# Semantics-Based Code Search

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

## Abstract

*Our goal is to use the vast repositories of available open source code to* generate *specific functions or classes that meet a user's* specifications. *The key words here are specifications and generate. We let users specify what they are looking for as precisely as possible using keywords, class or method signatures, test cases, contracts, and security constraints. Our system then uses an open set of program transformations to map retrieved code into what the user asked for. This approach is implemented in a prototype system for Java with a web interface.*

## 1. Motivation

One of the first things a programmer should do when writing new code is to find existing, working code with the same functionality, and reuse as much of that code as possible. With the large amount of open-source code available and the fact that most applications are not completely novel, one could imagine that a significant amount of the code that is being written today has been written before and is available in an open-source repository.

Unfortunately, very little open-source code is reused in this way. There are several reasons for this. The first is that equivalent code is difficult to find. Today's code search engines, for example Google (*codesearch.google.com*), Koders (*koders.com*), and Krugle (*krugle.org*), offer only a keyword-based search and file-level retrieval and generally have limited utility in looking for appropriate code fragments for a particular application. The second problem is that the identified code rarely meets the user's requirements. It might not compute the same function or do the right thing, it might have security or privacy problems, it might be too complex to be understood or reused, it might be too slow, or it might take slightly different parameters or return slightly different values. Moreover, determining these differences is up to the programmer and requires understanding the retrieved code, a difficult task, especially if there are hundreds of poorly-documented search results. The third problem is that even when code can be found that meets one's requirements, that code still has to be modified and adapted by the programmer. Overall, programmers often feel that the effort required here is more than the effort of writing the code in the first place.

The problem of software reuse is an old one, dating back to the 1960's [21,24,25]. There are several aspects of reuse that make it a difficult problem. These include creating reusable code, finding code to reuse, and adapting reusable code to the new application. Each of these problems has been tackled in various ways, but there has not been a comprehensive solution that has really worked. The current state-of-the-art, as seen in the various code search engines cited above, ignores much of this previous work because it was impractical either from the provider's or the requester's point of view.

The use of program semantics in searching for reusable components was common in the 1990s. Originally, this work by Wing looked at matching function signatures [41], but was extended to matching more complete formal semantics using λprolog and Larch-based specifications [31]. More recent work in this area includes the specification language of the CARE system [16], a relation-based approach that relied on semantics-based indexing [26], and a contract-based approach [18]. Other work in this area looked at the use of type systems for specification [32]. Another semantic approach involved defining the behavior to searched for. This was originally given as input-output pairs [28], and then generalized to allow slightly more flexible specifications [7,15]. More recent work in this area includes PARSEWeb that does static analysis on code fragments found by a text-based code search engine and then looks at input-output types [34]. These approaches generally are aimed at function level reuse. Other techniques such as program patterns [27] and keyword programming [20] are designed to work at the level of a code fragment.

These early techniques did not really succeed because either they attempted to do too little or because they attempted to do too much. Signature or type matching does not really find items of interest, although PARSEWeb shows that in combination with textual search it can be somewhat effective. Full semantic matching requires the user to specify too

*ICSE'09*, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

much and is quite difficult to accomplish, with the general problem being unsolvable and even approximations typically requiring the use of a theorem prover or similar technology. Our approach uses test cases and other programmer-friendly partial specifications that are generally easy to check and easier to provide.

A recent test-driven approach is CodeGenie [19]. This system lets the user define test cases as part of the development process in Eclipse and then uses the method names and signatures from the test case to build a query. It uses an internal search engine that understands program structure to find code to test and then presents the result to the user. Where this approach concentrates on using the test cases to generate a complex structured query that will pinpoint exact code that should work, our approach uses keywords to find candidates, uses transformations to expand the potential pool of solutions, and then uses semantics to restrict that pool. Test cases and semantics have also been used in a similar fashion for finding web services [11,29].

Current code search tools are based on information retrieval techniques. Early work here demonstrated that keywords from comments and variable names were often sufficient for finding reusable routines [12,22]. Later work here did query refinement either directly [33], by looking at what the programmer was doing [39,40], using an appropriate ontology [38], using learning techniques [10], using natural language [8], or using collaborative feedback [35]. Recent approaches, such as Assieme [17], Sorcerer [1], and Codifier [4], incorporate program structure and semantics as a search basis. The utility of many of these techniques can be seen in their use in today's web-based search engines. However, these techniques do not yield exact or semantically correct matches, one of the primary problems associated with attempting to reuse software.

## 2. Our approach

Our approach is to take a set of candidate solutions, attempt to transform that set into a more appropriate set, check the resultant set against the user's specifications, and then output all solutions that meet these specifications. The actual logical architecture is shown in Figure 1.

The system starts and ends with the user interface. This is responsible for gathering the user's specifications, initiating the search, and when done, displaying the formatted results. The first step in the search is to use keywords that describe the item being searched for to obtain a set of initial solutions. A suite of program transformations is then applied recursively to these solutions to obtain a more complete set of candidate solutions. These solutions are then restricted by eliminating any that don't match static specifications such as the signature. The next step is to augment each solution with additional dependent code from the original file.

