constraint system, similar to synthesis by sketching [17,61,62]. The main difference is that the code skeletons in my approach come from existing code written by others. This is where semantic code search begins to blur the line between search and synthesis, since the solver needs to synthesize a solution before it can return `sat`. I propose to explore four mutations to expand the search space.

Arithmetic and Relational Operators: Inspired by mutation testing [73], where test suite quality is measured by its ability to kill mutants by behaving differently on the mutated code compared to the original code. For example, in an arithmetic expression, such as `a+b`, this abstraction could replace the + operator with another operator in the group: {`-, *, /, %`}.

Library Calls: For libraries that are supported by semantic code search, an abstraction on the library call itself would increase the search space. For example, in the `java.lang.String` library, the two methods, `int indexOf(String str, int fromIndex)` and `int lastIndexOf(String str, int fromIndex)` both have the same type signature and could be replaced prior to encoding, as a mutation.

Variable Replacement: This mutation involves replacing one or more variable in a code fragment with another, assuming the same scope. For example, if there are multiple global variables available, the one in the snippet would be replaced with another in the mutated version. These mutations would occur within the variable scope. Swapping the parameter order is built-in to the search process, as described in Section 2.

Interface: The return type and input type(s) can be abstracted independently. The example in Figure 2 illustrates a case when abstraction on the return type may be useful. Another form of type signature abstraction involves the input parameters. These can be added, deleted, or modified. For deleting an input parameter, a forward slice on that variable [77] would identify all the code that depends on the variable and could be removed (which may also remove additional input parameters, based on the data dependencies). This is appropriate when the type signature of the input/output examples is a subset of the type signature of a method under consideration.

3.1.3 Evaluation

Our preliminary work in program repair provides one instance where we observed exact matches were rare. As a new direction, I propose to explore test-driven development (TDD). To begin, we will extract the examples from Kent Beck's textbook on TDD by example [4], which include TDD examples in Java, generally limited to primitives, as supported by our current implementation. Next, we will use a TDD dataset developed as part of a continuous testing study [45], which is more complex in its use of Java objects. After building an initial repository and before abstractions, we will measure the ability of the tool to generate methods that pass all the tests. Then, using each (or all) of the abstractions and mutations, we will repeat the experiment. Measuring efficiency and effectiveness, we will explore the most effective mutations and abstractions based on the context. By working with my industrial partner at ABB, we can evaluate the abstraction and mutation techniques using a database populated with just their industrial code and tasking their developers to use our tools to write programs.

Support for these programming tasks will be implemented as an Eclipse plugin. This will facilitate developer experiments and also provide a convenient delivery mechanism to developers at large. We will build a large repository of programs to be hosted in 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 and improve the comparison operators without impacting the plugin users. The Neo4J graph-based database will be explored to explicitly model relationships among the encodings.

Figure 4: Equivalence examples: each group of equivalent methods has an identified similarity measure.

```
int addTwo(int x) {
  int a = x + 2;
  return a;
}

int multTwo(int x) {
  int d = x * 2;
  return d;
}
```

(a) Equivalent with wild card on arithmetic operators

```
1  void first() {
2    int x = 1;
3    int y = 2;
4    int z = y-2; //*
5    int r = x; //*
6    z = x + y;
7  }
8
9  void second() {
10   int a = 2;
11   int b = 1;
12   int c =  a + b;
13 }
```

(b) Equivalent with backwards slicing in **first** on variable **z** on line 6. Lines with //* are not retained in the slice.

```
int X(int x, int y, int z) {
  return x;
}

int Y(int x, int y, int z) {
  return y;
}

int Z(int x, int y, int z) {
  return z;
}
```

(c) Equivalent with wild card on return variable

3.2 Thrust 2: Differentiating Between Solutions

Given two methods or code fragments, determining similarity and differences is important for users to understand which will meet their needs. It is also important to support Thrust 1 (Section 3.1) in determining the similarity between abstracted and mutated code so lattices can be combined.

Consider, for example, the groups of methods in Figure 4(a)–(c). In Figure 4(a), the methods are structurally equivalent except for the arithmetic operator (could be formed by mutating on the arithmetic operator, per Section 3.1.2). Behaviorally, these two methods will behave the same when **x=2** but differently for all other values of **x**. In Figure 4(b), these methods are different in the number of variables declared, number of arithmetic operators used, and number of assignments. However, backwards slicing on **first** on the variable **z** reduces **first** a set of three variables. Considering **second**, these methods are equivalent when **x**$\mapsto$**b**, **y**$\mapsto$**a**, and **z**$\mapsto$**c**. In Figure 4(c), these methods are structurally equivalent except they each return a different variable. They behave the same when **x == y == z**, but otherwise they behave differently.

