Creating Test Cases Using Code Search

Steven P. Reiss

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

***Abstract*—Testing and test cases are critical to maintaining modern, evolving systems. Yet generating a good set of relevant test cases that covers what is important remains a difficult task. We propose a new approach to generating test cases through the use of code search. Open source repositories have made an enormous amount of code available. This code contains unit tests for a wide variety of programs where the programmer has often given substantial thought to what should be tested, what is important to test, and how to test it. Our approach is to extract the relevant tests from on-line repositories and adapt these for testing user code. We evaluate the approach against existing user-generated tests in the repository.**

Keywords—Code search, test case generation, black box testing.

I Introduction

Testing is an essential part of software maintenance. Yet writing good test cases remains difficult and error-prone. The goal of this research is to simplify that process by building on the extensive libraries of test cases that have been developed for open source software.

Searchable code repositories have grown exponentially over the last few years. OpenHub (formerly Ohloh) claims to have indexed about 31 billion lines of source code and GitHub is larger. The code that is stored in these repositories includes the test cases. In fact, searching for "junit" and "Test" in Java files shows that GitHub has about four million Java test files and OpenHub another million. With this many tests, and more to come, it is becoming increasingly likely that a test case for existing or new code either already exists in the repository or there is code in the repository that can be adapted to form a test case.

The test cases in the repository are typically generated by programmers and often are designed to cover realistic and important cases. Combining test cases from multiple programmers can provide a better and more extensive test suite. Moreover, taking test cases from the repositories is a classic example of code reuse.

The contribution of our work is that it introduces a set of techniques for extracting and using test cases from searchable code repositories that are both practical and useful and demonstrates these techniques in a tool called TGEN that uses our existing S^6 semantic code search framework.

This work is supported by the National Science Foundation grant CCF1130822.

II Related Work

Because test case generation is both difficult and tedious for programmers, a wide variety of tools have been developed to automate the process. These tend to fall into two categories, white box testing and black box testing.

White-box test case generation uses the code to be tested as a basis. Automated techniques typically look at each function in a class and attempt to find inputs to that function to achieve a desired level of coverage, for example path coverage or statement coverage. The techniques often use symbolic execution along with heuristics and a framework such as genetic programming, to explore the input spaces to find appropriate values. Examples of such systems include PET [2] and KLEE [6]. While automated white-box test generation is useful, it can be time consuming and the test cases it generates are incomplete in that they can not specify what the desired output or effect of the code should be.

Automatically generated black-box tests consider the specification rather than the code as the basis for generating test cases. They assume that some model of the code has been defined and generate test cases based on that model. Models can come from specification languages such as JML [7,43] or UML sequence diagrams [44]. This however, assumes that programmers have defined specifications or models of their code which is often not the case. Black box test cases have also been generated for special cases, for example by monitoring program execution [31].

One problem with automatically generated tests is that the code being tested can require significant and complex data structures and set up. MSeqGen mines the source repository of the code being tested to find potential setup and calling sequences for such tests generated using random testing or dynamic symbolic execution [52].

There has been significant work done on code search. Early work in this area demonstrated that keywords from comments and variable names were often sufficient for finding reusable routines [14,30]. Later work did query refinement either directly [48], by looking at what the programmer was doing [58,59], using class signatures [22], using an appropriate ontology [57], using the surrounding context [20,56], using learning techniques [10], using natural language [9], using an execution trace [29], using topic graphs [55], using associations [49], using typestates

[36], or using collaborative feedback [53]. Recent approaches, such as Assieme [21], Sorcerer [3,4], Codifier [5], Exemplar [16] and Portfolio [35] expand basic keyword search to consider program structure and semantics. Other recent work has looked at more sophisticated IR techniques [50] and on automatic query reformulation [17,18,45]. More sophisticated search techniques use theorem proving techniques [46,47].

Another semantic approach involved defining the behavior to searched for. This was originally given as input-output pairs [38], and then generalized to allow slightly more flexible specifications [8,19]. 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 [51]. Other techniques such as program patterns [37,54] and keyword programming [28] are designed to work at the level of a code fragment.

Several search-based systems use test cases as input. CodeGenie [25,26] lets the user define tests as part of the development process in Eclipse and then uses the method names and signatures from the test 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. Other recent code search work on test cases includes [1,23,27] and our S^6. Test cases and semantics have also been used in a similar fashion for finding web services [12,39], but have the problem that the user must know exactly what is being searched for [24].

This work has been driven in recent years by the growth in available, open-source repositories such as GitHub and SourceForge. While a variety of techniques for doing the search have been proposed, most of the search engines available today are keyword-based and return files.

Our prior work in code search includes the S^6 semantic code search tool. S^6 concentrates on producing results that are of immediate value to the programmer [40,41]. It takes as input a set of keywords along with a method (or class) signature and a set of test cases. It uses existing search engines to find candidate solutions based on the keywords. Then it undertakes a series of program transformations that attempt to map these solutions to other solutions that match the given signature, that can compile, and that might pass the test cases. Next, it runs the solutions that do compile on the given test cases. The solutions that compile, run, and pass the test cases are then returned to the programmer. We have also developed an extension of S^6 that uses code search techniques to find implementations of user interfaces based on a user-provided sketch of the interface along with a set of keywords [42].

One of the problems other researchers have noted with code search is that a large fraction of the results are often test cases for the code rather than the code itself. This was recognized, for example, by the Sourcerer project where they added heuristics to explicitly remove such results [3,4]. Rather than discarding these results, our research attempts to make use of them, effectively extracting relevant test cases from open source repositories.

III Methodology

Our approach, embodied in the TGEN tool, is a major extension of S^6 that generates JUnit test cases for Java Programs using code search. TGEN starts with an input specification that describes the class and methods to be tested along with a set of keywords describing that class. It then uses external search engines (GitHub and OpenHub) to find candidate testing files. For each returned file, it finds the test cases, and attempts to transform each to call the target code, This yields a set of candidate solutions.

For each candidate solution TGEN next applies additional transformations to restrict the solution to just the appropriate testing code and to ensure that the solution compiles. For each non-trivial transformed result, it runs the solution as a JUnit test, and marks it as acceptable if a reasonable number of the tests actually pass. Finally, the system merges all the acceptable solutions into a single test file for the user's original code. The result can then be vetted, validated, and edited by the programmer.