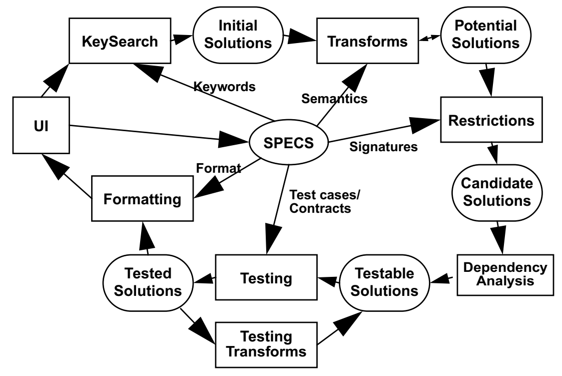


**FIGURE 1. Architecture for semantics-based code search**

The resultant set of test solutions are then checked against dynamic specifications such as test cases to see if they meet the user's criteria. Additional transformations can then be applied based on the result of testing. Finally, all solutions that meet the specifications are formatted using our prior work on adaptive formatting [30] and presented through the user interface.

Our central contributions are the use of both static and dynamic specifications to characterize the search and the use of transformations to modify and adapt the result of the initial search to meet the user's needs. In the next section we describe our specifications, how they are collected and how they are used. Section 5. then describes the various transformations and the strategies for keeping the set of potential solutions tractable. Section 6. describes the process of checking the specifications. The subsequent sections describes our experiences, results, and future work.

## 3. Specifying what to search for

The key to intelligent code search is to let users specify what they are searching for as precisely as possible. This means that they need to provide both the syntax and the semantics of their target. Moreover, these specifications must be easy for the user to provide; it shouldn't take longer to define what is needed than to write the actual code. Finally, the specifications should be incremental. The user should be able to easily augment or change the specifications based on the output that the original descriptions produced.

To meet these requirements we divided the specifications into separate sections and developed a web-based user interface to support them. We start with static specifications including a description of the target class or method and the target signature. We next include dynamic specifications through the use of test cases and, optionally, contracts.

The static specifications are given in two parts. The first is a natural language description of the class or method. Because users are accustomed to today's search engines, this is done through one or more sets of keywords. The second part of the static specification is the class or method signature.

A description and signature alone are not enough to specify the functionality of the target code. To do this precisely, one must provide a semantic definition. Such definitions take a variety of forms. Natural language semantics are too vague to be useful here. Mathematical languages such as Z [37] or Larch [14] are another alternative. These however are still difficult to write and more difficult to get correct, even for relatively simple processes. A more widely used approach is to use contracts as introduced in Eiffel [23]. Contracts are typically expressed in the form of preconditions and postconditions on methods or as conditions on a class. More recently, dynamic contracts have been used to specify the ordering and behavior of sets of methods of classes [3]. Contracts are relatively easy to specify, but they don't fully capture the semantics or the intent of the interface. Another solution can be found in the agile or extreme programming approach to development where test cases are developed first and the implementation is tested continually.

Of these various forms, test cases are generally the easiest for the user to provide. Moreover, they are nicely incremental in that users can look at the functions that pass the given test cases and, if they are not what they were looking for, they can add further test cases. Thus, the first part of our semantic specification is a suite of test cases. For methods, our approach lets these be given in the form of input-output pairs, code fragments, or external files. For the input-output pairs we support both types of object equivalence as well as expected exceptions. For classes, we support call sequences. Here each call invokes one of the methods in the signature, and the results of the call can be saved for future calls, checked or ignored.

In addition to test cases, we allow contract specifications. Here we use JML [5] as the definition language, letting the user define method preconditions and postconditions.

Contracts and test cases only define the functional aspects of semantics. Programmers typically care about many other dimensions of their code including privacy, security, threading behavior, complexity, and performance. Our goal is to handle these additional dimensions either directly as part of the specifications or indirectly as part of the selection and presentation process.

We currently support the specification of security constraints on the methods and classes that are being searched for. This is done using the Java security model [13]. The user can specify Java security conditions for files, sockets, AWT/Swing, system properties, and run time behaviors. Other nonfunctional semantics are supported by letting the user sort the resultant code fragments by code size, code complexity, or performance on the test cases.

Our initial front end is web-based, built using the Google Web Toolkit [9]; it can be seen in Figure 2. At the top of the window the user indicates whether they are looking for a class or a method, whether to look locally or using on the web, and if on the web, which external code search engine to use. The user also provides keywords much as they would for existing search engines.

Below this is a box where the user provides the actual semantics describing the target. Here the user first provides the signature. The signature is checked by the server and updated with full type names and parameter names if necessary. Below the signature, the user can enter test cases. The CALL type test cases shown here let the user set the parameters and the expected return value. The system supports abbreviated syntax for arrays, collections and maps, doesn't require quoted strings, and also lets the user enter appropriate code fragments. The code entered is semantically checked by the back end before the search.

Contracts and security constraints are entered by clicking on the appropriate button in the method box and filling in the resultant forms.

The results are displayed at the bottom of the window. Here the user can change the sort order and the formatting style. The result display includes the code source and the licensing or copyright information.

The interface is designed to be adaptable so that the user can easily add a new test case or change the keywords and see the modified results.