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, providing a more efficient storage structure and traversal of the constraint database, refactoring verification, and cross-language clone detection. For simplicity, we focus on the method level of granularity, though scaling to class, sub-system, and system levels is of particular interest to ABB, my industrial partner, and is within the long-term vision of this work.

3.2.1 Proving Equivalence

While undecidable in general, we can use state-of-the-art SMT solvers such as Z3 [12] 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 use of a common AST [31]). Instead, representing the code as constraints provides a common abstraction, and the solver can determine equivalence.

For example, consider the two methods in Figure 5. Method **A** is in Java and **B** is in Python. While structurally different, the return value for both is **x*x*6**. The constraints on lines 17–21 represent method **A** and

Figure 5: Example programs for proving equivalence

```
1  // Java method
2  int A(int x) {
3    int a = x * 2;
4    int b = x * 3;
5    int c = a * b;
6    return c;
7  }

8  # Python method
9  def B(x):
10     d = x * x
11     e = d * 6
12     return int(e)
```

```
13 (set-logic UFNIA)
14 (declare-fun A (Int) Int)
15 (declare-fun B (Int) Int)
16
17 (assert (forall ((x Int))
18   (exists ((a Int)(b Int)(c Int)(output Int))
19     (and
20       (= (A x) output) (= a (* x 2)) (= b (* x 3))
21       (= c (* a b))     (= output c)))))
22 (assert (forall ((x Int))
23   (exists ((d Int)(e Int)(output2 Int))
24     (and
25       (= (B x) output2) (= d (* x x))
26       (= e (* d 6))      (= output2 e)))))
27 (assert (forall ((x Int)) (= (A x)(B x))))
28 (check-sat)
```

lines 22–26 represent method `B`. Line 27 asserts that for all integers, the two methods behave equivalently. When provided to Z3, the outcome is `sat`, indicating these are functionally equivalent.[1] While a simple and contrived example, it shows the potential of using Z3 for cross-language clone detection.

In the event that the solver returns `unknown` due to model complexity, a back-up plan uses fuzz testing [19, 41] 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 [41]. 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, which the overwhelming majority of scenarios, there exists at least one input on which the methods behave differently. 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 27 in Figure 5 to `(assert (exists ((x Int)) (not(= (A x)(B x)))))`. Instead of asserting the methods are identical under all integer inputs, this revised line asks for an input on which the methods differ. If `sat` is returned, the satisfiable model identifies one such input. Contrastingly, removing the `not()` operator will identify an input on which the methods `A` and `B` are the same, if one exists, leading to the next section on describing similarity.

As wth proving equivalence, a back-up plan would use fuzzing to differentiate between methods that cannot be handled by the solver when the snippets are compilable.

3.2.3 Similarity

Modeling similarity is important for efficient database organization and traversal. I propose to quantify similarity using strong measures (e.g., via constraint representation) 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. This could form equivalence classes where we characterize the conditions under which the methods are alike, inspired by equivalence class testing [1] which breaks the input space into clusters

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

such that any input from a cluster is expected to behave similarity. 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 (Figure 5) returns `unsat`, the unsat core identifies the set of assertions that are mutually unsatisfiable. Similarity between two methods can be measured based on the size of the unsatisfiable core. Another approach would be to use MaxSAT [36] to measure the maximum number of clauses, or the percent of clauses, that are satisfiable. Similarity can also be measured using abstractions, as described in Section 3.1. For example, in Figure 4(b), the methods are equivalent after backwards slicing on `z` in `first`. Thus, it is possible to conclude that the behavior of `second` $\subset$ `first`. Further, using a canonicalization on the constraint representations and longest common sequence of constraints between two methods could identify a notion of similarity. Constraint reuse may also provide a notion of similarity [74]. 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 Modeling data-flow and control-flow with graphs and looking for isomorphisms is a common way to explore structural similarity [30, 32, 70]. Product lines [9] may be a practical formalism to describe code similarity when mutation describes the differences between programs. For the code in Figure 4(c), these methods seem to form a product-line, where there is an option on which return variable to select. Similarly, the methods in Figure 4(a) also form a product line with an option on the operator in the first assignment statement, either + or `-`. Re-framing Figure 4(b) as a product line, lines 4–5, 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.

My prior work in code similarity analysis [70] consider end-user programs in Yahoo! Pipes and looked for structural similarity considering several levels of structural abstraction.

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

There are objective and subjective evaluations needed for this thrust. Objectively, when the solver says two methods are equivalent based on their constraint representations, checking the validity of that claim is needed. Subjectively, when the proposed analysis claim a quantitative level of similarity between two pieces of code, the actual similarity must be evaluated from the perspective of the client, whether user or algorithm. The techniques in Thrust 2 will be evaluated in three contexts: cross-language clone detection, refactoring validation, and understanding code changes.