The new elements added to S^6 are a) mapping tests retrieved from the repository so that they call user code, b) restricting the solutions to only include the appropriate test cases, c) determining whether the code is actually a test case for the user's code, and d) merging all the solutions.

A Input Specification

In order to locate and validate test cases from the various open source repositories, our tool requires

- The class and methods to be tested.
- The name and package for the resultant test class.
- Keywords to extract code from the repositories.
- Access to the user's code in a runnable format.

TGEN includes a simple user interface, shown in the left of Fig. 1 to provide this information. The top part of the interface lets the user define the appropriate class path for the code being tested. Once the path is defined, the system constructs a list of potential testable classes and lets the user choose the class to test. Once the class is chosen, the system determines the set of testable methods and lets the user select which of these should be tested. Finally, the user needs to provide keywords for the search. Once the user hits the "Find Test Cases" button, the system creates a jar file containing the contents of the class path and an input file for our modified S^6. The input file is sent to the S^6 server through a web interface. Once the search is done, the interface displays the resultant code as shown in the figure.

The user interface generates an XML specification to provide this information to TGEN. Such a specification could also be generated directly by a programming environment or other tool that wanted to generate test cases. An example specification for testing a routine that converts an integer into a roman numeral is shown in Fig. 2. The *SIGNATURE* part of the specification describes both the result class (*S6TestRoman*) and the class and method(s) to test (*Roman.convertToRoman* in this case). The *KEYWORDS* fields provide the user-specified keywords that will be used

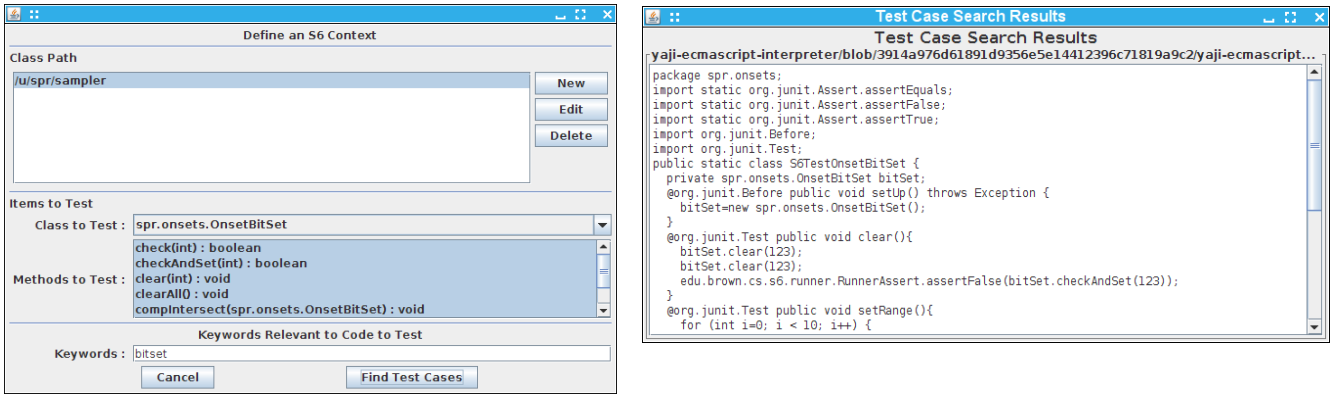

Fig. 1. User Interface for Test Case Generation. The input is specified in the window on the left. The result of the search is displayed in a separate window shown on the right.

```
<SEARCH WHAT='TESTCASES' FORMAT='NONE' LOCAL='FALSE'
                REMOTE='TRUE' OPENHUB='TRUE' GITHUB='TRUE'
                >
<SIGNATURE>
  <TESTING PACKAGE='spr.roman' NAME='S6TestRoman'>
    <TESTEE PACKAGE='spr.roman'>
      <CLASS NAME='Roman'>
        <METHOD NAME='convertToRoman' SIGNATURE='(I)Ljava/
                lang/String;' STATIC='TRUE' />
      </CLASS>
    </TESTEE>
  </TESTING>
</SIGNATURE>
<KEYWORDS>
  <KEYWORD>roman</KEYWORD>
  <KEYWORD>numeral</KEYWORD>
</KEYWORDS>
<CONTEXT FILE='/pro/s6/tmp/files/romantest.s6ctx' />
</SEARCH>
```

Fig. 2. Sample input file for test case searching

```
public class Test89 {
    @Test public void testToInteger(){
        assertEquals(8,RomanNumeral.toInteger("VIII"));
        assertEquals(9,RomanNumeral.toInteger("IX"));
    }
    @Test public void testToRoman(){
        assertEquals("VIII",RomanNumeral.toRoman(8));
        assertEquals("IX",RomanNumeral.toRoman(9));
    }
    @Test public void testIllegal() {
        try {
            RomanNumeral.toRoman(-1);
            fail();
        }
        catch (RomanException ex) { }
    }
}
```

Fig. 3. Example of code retrieved from the repository.

in the initial search. Finally, the *CONTEXT* element specifies a S^6 context file that is essentially a jar file of all the elements on the required class path that will be needed to compile and run the code to be tested.

B Getting Candidate Test Cases

Given such an input file, TGEN starts by searching both GitHub and OpenHub (as specified in the XML input) using the given keywords along with the keywords "org.junit" and "test". It looks at up to 200 returned files (100 files from each). Each file returned is considered as a potential solution and the combined result is the *initial solution set*. For example, in searching for the example from Fig. 2, it might get a result file such as that shown in Fig. 3.

The results returned from the repository cannot be used directly: the code being tested by the class is different from the code that the user wants to test; the code might be testing multiple methods in multiple classes; it might contain code that is not relevant to testing or to the particular test cases; it might be using an older version of JUnit (i.e. without *@Test* annotations); it might be dependent on other parts of the system or other libraries that are not available; it might do testing by human inspection (e.g. by printing rather than assertions); or it might not do any testing.

TGEN is implemented using the S^6 framework. It operates by applying code transformations to each solution in the solution to find new solutions that are then added to the solution set. Once all transformations are applied to a given solution, the solution is checked and discarded if it is not a valid potential candidate. This process is repeated until no new solutions can be added and all resultant solutions are potentially relevant. For test case generation, the check for relevance ensures there is at least one test case.

C Transforming Candidates

TGEN includes a set of specialized transformations to handle test case generation. The first attempts to integrate the returned candidate solution with the code the user wants to test, transforming the tests from the repository into code that calls the targeted user code. A sketch of the transformation algorithm is shown in Fig. 4.