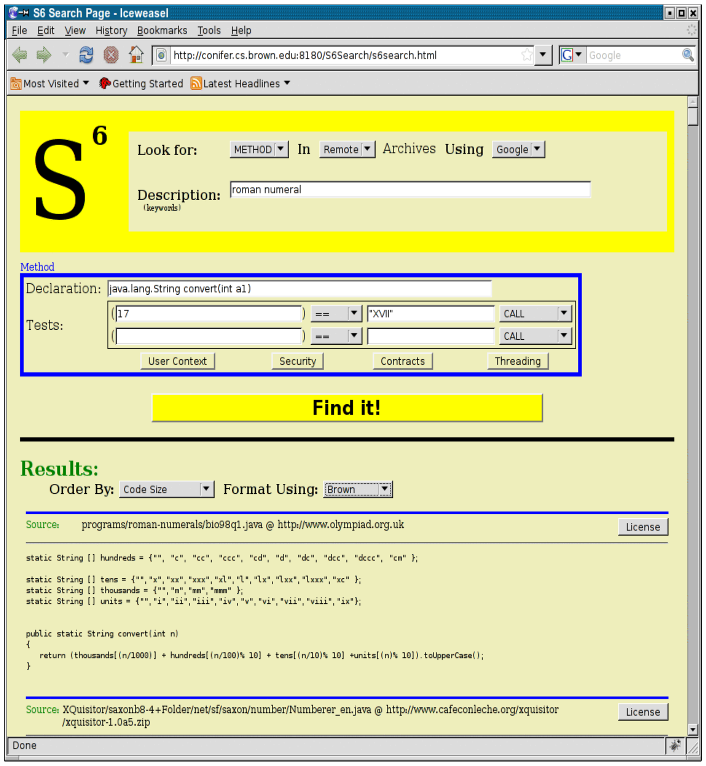
## 4. Finding the initial solutions

The first step in using this semantic information is to find an initial set of candidate solutions. A solution in our system consists of 1) the abstract syntax tree for the file the target is derived from; 2) the node in this syntax tree corresponding to the target method or class; 3) a set of auxiliary abstract syntax tree nodes needed to resolve dependencies; 4) a set of flags describing the solution state; and 5) a score.

We use Eclipse to build the basic abstract syntax trees. We have our own semantic analyzer operating over these trees and adding annotations representing type, method, and variable definitions and uses. This was needed because we needed to work outside of an Eclipse project, we needed an analyzer that was very forgiving since we are essentially compiling files without the proper compilation context, and we needed to maintain multiple representations of the same class.

While solutions are represented as abstract syntax trees, only the initial solutions are stored that way to save memory. Any derived solution is stored as a text delta from its predecessor, and the corresponding abstract syntax tree is generated on demand and cached while the solution is being processed.

The initial set of candidate solutions is obtained by using a code search engine to find a set of files that


S6 Search Page - Iceweasel
File Edit View History Bookmarks Tools Help
http://conifer.cs.brown.edu:8180/S6Search/s6search.html
Most Visited
Getting Started
Latest Headlines
S 6
Method
User Context
Security
Contracts
Threading
Find it!
License
Done

Look for: METHOD In Remote Archives Using Google

Description: (keywords) roman numeral

Declaration: java.lang.String convert(int a1)

Tests: (17) == "XVII" CALL

Results:

Order By: Code Size Format Using: Brown

Source: programs/roman-numerals/bio98q1.java @ http://www.olympiad.org.uk

```
static String [] hundreds = {"", "c", "cc", "ccc", "cd", "d", "dc", "dcc", "dccc", "cm" };

static String [] tens = {"","x","xx","xxx","xl","l","lx","lxx","lxxx","xc" };
static String [] thousands = {"","m","mm","mmm" };
static String [] units = {"","i","ii","iii","iv","v","vi","vii","viii","ix"};


public static String convert(int n)
{
   return (thousands[(n/1000)] + hundreds[(n/100)% 10] + tens[(n/10)% 10] +units[(n)% 10]).toUpperCase();
}
```

Source: XQuisitor/saxonb8-4+Folder/net/sf/saxon/number/Numberer_en.java @ http://www.cafeconleche.org/xquisitor /xquisitor-1.0a5.zip

FIGURE 2. Code search web-based user interface.

match the user-specified keywords. This is done for the Google, Krugle, and Koders web code search engines. using modules that simulate a query, interpret the results, and then obtain the actual source files for the first 100-200 reported matches. In addition, we have developed our own desktop code search engine, Labrador, based on Compass [2], for local search.

For each selected file, we extract a set of initial solutions. When searching for a method, this is the set of all methods in the file that have associated code. For a class, it is the set of all classes or inner classes in the file. We set the score of the solution to be the rank of the original file in the search results.

## 5. Transformations

The next step is to take this initial set of solutions and expand it to produce solutions that match the user's specifications. This is done using transformations.

A transformation is a mapping from a solution to a possibly empty set of additional solutions. It consists of a name, a set of conditions that determine when the transformation is applicable, and a set of mapping objects, each of which takes the abstract syntax tree of the original solution and produces an edited tree representing a new solution. The conditions are used to control the potential exponential blowup in the number of solutions to consider. Editing is accomplished using Eclipse's abstract syntax tree rewriting capabilities.

The set of transformations was chosen to let the system extract, convert, and otherwise manipulate the initial solutions into a form that meets the programmer's specifications. While most of the transformations preserve the original semantics, some will change the program's behavior. Because all solutions are eventually tested, this is not a problem. Moreover, these transformations have proven themselves useful.

### 5.1 Signature transforms

The set of transformations can be divided into different groups. The first group include the simple transformations that are needed to make the solution conform to the given signature. These include:

**NAME:** This transformation refactors the name of the candidate method to the name given by the input if the return type, parameters, and exceptions thrown for the method match the required signature.