Cross-language clone detection Say a company acquires a competitor and needs to integrate features from a newly acquired 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 scenario frequently faces 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 out the techniques and access to developers for interviews, observation, and evaluation.

Refactoring Validation Refactoring is a semantics preserving transformation over source code [15]. Best practices indicate that the developer should have a passing test suite prior to refactoring, and that the code should pass after refactoring. That is, as long as the code after refactoring passes, it is deemed to be a suc-

Figure 6: Solution Space Pruning Example

Test	Input	Out
1	0 0 0	0
2	1 1 1	1
3	2 2 3	2
4	2 3 4	3

```
// matches tests 1, 2, 3, 4
int middle (int a, int b, int c) {
  return b;
}

// matches tests 1, 2
int mult(int a, int b, int c) {
  return a * b * c;
}

// matches test 1
int sum(int a, int b, int c) {
  return a + b + c;
}

//matches IDs 1, 2
int weird(int a, int b, int c) {
  return (a + b) - c;
}
```

```
// matches tests 1, 2, 3, 4
int triangle(int a, int b, int c) {
  if(a <= 0 || b <= 0 || c <= 0) {
    return 0; // invalid, negative
  }
  if(a == b && a == c) {
    return 1; // equilateral
  }
  if(a == b || a == c || b == c) {
    return 2; // isosceles
  }
  return 3; // then it's scalene
}
```

cessful refactoring. However, as tests are often a weak specification for intended code behavior, soundness and completeness are not guaranteed. Using existing refactoring datasets [27] 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 of the refactorings can be validated by these techniques. In the event that refactorings are not equivalent, using the test cases with the original code, input/output specifications can be extracted via dynamic analysis [28] and used to search a database for code with a different structure, but same semantics, as a way to automate refactoring. The differentiating input can then be added as a test case to improve the robustness of the test suite.

Understanding Code Changes Given a method before and after a change, the similarity analysis would reveal how and when the behavior is different, considering the original method. In the hands of users, this could help them understand the impact of their code changes. Success would be measured in an A/B test with and without the similarity analysis, and ask participants to describe the impact of code changes.

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 the burden of the human or algorithm using the search. Independent of the client, lowering this burden implies having more complete and precise user specifications (i.e., more input/output examples). Requesting users for more useful examples to refine the space of synthesized programs is rapidly becoming the next bottleneck in semantic code search, as well as program synthesis [57]. Thrust 3 proposes techniques to help clients of semantic code search (and synthesis) navigate the solution space, using approaches with the human-in-the-loop, such as identifying oracles for given inputs, and the human-out-of-the-loop, such as ranking.

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. However, not all examples prune the solution space effectively. For example, consider the specification in Figure 6 (a subset of the specification in Figure 2(a) except for the type on the expected output) and the set of methods.

(a) Business Information Table (input 1)

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

(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	address	Score	date	latitude	longitude
66793	“CP - ROOM"	“CANDLE PARK"	100	20130112	37.712613	-122.387477

Figure 8: Inputs and Output Example

All methods in Figure 6 satisfy the first test. When adding the second test, the solution space reduces from five to four. However, if instead the fourth test was added, then the solution space would reduce from five to two. The challenge is identifying inputs that quickly allow the client to converge on the desired solution.

In preliminary work, with my collaborators, we tackled this challenge in the domain of data wrangling [57]. Consider the two input and one output tables in Figure 8. The example is from the Zipfian Academy, a group that teaches how to analyze large data sets[2]. In this use case, a fictional data scientist 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 for the scientist is that the data needs some wrangling (e.g., merging, formatting, filtering) before it can be analyzed. For example, the scientist needs to join Figure 8(a), containing business information, and Figure 8(b), containing inspection data, and then filter rows with invalid latitude and longitude values to obtain the output shown in Figure 8(c). Given this example, our synthesis engine produces 4,418 Python programs that perform the transformation.

The problem is traversing the space of solutions to find the one that is intended. 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.

We explored an approach 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 example in Figure 8, three more inputs are needed to reduce the space from 4,418 to two, shown in Figure 7

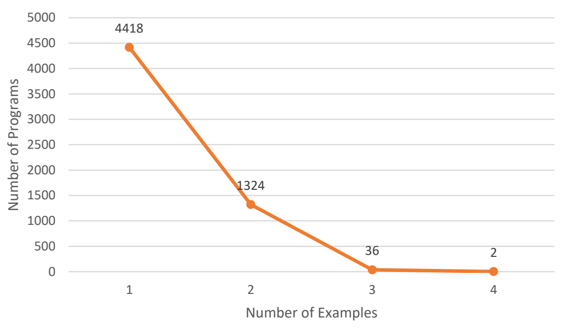


Figure 7: Narrowing of the program space after each intput/ouput example is added to the spec.

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. 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.

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

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 techniques to ranking programs 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. Some contexts may require different ranking approaches. For example, program repair may benefit more from conformance to the specification versus test-driven development, which may benefit more from elements familiar to the developer.