This transform finds all calls in the solution that could potentially be calls to the code to be tested. It first identifies all calls in the test code to external methods. It then matches these calls to the methods the user wanted to test based on method compatibility. For each match, it creates a MapCandidate data structure that includes the method in the user's code being tested, the external method that was invoked by the call, and a mapping of class names from the testing code to classes in the code being tested. Comparability implies the parameter counts are the same, the various parameters are type compatible, and the method being invoked does have a void return type in a context that requires a value. Ideally, one would want to check consistency of the return type, but there is not enough information available to do so at this point. Currently, the only class

```
struct MapCandidate {
    to: The class/method in the code to test
    from: The class/method references in the testing code
    map: A mapping of classes from the testing code to the code to test
{

function TransformSetupTesting:
    foreach call C in the testing code invoking method CM'
        If class(CM) != the class of the testing code
            foreach method TM being test
                if the call Cis compatible with the method M then
                    Create a new MapCandidate with TM,CM and
                        class(CM) -> class(TM)

    Let same = all MapCandidates where method(from) = method(to)
    Remove all MapCandidates that invoke a method with a different
        name where there is a MapCandidate that maps one with the
        same name for the given class.

    Find all consistent sets of MapCandidates
    Create new solution for each such set


A call C is compatible with a method M if
    The arguments are compatible types or are in the class mapping
    The method doesn't return void in a context that requires a value

Two MapCandidates are consistent with one another if
    Their class mappings are consistent
    They do not map different methods to the same method to test
    They do not map different test methods from the same method
```

Fig. 4. Mapping calls in the testing code to calls in the code being tested

mapping we use is between the class of the method being called and the class of the code being tested. The underlying data structure allows for more complex mappings to handle cases where we might want to map parameter types to user types.

The transformation next tries to prune this set by checking for cases where the user's method and the call in the test code have the same name. Assuming that the original code being tested is doing something similar to the user code (which is required for the tests to be relevant), this will typically mean that the two functions have a similar purpose and thus that the matching with the same name is the most logical one. If such mapping occur, the transformation eliminates any mappings that involve the same classes but map a function to one with a different name where there is an existing mapping that involves the same name.

The transformation next finds all consistent sets MapCandidates using a simple recursive approach. Consistency here means that method being tested should only be mapped to by one call function, that a particular call function should only map to one method being tested, and that the class mappings of all the candidates are consistent.

If there are multiple functions being tested that have similar signatures and multiple potential candidates in the original solution, this set of mappings can be large. To limit this to a realistic subset, we ignore cases where the number of viable MapCandidates (after checking for equality) is too large (currently > 64), or where there are too many consistent mappings (currently > 512). While it would be possible to continue, the system has no way currently of ranking these large numbers of candidates. Rather than taking a long time to find the proper mapping, the system assumes that it will probably find other, simpler solutions that can be used to generate test cases.

```
public class Test89 {
    @Test public void testToInteger(){
        assertEquals(8,Roman.toInteger("VIII"));
        assertEquals(9,Roman.toInteger("IX"));
    }
    @Test public void testToRoman(){
        assertEquals("VIII",Roman.convertToRoman(8));
        assertEquals("IX",Roman.convertToRoman(9));
    }
    @Test public void testIllegal() {
        try {
            Roman.convertToRoman(-1);
            fail();
        }
        catch (RomanException ex) { }
    }
}
```

Fig. 5. Solution transformed to call user code. Differences from the original code are italicized.

For each resultant consistent mapping set, the transformation generates a new solution that replaces the original calls with the matched calls and replaces all instances of the original types with their mapped matching user types. The result is a candidate solution that invokes the user's code. For example, a transformation of the example in Fig. 3 can be seen in Fig. 5. Note that the result can make calls to methods that don't exist (e.g. *Roman.toInteger*) and is not guaranteed to compile. In particular, the return types of the original and matched methods are not checked and may be incompatible since the information to check this (i.e. a full compilation) is not available to the transform.

Other transformations handle JUnit issues. They take into account the different versions of JUnit, and map the code to use JUnit 4.x. This means adding *@Test*, *@Before*, *@BeforeClass*, *@After*, and *@AfterClass* annotations where appropriate, making all JUnit references be fully qualified names, replacing undefined exceptions in *@Test* clauses with the generic *java.lang.Exception*, removing *TestCase* as a superclass, replacing *assert* statements with JUnit *assert* calls, and generally normalizing the testing code. These transformations also ensure that the class is public and has an appropriate public, no-argument constructor as JUnit requires.

Standard S^6 transformations are then used to clean up and simplify the code. These remove code that has undefined elements and hence won't compile, change names to match the user's target class, fix import statements, take care of try-catch blocks and throws clauses that are no longer needed or that use unknown exceptions, add return statements that might be needed after code removal, fix code from older versions of Java (e.g. using *enum* as a variable), and remove unnecessary annotations.

Two other testing-specific transformation remove unneeded code. The first assumes that all testing code (now marked with appropriate annotations), along with the class constructor, are needed. It does a dependency analysis to find everything else that is relevant and then discards all code that is not needed. It also removes test cases that do no testing which might have been generated by the prior transformations. The second transformation starts with calls to the user's routines, does a separate dependency analysis, and removes all code that is not relevant. This removes test cases that do not invoke the user code.

```
public class S6TestRoman {
    @Test public void testToRoman(){
        assertEquals("VIII",Roman.convertToRoman(8));
        assertEquals("IX",Roman.convertToRoman(9));
    }
    @Test public void testIllegal() {
        try {
            Roman.convertToRoman(-1);
            fail();
        }
        catch (Exception ex) { }
    }
}
```

Fig. 6. Potential solution generated from the code. Non-compiling and irrelevant code has been eliminated.

```
public class S6TestRoman {
    public S6convertToRomanTest() { }
    @Test public void testToRoman(){
        edu.brown.cs.s6.runner.RunnerAssert.assertEquals("VIII",
            Roman.convertToRoman(8));
        edu.brown.cs.s6.runner.RunnerAssert.assertEquals("IX",
            Roman.convertToRoman(9));
    }
    @Test public void testIllegal() {
        try {
            Roman.convertToRoman(-1);
            edu.brown.cs.s6.runner.RunnerAssert.fail();
        }
        catch (Exception ex) { }
        edu.brown.cs.s6.runner.RunnerAssert.success();
    }
}
```

Fig. 7. Code that is ready to test..