**RETURN:** This transformation changes the return type of the candidate method to that of the given signature. It can only be applied if the parameter types are correct, and if the desired return type is either void or is compatible with the declared type.

**PARAMETER TYPES:** This transformation will replace the parameter types of the candidate method with types from the signature if they are compatible.

**PARAMETER ORDER:** This transformation will change the order of the parameters for the candidate method for all orders that match the signature types.

**EXCEPTION:** This transformation will change the exception types from those given in the original code to those given by the signature. It both changes the method declaration and any internal throw statements as needed. It is only applied if the method has the correct parameter types and return type and if there are no internal calls that would require an exception to be thrown.

**STATIC:** This transformation will convert the given method to a static method if that is what is required by the signature and if the method does not access any non-static fields or methods.

While the above transformations are described in terms of methods, they also work for solutions that represent classes. In the class case, each transformation is applied for each pairing of a method of the class and method signature for the class.

## 5.2 Generative transforms

The second set of transformations are designed to build new solutions from existing code. These are central to the generative techniques that are the focus of our search strategy. More often then not, the functionality that one programmer wants as a single function does not match exactly what another programmer has done. Code for tokenizing command line arguments can be hidden in a routine that both computes the tokenization and executes the result; a program that generates roman numerals might take extra parameters indicating whether to use upper or lower case and how to handle special cases such as four; an algorithm might be specialized for a specific class in the original application, but could have been made generic.

The transformations we have defined are able to handle these and related cases. As we uncover more variations that are needed, new generative transformations can be added. These transformations include:

**CHUNK:** This transformation attempts to find segments of the candidate method that might meet the user's specifications. The motivation here is that the desired functionality might be a subset of a method rather than the whole method.

This transform works in stages. It first identifies all the variables used in the method and their types. Next it determines for each top-level statement of the method which variables are set, which variables are created, and which variables are used. From this information it is able to identify the set of statements that set a value of return type. For each such statement, the transformation effectively does a backward slice, adding prior statements that are required one by one. After each statement is added, the set of open variables is checked. If this set only contains variables of the desired parameterized types, then a new method is created with the dependent statements and the target statement. When creating this method the transformation handles adding any required declarations and initialization, changing return statements appropriately, and setting up the proper method signature and body.

**EXTRA PARAMETERS:** This transformation looks for methods that have the proper return type, have parameters that include the target types, and have extra parameters. It replaces each of the extra parameters with an initial assignment statement.

The tricky part of this transformation is determining what value to assign to the replaced parameter. This is done based on the parameter type and on the contents of the method. For boolean parameters, both *true* and *false* are tried; for numeric types, both 1 and 0 are tried; for object types *null* is used. If an array or collection is being passed, then its size is used. In addition, the code for the method is scanned to look for conditionals and switch statements that involve the given value. Any values used in those comparisons are also tried as candidate replacements.

**GENERALIZE:** This transformation attempts to replace user-defined types throughout a class with the types used in the user's specifications. For example, if the actual code works on an internal class, but the specifications asked for an instance of *java.lang.Object* or *java.lang.String*, this transformation would attempt to change the method to use the system class. This is done by first checking if the actual class used in the code can be generalized by seeing if its methods and fields are actually used in the identified method.

## 5.3 Compilation transforms

The third set of transformations are used to make the code compile correctly in the test environment. These transformation can also be viewed as generative in that they attempt to adapt the original code to meet the new environment by removing unnecessary functionality or ensuring existing functionality is consistent with what the user wants. The current transformations here include:

**IMPLEMENTS:** This transformation removes unneeded implements clauses on the target class. This is required since the system will eliminate methods that are not part or required by the user's specification.

**REMOVE UNDEF:** This transformation removes any statements from the candidate method that access undefined fields, methods or classes. It will not remove return statements or the last statement in a method. This transformation has been useful in eliminating extra statements used for debugging, logging, and additional error checking from otherwise acceptable code.

**STATIC CLASS:** This transformation ensures that all target classes are static classes, even if they were derived from an internal non-static class.

**THROW:** This transformation replaces throw statements in the code that throw exceptions that are not part of the user's defined signature either with exceptions that the user has declared or with *java.lang.Error.*

## 5.4 Testing transforms

The final set of transformations are applied after testing and attempt to modify the method based on the test results. The one transformation here handles a variety of different cases:

**FIX RETURN:** This transformation looks at the test results and checks if there is a simple transformation that will convert the actual results into the desired ones. Currently it handles inverting boolean returns, changing the case of string results, and the addition of a constant value to integer results. Multiple test cases are required for the boolean and numeric transforms.

## 5.5 Transform application

The transformations are applied repeatedly in stages to the set of solutions until no new solutions are found. Each solution is only considered once during each stage. When it is considered, all the potential transformations for that stage are applied, and then the solution is checked to see if it should be retained or discarded. The number of solutions produced by each stage is kept under control by using the appropriate conditions on the transformations, eliminating duplicate solutions, and removing infeasible solutions after they have been candidates for transformation.

We currently support three stages. The initial stage handles transformations that involve basic setup such as STATIC CLASS. Solutions are eliminated here only if they directly involve non-static classes.The normal stage handles the other transformations except for FIX RETURN. Solutions that don't match the specified signature are eliminated at this point after they are considered by all available transformations. The final stage involves transformations based on test results such as FIX RETURN. Transformations that did not pass the test cases are eliminated here after they have been considered by this transformation.