Ranking by Behavioral Similarity Regardless of whether the solution space was built by finding existing code or abstraction or mutation were used, there are likely to exist behavioral similarities among the solutions. I proposed to begin exploring this approach by creating behavioral clusters based on equivalence or high similarity between pairs of programs (e.g., as measured in Thrust 2, Section 3.2.3), sorting the clusters by size from largest to smallest, and then selecting a member from each of the top *n* clusters to be the top *n* results. Given the high redundancy in open source code [3, 16], it would make sense that someone has written the intended code, and possibly many people. Thus, semantic popularity could be useful, particularly when finding code to 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 code snippets to questions asked on StackOverflow [69]. For code reuse, it was determined that code which provides *more* behavior than the specification in terms of program paths were the most relevant. That is, the user-provided input/output specification under-approximates the desired behavior, so those solutions that provide 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 that will be discovered during evaluation.

Familiar Programming Constructs In program repair, higher rankings for code that is similar to the buggy code may make sense since minimal changes to the code may be more understandable to the developer. This is similar in philosophy to techniques that use machine learning to generate fix templates or code that is most like developer patches [39].

3.3.3 Evaluation

Navigating the solution space via input selection depends on user involvement, and the assumption that the cost of creating/evaluating an output is lower than the cost of providing a differentiating input/output example from scratch; this requires exploration, likely in the context of reuse or test-driven development. 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. The technical approach and user studies will use Python as the target language. 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 [67], 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 [76], and compare against semantic search for SQL queries [68]. 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, providing access to a free and qualified pool of study subjects.

4 Related Work

Specifications used in previous semantic code search work include formal specifications [18, 50, 79] and test cases [51, 53]. 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 [29] or reformulating queries for concept localization [21]. Several code search engines or example recommendations tools exploit structural information, such as Strathcona [25], Sourcerer [2], and XSnippet [56]

Perhaps most similar to my proposed approach is CodeGenie [33, 35], 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, 26, 49, 75]. For code that has behavioral interface specifications, those could be exploited to compose constraint-based summaries of behavior [6, 22]. 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 Exploring semantic-search enabled TDD has a benefit for education. I am designing a special topics course on software testing for undergraduates to be piloted in Spring 2018. This hands-on course will involve hands-on exercises in performance testing, unit testing, usability testing, integration testing, and regression testing of actual software projects. For unit testing in particular, if successful, the proposed research will begin to automate the development component of test-driven development. Depending on the strength of the test suite, students may or may not create a solution as intended. This process may have educational benefits to show them first-hand the importance of a robust test suite that covers a variety of behaviors.

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 [58]. A particularly successful unit in the course was on synthesis [42, 43, 47] and semantic code search [28] 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 and focus mentoring discussions on balancing an academic career and a family.

Technology Transfer Semantic code search in Java has piqued the interest of NASA's JavaPathfinder team (JPF). In summer 2017 (and previous summers), they host a Google Summer of Code, for which the PI has a student exploring semantic code search driven program repair in Java. 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.
Additionally, ABB has written a letter of support for this research, with a particular interest in its applications to cross-language clone detection. ABB has offered access to industrial systems to test out 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 [64, 66–69], 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, two are female and one is a veteran.

6 Results of Prior NSF Support

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:** We adapted semantic code search to work in the context of fault fixing [28]. This funding has resulted in advances in code search [11, 55], crowdsourcing [71, 72], and automated repair [3, 28, 34, 46, 60]. **Broader impact:** This work has resulted in career development of two female junior faculty, Claire Le Goues and Kathryn Stolee (the PI). The benchmark datasets developed for automated program repair [34] will advance research in the automated repair field by standardizing evaluations and enabling comparison among techniques (e.g. [60]).

PI Stolee has NSF support for semantic-code search driven automated program repair. The extensions of semantic code search in this proposal are complementary to those efforts and applied to new domains, test-driven development, cross-language clone detection, and refactoring verification. The advances in this proposal, specifically the abstractions, mutations, distinguishing between matches, navigating the solution space and ranking, and the targeted applications, are new.

7 Research Plan

Two graduate students and the PI will perform the proposed research under the following schedule:

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. A TDD prototype will be developed.

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] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] 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.

[6] 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.

[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] 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. Conditionally Accepted.

[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, 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.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] Symbolic PathFinder, December 2012.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.

[39] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, Jan. 2016.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Reducing feedback delay of software development tools via continuous analysis. *IEEE Transactions on Software Engineering*, 41(8):745–763, 2015.

[46] 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.

[47] 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.

[48] W. F. Opdyke. Refactoring object-oriented frameworks. 1992.

[49] 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.

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

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

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

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

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

[76] 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.

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

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

[79] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6:333–369, Oct. 1997.