The result of this process is a set of potential solutions that should compile, that invoke the code to be tested, and that include JUnit test cases. For example, the code from Fig. 5 might be transformed into that shown in Fig. 6 by changing the package, eliminating uncompilable code, and then removing unused tests.

D Checking Potential Solutions

The next step involves running each of these solutions to determine if it is actually a test case for the user's code. This can be difficult since TGEN does not know the semantics of the user's or the testing code. However, it can use the results of running the tests to get a heuristic approximation.

The system first applies a set of transformations to get the code ready to test. The replaces all the calls to the various JUnit assert routines (e.g. *assertEquals*), as well as to *fail*, with calls to an S^6 module that records test results without actually failing. A second test-preparation transformation handles tests that are passed by just returning, for example where the user checks the result explicitly, calls *fail* if there is a problem, and just returns if the test was okay. In order to know that a test was done and to distinguish it from code that doesn't do any testing, we added a new function, *success*, that can be invoked explicitly when a test would otherwise exit without calling one of the routines in the S^6 module that records test cases. This transformation adds calls to *success* in the appropriate contexts. A third transformation detects tests where the expected return is an exception and adds a call to *fail* at the end of such tests to record the fact that an exception was not thrown. For example, these transformations would map the code from Fig. 6 into that shown in Fig. 7.

Next we run the solution using JUnit along with the user's code (from the input context file), and record the results in terms of the number of assertions recorded, the number that passed, the number that failed, and the number of tests that aborted. In order to be relevant, the solution has to meet several criteria. First it has to invoke at least one test. If nothing is being tested, the solution can't be relevant. Second, at least one assertion has to succeed. If no assertion succeeds, then the test code is probably not testing the function the user has written, but is rather testing something else. Finally, the number of failing assertions should not be excessive compared to the number of passing ones. If there are too many failing assertions and compared to succeeding ones, then the test case is probably testing the wrong function.

Currently, TGEN discards solutions if more than 20% of the assertions fail. A test case that aborts with an unexpected exception counts as a failing assertion in this context. The 20% value was chosen heuristically to try to avoid accepting tests that aren't testing the user's code while at the same time including useful tests that just happen to fail on the user's code. The main difficulty occurs when there are multiple routines in the testing code and the system has identified some subset of these correctly, but has misidentified some others. In this case, the correct ones will generally pass, while the incorrect ones will fail. As long as the vast majority of the tests pass, we wanted to accept such solutions and then let the programmer determine if they are completely relevant, completely irrelevant, or, if partially relevant, what should be kept and what should be discarded.

E Generating a Final Solution

A final test-specific transformation is then done on all the solutions that run and pass the above criteria. This first looks at the individual solutions and eliminates duplicates. Duplicates can arise because tests might be in the open source repository more than once or just because two programmers used the same test. It also chooses a single solution for each original source file if there are multiple solutions. Multiple solutions can be generated because of different original mappings of calls to user code or because of different sequences of transformations. For each original source, it chooses what it considers the best solution. What is the best solution is open to interpretation. Generally, having more passing tests is better, but this has to be balanced by the number of failing tests. In the cases where the different solutions represent different mappings, this provides a way of comparing the mappings. Moreover, for two results with the same numbers, the simpler result is generally better. Based on our initial experiments, we develop a heuristic formula based on the number of tests passed P, the number of tests failed F, and the size of the code C (the number of AST nodes) using the formula $P \cdot 1024 - (F \cdot 512) - (C / 64)$ which reflects these concerns. This weights passing tests as twice as important as failing tests. The code complexity factor is then small enough to only be involved if the number of passing and failing tests yield the same scores.

This transformation next restores the original JUnit calls, removes the extra calls to *success* and *fail*, and merges all the individual test classes into a single test class using the name and package specified by the user. It handles name mapping to avoid conflicts from different solutions. It also

```
public class S6TestRoman {
    public S6convertToRomanTest() { }
    @Test public void testToRoman(){
        assertEquals("VIII",Roman.convertToRoman(8));
        assertEquals("IX",Roman.convertToRoman(9));
    }
    @Test public void testIllegal() {
        try {
            Roman.convertToRoman(-1);
            fail();
        }
        catch (Exception ex) { }
    }
    << Other tests >>
}
```

Fig. 8. Final test cases generated.

Table 1: Tests Generate for Textbook Examples

Test	# Tests Found	Time (s)
ArrayIterator	0	38
ArrayStack	20	43
Coin	2	46
Deck	38	134
DictionaryElement	4	12
Die	33	42
DifferentEquals	4	12
DisjointSetCluster	0	16
Employee	0	29
GeneralizedSelectionSort	0	11
Hanoi	0	10
Heap	0	11
Huffman	0	22
LetterCollection	0	15
LinkedQueue	33	92
MaxHeap	10	160
MedianQuick	4	41
Memory	0	32
MergeSort	0	14
NodePool	0	8
Pair	26	37
Pet	1	35
QuickSort	0	14
RabinKarpStringMatcher	0	12
SelectionSort	2	14
Stack	8	23
TwoTypePair	29	36

handles merging constructors as well as the @Before and @After methods from the different solutions into common routines. The resultant file is then returned to the user. For example, the final code from Fig. 7 can be seen in Fig. 8.

The final filtering of which test cases are actually relevant and which are not, is left to the user. The returned file can be run with JUnit in the user's environment and the user can then look at the tests that pass and fail. They can then discard any that are irrelevant or effectively duplicates. The result, after this, is a set of black box test cases for the user's code that other programmers, writing similar code, have designated as relevant or useful.

IV Initial Evaluation

Once the system was working, we used it to generate test cases for some easy example and for pieces of our own systems. Then we attempted to generate test cases for a set of suite of 27 textbook examples used in a study on fault localization [15] The examples of our own code where we used it ranged between 100 and 1800 lines.

A summary of results on the text book examples is shown in Table 1. We were able to generate test cases in about half of the examples. Moreover, the run times were generally under 40 seconds, especially where no test cases were generated.

This experiment illustrated several problems. The first is that it is difficult to generate test cases for code that was not designed for testing. For example, one of the examples was a program for the tower of Hanoi problem that simply printed the steps involved but had no testable return values. Others, such as NodePool had no functions that returned values; others such as Coin, returned random values.