Another approach used to keeping the number of solutions under control is to limit the number of active solutions at each phase. We limit the number of initial solutions (4000), and the number of solutions that remain active during the normal stage (1000), as well as the number of solutions that will actually be tested (250). Where there are more potential solutions, the score associated with the solution is used for elimination. In practice, these limits are rarely reached.

## 6. Testing

The result of applying all but the final stage of transformations is a set of solutions that have the signature specified by the user. The next stage in our system involves seeing if these potential solutions pass the test cases, contracts, and other semantic conditions given by the user.

In order to do testing however, we need to first ensure that the identified code will compile. This is done during a dependency analysis phase.

This phase serves multiple purposes. Its primary aim is to use information from the abstract syntax tree to identify transitively any helper methods, inner classes, and fields that are referenced by the target code. If the user is searching for a method, these are added to the solution. If the user is searching for a class, the system has identified a class that has at least the methods provided by the user. In this case, the job of dependency analysis is to remove from the class all declarations that are not required by the solution.

Another job of dependency analysis is to eliminate any solutions that will not compile because they contain undefined references. The final job of this analysis is to identify the set of system classes that are used by the code so that appropriate import statements can be generated for compilation and testing.

The actual testing is done by generating a Java source file containing the identified code, the identified helper code, imports clauses, and the test cases, and then using ant and junit.

Contract checking is handled during this testing process. First, the contract information associated with a method is inserted as JML annotations in the appropriate code. Second, *jmlc* is used to compile. Then the tests are run using the appropriate JML libraries.

Each tested solution is examined to see if it succeeded or if it failed due to compilation errors, testing errors, or just failed test cases. The FIX RETURN transformation is applied to any solution where all the test cases ran, but some failed. Any new solutions that this transformation then builds are tested in a new dependency and testing pass.

## 7. Experience

We have used our code search system for a variety of different examples some of which are shown in Table 1. While most of these cases are searching for a particular method, the last three, as noted by their signatures, are searching for a class.

The two tokenize examples define a function that takes a string and divides it into words. The second one handles both single and double quotes as would be done on a shell command line. This can be seen in the test cases shown in Table 2. The robots.txt example is looking for a function that checks *robots.txt* for a URL to see if that URL is crawlable.

One thing to note from these examples is the small number of test cases that are sufficient for searching. The use of keywords limits the search to files that might be relevant. The odds of a random function

| Name | Keywords | Signature |
|---|---|---|
| Simple Tokenizer | tokenize quote | List<String> tokenize(String) |
| Quote Tokenizer | command tokenize split argument quote list | List tokenize(String) |
| Robots.txt | robots.txt | boolean check(URL) |
| Log2 | log base | int log2(int) |
| To Roman | roman numeral | String toRoman(int) |
| From Roman | roman numeral number conversion | int convert(String) |
| Primes | prime number | boolean checkPrime(int) |
| Perfect Numbers | "perfect number" | boolean isPerfect(int) |
| Day of Week | day of the week | String dayOfWeek(String date) |
| Easter | Easter date holiday year | Date computeEaster(int); |
| Multimap | multiset, multimap | class MultiMap { Multimap(); void add(Object); int count(Object); } |
| Union-Find | union find | class UnionFind { UnionFind(); void add(Object); Object find(Object); void union(Object,Object); } |
| Text Delta | text delta | class TextDelta { TextDelta(String newstr,String oldstr); String apply(String old) } |

**TABLE 1. Examples for semantic searching**

returning the right results in these cases is quite small. So far we have found that about one solution in ten is not what we were searching for. For example, a function that returns the number of low order zeros rather than the log or a function that only converts to roman numerals for numbers up to fifty. However, such solutions are easy to spot and could be eliminated with one additional test case.

The class examples are a bit more complex. The first represents a set that counts the instances of its elements. The second represents an implementation of the union-find set algorithm. For the final one, the class is supposed to contain a compact representation of the difference between its two input strings and then regenerate the output string given this representation and the original input. The test cases for these are given as a sequence of calls as shown in Table 3.

A summary of our results for these examples is shown in Table 4. The search engines used in the tests are arbitrary. The third column gives the run time in seconds using only one thread to do all the processing. The fourth column shows the run time using 8 threads on a 8-core machine and illustrates that significant speed up can be obtained by parallelizing the search process. The fifth column indicates the number of files found by the given search engines. The sixth column indicates the number of initial solutions that were found in these files. The seventh column indicates the total number of solutions considered. This reflects all the new solutions that were generated by the various transformations. The eighth column indicates the number of solutions that had the proper signature and could be tested. The final column indicates the number of solutions found. In several cases, the system found several variations of the same solution, each slightly different due to different transformations. A summary of the transformations that were used in the generated results for these examples is shown in Table 5.

Our experiments show several things. First, we have found that the system works in the sense that if one of the source files located based on the keyword search contains code that can be transformed into the appropriate form using the existing transformations, then that code will be found.