A second problem that makes it difficult to test is when the code uses non-standard interfaces. This occurred in several of the text book examples. For example one of the text book examples created an array iterator but required being passed the size of the array while most similar code in the repository is just passed in the array; two others sorted part of an array, but required the index of the first and the last element as opposed to the more common first and one beyond the last or the first and a length.

A third problem arises when the code is very specific and not likely to appear in the repository. For example, the Huffman text book example creates a Huffman tree for a particular input string rather than a general one.

A fourth problem is that the whole process is quite sensitive to keyword selection. While our original work on S^6 showed that keyword sensitivity is a problem, the issue is acerbated when searching for test cases since programmers will typically put less work into documentation and using appropriate variable and method names when creating test cases, and the actual testing code may have little to do with what is being tested. However, code search is improving: Code Exchange at UCI, for example, can give very good results initially [32-34]. We expect that there results will find their way into repository search engines over time and that code search will improve significantly. Since our tool is built on top of existing search engines, we can easily piggy back on these improvements. Moreover, the time our tool takes is small enough so that a programmer can afford to run it multiple times with different keywords.

V Experimental Evaluation

Despite the problems identified above, our initial efforts demonstrated that there were a variety of situations in which a code search-driven approach to test generation seems to work. To better understand the quality of the resultant test cases and validate our approach, we performed an experiment to compare existing test cases in open source repositories with test cases that are generated by our tool.

We restricted ourselves to code in the repository that already has test cases so that the existing test cases give us a basis for comparing the quality of our generated tests with those that were saved in the repository. We also restricted ourselves to code that was easy to compile since we needed to run not only the original test cases, but also the new cases we generate and collect relevant coverage information.

A Methodology

We first obtained a set of programs that had known test cases using the OpenHub search engine. We started with a search using the keywords "test", "org.junit", and "assertE-

quals". This provided us with candidate test files that had actual tests in them. We looked at the first 1000 pages of results from this search. For each file that was returned, we checked that a) the file was in a named package (this made it easier to work with); b) the file included a @Test annotation (this excluded older versions of JUnit, but again made it easier to interpret and run the tests); c) the file referenced (i.e tested) exactly one external, non-library class; and d) there was at least one call to a JUnit assertion method. The restriction to one external class was meant to guarantee that the test was only testing one class in the system and thus we could run it standalone and could reasonably generate test cases for it without having to understand the whole system. It also let us understand coverage results since we could restrict ourselves to coverage of that class. It excluded a large number of relevant cases, however, where a logging class, a specialized test framework, or something simple like an external exception was used.

If all of these conditions were true, we then attempted to find the class that was tested by the candidate test file in the same package as the test in the repository. If we located this class, we then checked if it was worth testing by seeing a) if it was at least 60 lines long; b) if it contained at least one loop; and c) was not a duplicate of a previous accepted file. The first restriction eliminated trivial or empty classes that did provide something interesting to test; the second ensured the class wasn't just an object with getter and setter methods. Finally, if the file to test seemed worth testing, we checked if the two files would compile together without external resources. This made sure that the class being tested was also a standalone class, which was necessary since we did not have access to the whole project and its libraries from the repository and hence could not easily compile something more complex.

If the test code and code to test got this far, we next checked that tests were actually performed. To do this, we created a test directory for those files, and changed the package name on both to reflect that directory. Within the directory, we used *ant* to compile and run the tests. We checked the output to ensure that tests were actually run and that all tests passed. If no tests were run or if a test failed, we discarded the program. This eliminated tests that required external resource files that we did not have.

If the tests and code would compile and run successfully, we created an input file for S^6 test case generation for the code to be tested in the test directory. This file included the code to be tested as an S^6 context in binary form, the set of public methods in that code as methods that could be tested, and an initial set of keywords generated by using the class name, the method names, and the parameter names.

This process created a set of 52 programs that we used for our experiment. ,For each we could create test cases through code search using our tool, and compare the test cases that were thus created to the original test cases from the repository. We ran TGEN on each of the programs using only GitHub. The idea of using only GitHub was that our initial studies showed little overlap between GitHub and OpenHub and hence it was not that likely that the exact test file and test cases would exist in both. Note that TGEN will always attempt to look up to 200 source files, so in these cases it was looking at the top 200 results from GitHub as opposed to the top 100 from GitHub and 100 from OpenHub.

For each test program we generated an appropriate set of keywords. We first tried the keywords that were automatically generated, although we didn't put much effort into choosing these. If no test cases were generated or if the test results were trivial, we looked at the code that was being tested and replaced the keywords with our own set of keywords. If no test results were generated with the keywords we created, we tried a second time with different keywords. If no test were created the second time, then we stopped trying.

Finally, we ran the modified S^6 on each test with the derived keywords to generate a result test file. For each program, we created a working directory that included the code to test, the original test cases, the generated test cases, and appropriate *ant* build files. We then ran JUnit on both the original test and the generated test using *ant* and *jacoco* [11] to get coverage information for both cases. Finally, we created a CSV file with information from each test including the test results, coverage information, and the amount of time spent by S^6 generating the test cases. The results are summarized in Table 2 and Table 3.

B Summary of Tests

Table 2 describes the different test cases that were retrieved from OpenHub. It shows the class that was tested, the keywords used in the test, and with the number of methods, lines, branches and instructions that could be used for coverage in each case. Test names marked with an asterisk used the original keywords rather than manually selected keywords. While the table contains duplicate class names (*StringUtil* and *Tokenizer*), it should be noted that these are actually different classes with a common name.

The time column indicates the amount of time in seconds TGEN needed to generate the test cases. The times ranged from about four seconds to a little under three minutes. The average time was about 41 seconds for these tests. while the median is under 27. The cases that took longer are generally those where more tests were generated, with the system returning relatively quickly if it couldn't find anything. The timings were somewhat keyword dependent since they varied with the number of results returned from the repository and the relevance of those results.

C Summary of Results

Table 3 shows the test quality results of the experiment. The first column provides the name of the test. The next three columns describe the tests that were retrieved from OpenHub, providing the number of tests, the number that passed and the number that failed. Note that all the tests should pass based on our selection criteria.