This highly qualified statement points out some of the characteristics of our approach. First, the system is highly dependent on being able to find the source files that contain appropriate code. Keywords provide a starting point, but we still found instances where our vocabulary and the vocabulary of the original author differed so that we did not find any appropriate code. For those cases where we obtained no results, we went through the first several pages of standard code search

| Name | Test Cases | |
|---|---|---|
| | Input | Output |
| Simple Tokenizer | “this is a test” | [ “this”, “is”, “a”, “test” ] |
| Quote Tokenizer | “this is a test” | [ “this”, “is”, “a”, “test” ] |
| | “this is a ‘test with’ quoted \”string types\” in it” | [ “this”, “is”, “a”, “test with”, “quoted”, “string types”, “in”, it” ] |
| Robots.txt | “http://www.cs.brown.edu/people/spr” | true |
| | “http://www.cnn.com/topics” | true |
| | “http://www.nytimes.com/college/students” | false |
| Log2 | 0 | RuntimeException |
| | 1 | 0 |
| | 4 | 2 |
| | 32 | 5 |
| To Roman | 13 | xiii |
| From Roman | VIII | 8 |
| | xxvi | 26 |
| Primes | 5 | true |
| | 39 | false |
| | 59 | true |
| Perfect Numbers | 6 | true |
| | 12 | false |
| | 28 | true |
| Day of Week | “08/07/08” | “Thursday” |
| Easter | 2008 | new Date(108,2,23) |

**TABLE 2. Test cases for method examples**

| MultiMap | Union-Find | Text Delta |
|---|---|---|
| MultiMap rslt = new MultiMap();<br>rslt.add(“Hello”);<br>rslt.add(“Tata”);<br>rslt.add(“Hello”);<br>rslt.add(“Goodbye”);<br>rslt.add(“Tata”);<br>rslt.add(“Tata:);<br>rslt.count(“Unknown”) == 0<br>rslt.count(“Hello”) == 2<br>rslt.count(“Tata”) == 3<br>rslt.count(“Goodbye”) == 1 | UnionFind set = new UnionFind();<br>set.add(“abc”);<br>set.add(“def”);<br>set.add(“ghi”);<br>set.find(“abc”) == “abc”<br>set.union(“abc”,”def”);<br>set.find(“def”) == “abc” | TextDelta t = new<br>TextDelta(“abcdef”, “abxef”);<br>t.apply(“abxef”) == “abcdef” |

**TABLE 3. Test cases for class search examples**

output and verified that there were no functions there that even came close to meeting our criteria.

Second, the set of transformations is powerful but incomplete. The system will become more effective as we add more transformations. One particular problem arises with handling generics since about half of the currently available open-source code assumes Java 5 and uses generics and the other half only assumes Java 1.4 and does not. Additional transformations are relatively easy to encode.

Third, the number of total solutions considered tends to be within a small multiple of the number of initial solutions, often within two. This shows that the conditions on the transformations and the continual pruning we use are effective. Currently, the most significant increases occur during searching for a class where each of the methods needs to be transformed individually. We also found that the test cases we have looked at do an average of six levels of transformations, with the maximum being one of the class cases with thirteen.

| Example | Engines | 1-thread Time | 8-thread Time | # Source Files | #Initial Solutions | #Total Solutions | # Tests Run | #Results Found |
|---|---|---|---|---|---|---|---|---|
| Simple Tokenizer | Labrador | 72.725 | 28.956 | 138 | 3860 | 4159 | 35 | 14 |
| Quote Tokenizer | Koders, Labrador | 22.267 | 15.854 | 4 | 164 | 213 | 10 | 6 |
| Robots.txt | Krugle, Labrador | 82.889 | 27.267 | 124 | 145 | 883 | 20 | 1 |
| Log2 | Google | 165.414 | 45.750 | 100 | 1464 | 1771 | 101 | 1 |
| To Roman | Koders, Labrador | 107.526 | 32.664 | 56 | 888 | 1010 | 48 | 6 |
| From Roman | Google | 266.671 | 107.819 | 38 | 2730 | 3244 | 140 | 3 |
| Primes | Google, Labrador | 199.020 | 50.587 | 228 | 4000 | 5266 | 119 | 14 |
| Perfect Numbers | Google | 34.811 | 12.421 | 28 | 92 | 113 | 13 | 5 |
| Day of Week | Labrador | 175.939 | 52.445 | 89 | 1628 | 2143 | 144 | 0 |
| Easter | Koders | 11.888 | 10.658 | 6 | 72 | 75 | 1 | 1 |
| Multimap | Google, Labrador | 235.464 | 113.499 | 165 | 231 | 2664 | 31 | 2 |
| Union-Find | Labrador | 467.540 | 168.870 | 149 | 416 | 6248 | 11 | 1 |
| Text Delta | Google, Labrador | 96.363 | 37.749 | 249 | 443 | 1649 | 3 | 1 |

**TABLE 4. Search results summary**

| Name | NAME | RETURN | PARAM TYPES | PARAM ORDER | EXCEPTION | STATIC | CHUNK | EXTRA PARAM | GENERALIZE | IMPLEMENTS | REMOVE UNDEF | STATIC CLASS | THROWS | FIX RETURN |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Simple Tokenizer | X | X | | | X | X | X | | | | | | | |
| Quote Tokenizer | X | X | | | | | X | | | | | | | |
| Robots.txt | X | | | | | | | | | | | X | | |
| Log2 | | | | | | | | | | | | | | |
| To Roman | X | | X | | | | | X | | | X | | X | X |
| From Roman | X | | | | X | | X | | | | X | | | |
| Primes | X | | X | | | X | X | | | | | | | |
| Perfect Numbers | X | | | | | X | | | | | | | | |
| Day of Week | | | | | | | | | | | | | | |
| Easter | X | | | | | | | | | | | | | |
| Multimap | X | X | | | | | | | X | X | | X | | |
| Union-Find | X | | | | | | | | X | | | | | |
| Text Delta | | | | | | | | | | | | X | | |

**TABLE 5. Transformations used to generate results**

The time the system takes varies considerably. A significant portion of the time is dependent on needing to run the various test cases rather than statically eliminating unworkable solutions. Examples with very common signatures generally yield a much larger set of possible solutions that then have to be eliminated. Moreover, the testing process is the easiest to parallelize and, especially because it is mainly I/O bound, can achieve significant speed up. The overhead of using existing search engines varies from five to twenty seconds depending on the search engine used and the number of results that are retrieved. This is especially true for web-based engines where we need to make a separate web request not only to get the search results, but also for each resultant file. The overhead in generating and maintaining solutions as annotated abstract syntax trees is generally small, but is significant in those examples that involve large source files. Overall, we have not worked significantly on performance issues and expect that we can achieve meaningful speed ups with the current approach.

Finally, we have noted that while there is a lot of open source code available, the code that a programmer actually wants for an application is probably not available per se. We only found one instance of the Easter computation, for example. However, there are many programs out there that compute the date of Easter -- they just take their inputs and outputs in significantly different forms than what was asked for. For the Text Delta example, code could be found that computed edit differences, but nothing that took a constructor and built and object that could then be latter used (the one instance found was the code we wrote when we failed to find external code to do the task).

The system is currently available on the web at *http://conifer.cs.brown.edu/s6*.

## 8. Future work

We are working on several ways of extending this work. These include additional transforms, using context information, additional semantics, and an improved user interface.

By looking at different examples and examining what programmers actually do when converting open source code into code for their project, we can identify other transformations that will be helpful and that can be incorporated into the system. Some of these will deal with advanced type conversions, for example converting between arrays and collections or handling generics. Others will deal with removing unwanted dependencies, especially dependencies that cannot be resolved. A third set will deal with using classes to represent methods. A fourth set will deal with different solution approaches, for example returning a list or providing an iterator.

One of the drawbacks of our approach is that the function being searched for has to compile independently. Often, when programmers search, they are looking for code that will fit into an existing package, using existing classes and interfaces. CodeGenie addresses this by working inside Eclipse [19]. We will extend our approach so that the user can specify an appropriate context in which the new code should work and then use this context as part of the search process, using either Eclipse or a web-based interface. Transformations will be used to have the identified code use classes and methods from the given context, handling mappings between user classes and classes in the identified code as in [36].

Context information also flows the other way. The files that we find in the keyword search are generally part of a large package. We should be able to include dependencies from these additional files in constructing a viable solution.

Our notion of semantic search currently only handles a limited set of semantics. We want to enrich this to further improve what the user can specify and provide additional confidence in the results. Here we are looking into doing formal JML-based checking for valid solutions; using the initial JML specifications to generate test cases [6]; letting the user specify security conditions along the lines of Java security policies; letting the user specify threading conditions, specifying whether the code should be thread-safe and what should be externally synchronized; and letting the user specify data flow constraints.

Finally, we are continuing to work on the user interface, attempting to make it as easy as possible for the user to specify what they are looking for. Here we are concentrating on making it easier for the user to provide more complex test cases; making the interface more interactive so the user can easily see the effects of additional test cases or of modifying the keywords; augmenting search keywords with information from the test cases; and letting the user save test cases.

These continuing efforts are based on the success of the approach even in its current form. Our work demonstrates that semantics-based search is a more promising approach than traditional keyword based search. Our experience shows that only a small number of easily defined test cases are needed to properly identify appropriate code. Our system demonstrates that a transformation-based approach is feasible and can provide results that the programmer can immediately use. Finally, our approach is practical.

## 9. Acknowledgements

This work is supported by the National Science Foundation through grant CCR0613162. J. Travis Webb worked on the Labrador search engine; Christina Salvatore worked on the code formatter.

## 10. References

1. Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, “Sourcerer: a search engine for open source code supporting structure-based search,” *Proc. OOPSLA 2006*, pp. 682-682 (October 2006).

2. Shay Banon and Alan Hardy, “Compass Reference Documentation,” *http://www.compass-project.org/docs/1.2.2/reference/pdf/compass- reference.pdf*, (2008).

3. Mike Barnett and Wolfram Schulte, “The ABCs of specification: AsmL, behavior, and components,” *Informatica* Vol. **25**(4)(November 2001).

4. Andrew Begel, “Codifier: a programmer-centric search user interface,” *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).

5. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik

Poll, "An overview of JML tools and applications," *Intl. Journal on Software Tools for Technology Transfer* Vol. **7**(3) pp. 212-232 (June 2005).

6. Yoonsik Cheon and Carlos E. Rubio-Medrano, "Random test data generation for Java classes annotated with JML specifications," *SERP* Vol. **11** pp. 385-392 (June 2007).

7. Shih-Chien Chou, Jen-Yen Chen, and Chyan-Goei Chung, "A behavior-based classification and retrieval technique for object-oriented specification reuse," *Software Practice and Experience* Vol. **26**(7) pp. 815-832 (July 1996).

8. Shih-Chien Chou and Yuan-Chien Chen, "Retrieving reusable components with variation points from software product lines," *Information Processing Letters* Vol. **99** pp. 106-110 (2006).