The next three columns describe the tests generated. The first is the number of tests that were generated, then the number that succeed and the number that fail. The number of tests generated is generally reasonable, with most being in the range of 1 to 20. One, *WeightedQuickUnion*, repre-

Table 2: The Tests Used in Evaluating Test Case Generation

Class	Keywords	Time	#method	#line	#branch	#inst
ArrayTools	byte array indexOf	13.2	3	25	26	110
ArrayUtilities	reverse array	16.61	5	20	10	118
ArrayUtils	array concat	12.16	4	24	12	103
AtomicPositiveInteger	atomicpositiveinteger	8.50	22	73	50	365
Axis	orientation pixel tick	8.04	11	43	16	233
BaseConverter	base62	106.41	13	26	6	118
Board*	height board width	111.56	6	26	30	137
ByteUtilsArt	toByte toInt	34.72	8	12	4	151
CamelCaseConverter	camecase words	37.47	4	19	22	136
Capitalizer*	capitalize words	163.57	3	17	10	72
Chunker*	reader word next	27.89	3	43	32	179
ConsoleData	color row column	42.42	9	48	8	219
CRC16	crc16 checksum junit	15.81	9	26	8	134
CreatToken	token	7.06	9	229	261	931
DateNormalizer	date parse format	27.24	4	62	28	302
DeployerUtils	version filename junit	19.68	4	41	34	187
DescriptionHolder	escape html string	24.07	7	32	18	104
DuplicatePartSearcher	duplicate substring	18.47	9	48	24	215
EncodingStyle*	encoding style content	98.16	7	23	12	116
EquationUtil*	equation reference	25.97	5	28	24	132
ExpressionTokenizer	tokenize regex	43.18	8	41	16	190
FailInfo*	info value fail	30.60	7	15	4	106
FizzBuzz*	number	23.52	3	18	10	70
HangupCause*	cause code	27.67	8	73	6	708
HtmlSanitizer	html sanitize	24.77	2	82	10	546
HuffmanTree	weight input code	18.46	6	39	22	222
IntArray*	positive negative first	83.60	10	27	26	161
IntHashMap	inthashmap	51.10	26	152	74	662
IOUtil	relative path	25.53	3	17	8	68
IpAddress	ipaddress	91.48	10	28	8	169
MathUtil	'sum of digits'	21.57	13	42	36	222
Matrix	matrix translate multiply	53.32	9	55	4	644
NumberOfInversions	inversions array	65.2	5	38	22	218
NumSetBits	bits number offset	16.18	4	22	18	168
ObjectId*	hex machine increment	41.49	17	114	50	555
OGCServiceType	identifier identify lean	3.53	11	43	76	438
ParseUtil	quote split parse	22.72	2	30	24	137
PeriodFormat*	seconds period format	25.51	4	36	20	166
RUtil	escape 'R string' quoted	33.41	3	29	12	166
SipHash	sip hash code	13.95	5	80	4	814
SizzleCasts	stringToBoolean junit	54.46	13	45	30	258
Status	status code http	16.79	5	49	6	532
StringUtil	count lines	43.43	4	14	6	55
StringUtil	string indexof	13.83	4	24	24	125
StringUtilities	camel space	25.08	2	8	14	61
TheOnes	ones count number	22.3	5	18	8	86
Tokenizer	tokenizer next	105.34	57	181	198	1097
Tokenizer	tokenizer next	129.34	55	175	186	1066
UploadParameters	username password placeholder	84.90	10	22	8	68
WeightedQuickUnion*	union weighted connected	30.07	5	25	12	146
WhatTime	bracket tokenize	77.34	9	59	38	256
XmlUtils*	escape xml	30.79	3	23	16	77

sented a common program, and yielded 100 separate tests. While the latter might be excessive for human evaluation, most of the results are reasonable in size. The system could easily be changed to restrict the number of test cases generated as part of the final transformation.

Of the 52 examples, there were eleven cases where no test cases were generated. Analyzing these cases showed that the causes were pretty much as expected. For example *CreatToken* creates tokens that are defined in the class to test for a particular parser. It is unlikely that another class in the repository will create the same tokens. *FizzBuzz* does its work mainly by writing output and its return value is meaningless. *CRC16* seems to compute a nonstandard checksum value. *DuplicatePartSearcher* has a non-standard interface, requiring a constructor call, a set call, and then an evaluation call. *EncodingStyle* and *OGCServiceType* return internal enumeration values. *HuffmanTree* creates trees over integer arrays while most programs in the repository that create such trees do it over strings. For others, such as *Axis*, the keywords we used did not find any relevant tests.

For some of the other tests, a significant number of the generated tests failed. For example, in *DescriptionHolder* only 4 of the 9 tests passed. This seemed to be due to one of the test functions being mapped incorrectly. Most of the test cases did multiple tests. In the failing tests, some of the actual tests passed and one failed. Another case, *IntHashMap*, had 7 tests where, in addition to checking valid values, they tested for null and the actual code had a different behavior for this case. *MathUtil* was similar, but in this case the culprit was negative numbers.

The next four columns of the table provide coverage information for the original tests as reported by *jacoco*, first in terms of methods, then in terms of lines, branches, and instructions. These are given as percentages of the total number of methods, lines, branches, or instructions as noted in Table 2.

Table 3: Experimental Resuls

Class	Orig #test	Orig #succ	Orig #fail	Gen #test	Gen #succ	Gen #fail	Orig % method	Orig % line	Orig % branch	Orig % inst	Gen % method	Gen % line	Gen % branch	Gen % inst
ArrayTools	1	1	0	4	4	0	33	48	65	65	67	76	77	85
ArrayUtilities	3	3	0	13	11	2	80	95	80	97	80	95	80	97
ArrayUtils	1	1	0	11	10	1	25	29	0	26	75	92	83	95
AtomicPositiveInteger	9	9	0	7	7	0	50	55	48	56	41	45	40	44
Axis	1	1	0	0	0	0	9	37	31	42	0	0	0	0
BaseConverter	1	1	0	7	6	1	31	65	83	67	54	77	100	79
Board	7	7	0	13	12	1	83	81	100	88	100	96	93	97
ByteUtilsArt	1	1	0	7	6	1	13	8	0	11	25	50	50	27
CamelCaseConverter	7	7	0	21	18	3	50	89	91	95	100	100	95	100
Capitalizer	1	1	0	19	13	6	67	82	80	90	100	88	80	94
Chunker	9	9	0	20	20	0	100	79	81	77	100	79	81	77
ConsoleData	2	2	0	0	0	0	56	65	75	58	0	0	0	0
CRC16	1	1	0	0	0	0	67	77	100	75	0	0	0	0
CreatToken	1	1	0	0	0	0	89	41	38	58	0	0	0	0
DateNormalizer	1	1	0	0	0	0	50	45	0	40	0	0	0	0
DeployerUtils	1	1	0	2	2	0	50	54	18	55	50	12	3	17
DescriptionHolder	2	2	0	9	4	5	71	75	61	82	57	53	44	56
DuplicatePartSearcher	1	1	0	0	0	0	100	100	100	100	0	0	0	0
EncodingStyle	1	1	0	0	0	0	57	48	17	60	0	0	0	0
EquationUtil	6	6	0	9	7	2	80	93	83	96	80	96	92	98
ExpressionTokenizer	1	1	0	0	0	0	50	83	81	88	0	0	0	0
FailInfo	2	2	0	6	6	0	71	93	100	92	71	93	100	92
FizzBuzz	1	1	0	0	0	0	100	100	100	100	0	0	0	0
HangupCause	1	1	0	2	2	0	63	95	50	97	63	95	67	97
HtmlSanitizer	1	1	0	2	1	1	100	95	40	96	100	99	80	100
HuffmanTree	1	1	0	0	0	0	100	100	100	100	0	0	0	0
IntArray	6	6	0	6	6	0	80	67	65	64	80	67	65	64
IntHashMap	1	1	0	25	18	7	19	28	16	27	46	55	51	54
IOUtil	1	1	0	5	4	1	33	59	50	50	67	76	75	76
IpAddress	3	3	0	2	2	0	50	64	75	74	30	50	63	34
MathUtil	3	3	0	13	10	3	31	48	25	32	23	24	8	15
Matrix	1	1	0	9	4	5	44	55	25	61	89	98	100	99
NumberOfInversions	1	1	0	2	2	0	100	100	82	99	100	100	95	100
NumSetBits	4	4	0	4	4	0	75	91	72	88	75	91	72	88
ObjectId	5	5	0	4	4	0	82	78	76	81	71	64	76	58
OGCServiceType	2	2	0	0	0	0	45	65	11	61	0	0	0	0
ParseUtil	1	1	0	3	3	0	50	90	79	88	50	90	88	88
PeriodFormat	2	2	0	3	2	1	75	97	100	98	75	97	100	98
RUtil	1	1	0	12	11	1	67	90	75	83	67	83	58	73
SipHash	1	1	0	1	1	0	60	95	100	98	60	95	100	98
SizzleCasts	1	1	0	1	1	0	15	9	0	8	8	2	0	1
Status	1	1	0	4	3	1	80	98	67	99	80	98	83	99
StringUtil	1	1	0	5	5	0	25	43	33	35	75	71	67	60
StringUtil	2	2	0	1	1	0	75	96	100	98	50	67	46	58
StringUtilities	1	1	0	5	4	1	50	88	71	93	50	88	79	93
TheOnes	2	2	0	2	1	1	100	100	100	100	60	44	38	29
Tokenizer	6	6	0	6	5	1	61	48	30	53	61	49	33	55
Tokenizer	6	6	0	22	19	3	58	46	27	52	69	62	41	63
UploadParameters	1	1	0	6	4	2	50	73	75	76	80	59	25	51
WeightedQuickUnion	1	1	0	104	100	4	80	92	75	91	100	100	100	100
WhatTime	4	4	0	2	2	0	100	90	84	95	33	27	0	27
XmlUtils	1	1	0	4	3	1	67	87	88	92	100	96	94	99

The final four columns provide the equivalent coverage information for the generated tests. Comparing, for example, instruction coverage, shows that the generated tests improved coverage for 18 of the 52 tests (35%), had the same coverage for 11 of the tests (21%), and had worse coverage for 23 of the tests (44%). Note that these numbers include the 11 cases where no tests were generated, so in essence, only12 of the 41 generated tests (29%) had lower instruction coverage than the original tests.

Most of the cases of lower coverage occurred in situations where the code to be tested included multiple functions. Our keyword search generally only targeted one of these functions for coverage and hence only tried to generate test cases for a portion of the code. In these cases it might be possible to change the keywords to test the other functions in the original source and then combine both generated test cases as the result.

D Threats to Validity

There are several potential problems and threats to the accuracy and validity of this experiment. First there is some bias in the way the samples were selected. Because we only looked at files that could compile on their own, the code that is being tested is relatively simple. The methodology we use, unlike some test generation strategies, is not affected by the complexity of the code being tested. The fact that the code is simple might also make it more likely that there is similar code elsewhere in the repository and thus that test cases for the are easier to find. While this might be true, we expect that the repositories will keep growing and this will become less of a problem.

Second, because we only looked at code that already had test cases, the code is both testable and possibly designed to be tested. As noted with the text book examples, it is possible to write code that is difficult or impossible to test. Our

tool is not designed to address these situations. However, the text book examples, which were not designed with testing in mind, do show that our method can work for arbitrary code.

Third, in doing the experiment, we did find a little overlap between GitHub and OpenHub. A small number of the tests therefore included something like the original tests in their output.

Fourth, there is some randomness involved in the way S^6 handles choosing which solutions to transform if the set of viable solutions grows larger at any point grows larger 2000 elements. This could cause slightly different results on different runs and slightly different timings between runs. Similarly, the results returned by GitHub (and OpenHub) tend to change over time both because the underlying repositories are updated and because the search engines may give slight different results at different times. This is ameliorated somewhat by the fact that we cached the results from the search engines for the experiment.

Fifth, the whole system is sensitive to the initial selection of keywords describing the tests to be found; using different keywords could result in better or worse results and different timings. As noted, we experimented with different keywords on many of the examples. Note that the time involved is small enough so that such experiments by the end programmer are quite feasible.

Finally, we note that measuring the number of test cases and the coverage provided by the test cases is only one way of assessing the quality of tests. Another measure, for example, would be whether the test cases can actually find bugs. Since we are assuming that the repository code we retrieved is working, this would be more difficult to assess and would require a very different experiment. We do note that some of the examples where our test cases failed, for example in the handling or null or negative numbers, could indicate possible bugs in the original code.

E Evaluation Conclusion

The experiment shows that finding test cases for testable code using code search is feasible and can match the test numbers and coverage of existing human-written tests. We are able to generate tests in 79% (41/52) of the tests and to generate test coverage that is as good or better than the original tests in 56% (29./52) of the tests. Moreover, the time involved in generating the test cases is generally under a minute, with a median time of under 30 seconds. Even the longest runs took only several minutes, and these generated a significant number of tests.