9. Google Corporation, "Google Web Toolkit Documentation," *http://code.google.com/webtoolkit/overview.html*, (2008).

10. Christopher G. Drummond, Dan Ionescu, and Robert C. Holte, "A learning agent that assists the browsing of software libraries," *IEEE Trans. on Software Engineering* Vol. **26**(12) pp. 1179-1196 (December 2000).

11. Michael D. Ernst, Raimondas Lencevisius, and Jeff H. Perkins, "Detection of web service substitutability and composability," *WS-MaTe 2006: International Workshop on Web Services -- Modeling and Testing*, pp. 123-135 (June 2006).

12. William B. Frakes and Thomas P. Pole, "An empiracal study of representation methods for reusable software components," *IEEE Trans. on Software Engineering* Vol. **20**(8) pp. 617-630 (August 1994).

13. Li Gong, "Java 2 platform security architecture," *Sun Microsystems* (*http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security- spec.doc.html*), (2002).

14. J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software* Vol. **2**(5) pp. 24-36 (March 1985).

15. Robert J. Hall, "Generalized behavior-based retrieval," *Proc ICSE'93*, pp. 371-380 (May 1993).

16. David Hemer and Peter Lindsay, "Supporting component-based reuse in CARE," *Australian Computer Science Communications* Vol. **24**(1) pp. 95-104 (2002).

17. Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proc. UIST 2007*, pp. 13-22 (October 2007).

18. Jun-Jang Jeng and Betty H. C. Cheng, "Specification matching for software reuse: a foundation," *Proc. ACM Symp. on Software Reuse*, pp. 97-105 (April 1995).

19. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Ricardo Morla, Paulo Masiero, Pierre Baldi, and Cristina Lopes, "CodeGenie: using test-cases to search and reuse source code," *ASE '07*, pp. 525-526 (November 2007).

20. Greg Little and Robert C. Miller, "Keyword programming in Java," *Proc. ASE 2007*, pp. 84-93 (November 2007).

21. Daniel Lucredio, Antonio Franciso do Prado, and Eduardo Santana de Almeida, "A survey of software components search and retrieval," *Proc. EUROMICRO'04*, pp. 152-159 (2004).

22. Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Trans. on Software Engineering* Vol. **17**(8) pp. 800-813 (August 1991).

23. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall (1988).

24. A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering* Vol. **5** pp. 349-414 (1998).

25. Hafdeh Mili, Fatma Mili, and Ali Mili, "Reusing software: issues and reseach directions," *IEEE Trans. on Software Engineering* Vol. **21**(6) pp. 528-562 (June 1995).

26. Rym Mili, Ali Mili, and Roland T. Mittermeir, "Storing and retrieving software components: a refinement based system," *IEEE Trans. on Software Engineering* Vol. **23**(7)(July 1997).

27. Santanu Paul and Atul Prakash, "A framework for source code search using program patterns," *IEEE Trans. on Software Engineering* Vol. **20**(6) pp. 463-475 (June 1994).

28. Andy Podgurski and Lynn Pierce, "Retrieving reusable software by sampling behavior," *ACM Trans. on Software Engineering and Methodology* Vol. **2**(3) pp. 286-303 (July 1993).

29. Steven P. Reiss, "A component model for Internet-scale applications," *Proc. ASE 2005*, pp. 34-43 (November 2005).

30. Steven P. Reiss, "Automatic code stylizing," *Proc. ASE '07*, pp. 74-83 (November 2007).

31. Eugene J. Rollins and Jeannette M. Wing, "Specifications as search keys for software libraries," *Proc. 8th Intl. Conf. on Logic Programming*, pp. 173-187 (1991).

32. Colin Runciman and Ian Toyn, "Retrieving re-usable software components by polymorphic type," *Proc. 4th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pp. 166-173 (1989).

33. Vijayan Sugumaran and Veda C. Storey, "A semantic-based approach to component retrieval," *Advances in Information Systems* Vol. **34**(3) pp. 8-24 (2003).

34. Suresh Thummalapenta and Tao Xie, "PARSEWeb: a programmer assistant for reusing open source code on the web," *Proc. ASE'07*, pp. 204-213 (November 2007).

35. Taciana A. Vanderlei, Frederico A. Durao, Alexandre C. Martins, Vinicius C. Garcia, Eduardo S. Almeida, and Silvio R. de L. Meira, "A cooperative classification mechanism for search and retrieval software components," *Proc SAC'07*, pp. 866-871 (March 2007).

36. Yiqiao Wang and Eleni Stroulia, "Semantic structure matching for assessing web-service similarity," *Service-Oriented Computing: ICSOC 2003*, pp. 194-207 (2003).

37. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).

38. Haining Yao and Letha Etzkorn, "Towards a semantic-based approach for software reusable component classification and retrieval," *ACMSE'04*, pp. 110-115 (April 2004).

39. Yunwen Ye and Gerhard Fischer, "Supporting reuse by delivering task- relevant and personalized information," *Proc. ICSE'02*, pp. 513-523 (May 2002).

40. Yunwen Ye, "Programming with an intelligent agent," *IEEE Intelligent Systems* Vol. **18**(3) pp. 43-47 (May 2003).

41. Amy Moormann Zaremski and Jeannette M. Wing, "Signature matching: a key to reuse," *Software Engineering Notes* Vol. **18**(5) pp. 182-190 (December 1993).