One conclusion is that this type of facility can be an aid to test generation, but not a replacement for it. The technique can be used to generate an initial set of tests for existing code with little cost. It can also be used to augment an existing test suite with additional tests that others have thought of.

VI Future Work

So far we have developed the basic framework for generating test cases using code search and demonstrated that it can be done and can be done practically. One difficulty is properly translating tests in the repository to testing the user code. There are several ways our work could be extended to address this. For example, one could extend the transformation that handles matching calls in the found solutions with calls to the routines to be tested by taking into account different parameter orders, missing or extra parameters and parameter type conversions. This could be done, for example, using the type-directed synthesis techniques of [13]. Similarly, one could look at the results of the tests and take into account simple consistent errors such as off-by-one or wrong case string results.

Another problem is that it is difficult to test code that is part of a complex system. All of the cases cited above were for what we call leaf classes, that is classes that can be compiled and run without reference to the rest of the system. It is generally much easier to find external test cases for these since one does not have to find supporting classes in the repository code that are equivalent to the supporting classes in the code to be tested and since testing the class does not involve creating and initializing the overall system into a state where the class can be tested. We are considering several approaches to dealing with this. One is to extend the initial transformations to map classes and methods in the retrieved code to appropriate user classes and methods. The problem here is that the number of potential mappings can be very large and some semantic information or programmer guidance will be needed to make the checking tractable. A second alternative would be for the user to provide appropriate initialization code and sample values for the test and then doing the search. A third alternative is to search the project being tested for appropriate initialization code and potential parameters as is done in MSeqGen [52].

VII Conclusion

We have demonstrated a system that can quickly and effectively generate test code for a set of user function in a class by searching for and adapting test cases from open source repositories. This is a new approach that has the potential to make it easy to build or augment a relevant set of test cases for arbitrary code with little work on the part of the programmer. While work remains, we feel that this approach will eventually yield a practical system for test code generation.

S^6 source code is available from our ftp site (ftp://ftp.cs.brown.edu/u/spr). This includes the code used in test case generation as well as the code that scans OpenHub to generate the set of test cases. Code for our test cases and the scripts we used in generating the output are available upon request.

VIII References

1. Marat Akhin, Nikolai Tillmann, Manual Fahndrich, Jonathan de Halleux, and Michal Moskal, "Search by example in touch develop: code search made easy," *Proceedings SUITE 2013*, pp. 5-8 (June 2012).
2. Elvira Albert, Miguel Gomez-Zamalloa, and G. Puebla, "PET: a partial evaluation-based test case generation tool for Java bytecode," pp. 25-28 in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, (2010).
3. 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," *Proceedings*

ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2006, pp. 682-682 (October 2006).

4. Sushil Bajracharya, Joel Ossher, and Cristina Lopes, "Sourcerer: an infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming* **79** pp. 241-259 (2014).
5. Andrew Begel, "Codifier: a programmer-centric search user interface," *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).
6. Cristian Cadar, Daniel Dunbar, and Dawson Engler, "KLEE: unassisted and automatic generation of high- coverage tests for complex systems programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209-224 (2008).
7. Yoonsik Cheon and Carlos E. Rubio-Medrano, "Random test data generation for Java classes annotated with JML specifications," *SERP* **11** pp. 385-392 (June 2007).
8. 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* **26**(7) pp. 815-832 (July 1996).
9. Shih-Chien Chou and Yuan-Chien Chen, "Retrieving reusable components with variation points from software product lines," *Information Processing Letters* **99** pp. 106-110 (2006).
10. Christopher G. Drummond, Dan Ionescu, and Robert C. Holte, "A learning agent that assists the browsing of software libraries," *IEEE Transactions on Software Engineering* **26**(12) pp. 1179-1196 (December 2000).
11. EclEmma, "JaCoCo JHava Code Coverage Library," http:// www.eclemma.org/jacoco/ (2015).
12. 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).
13. Y. Feng, Y. Wang, R. Martins, A. A. Kaushik, and I. Dillig, "Type-directed component-based synthesis using Petri nets.," *Technical Report, University of Texas, Dallas Department of Computer Science*, (2016).
14. William B. Frakes and Thomas P. Pole, "An empiracal study of representation methods for reusable software components," *IEEE Transactions on Software Engineering* ***20***(8) pp. 617-630 (August 1994).
15. Zachary P. Fry and Westley Weimer, "A human study of fault localization accuracy," *Proceedings of IEEE International Conference on Software Maintenance*, pp. 1-10 (2010).
16. Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby, "A search engine for finding highly relevant applications," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (May 2010).
17. Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia, "Evaluating the specificity of text retrieval queries to support software engineering tasks," *Proceedings of the 2012 International Conference on Software Engineering*, pp. 1273-1276 (2012).
18. Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies, "Automatic query reformulations for text retrieval in software engineering," *Proceedings of the 2013 International Conference on Software Engineering*, pp. 842-851 (2013).
19. Robert J. Hall, "Generalized behavior-based retrieval," *Proceedings International Conference on Software Engineering,93*, pp. 371-380 (May 1993).
20. Emily Hill, Lori Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL- queries for software maintenance and reuse," *International Conference on Software Engineering 2009*, (May 2009).
21. Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proceedings UIST 2007*, pp. 13-22 (October 2007).
22. Oliver Hummel, Werner Janjic, and Colin Atkinson, "Code Conjurer: pulling resusable software out of thin air," *IEEE Software* **25**(5) pp. 45-52 ().
23. Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson, "Lowering the barrier to reuse through test- driven search," *SUITE,09*, pp. 21-24 (May 2009).
24. Werner Janjic and Colin Atkinson, "Leveraging software search and reuse with automated software adaptation," *Proceedings SUITE 2013*, pp. 23-26 (June 2012).
25. 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).
26. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Paulo Masiero, and Cristina Lopes, "Applying test-driven code search to the reuse of auxiliary functionality," *Proceedings ACM Symposium on Applied Computing*, pp. 476-482 (2009).
27. Otavio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology* **53**(4) pp. 294-306 (April 2011).
28. Greg Little and Robert C. Miller, "Keyword programming in Java," *Proceedings ASE 2007*, pp. 84-93 (November 2007).
29. Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," *Proceedings of the twenty-second IEEE/ ACM international conference on Automated software engineering*, pp. 234-243 (2007).
30. Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering* **17**(8) pp. 800-813 (August 1991).
31. Leonardo Mariani, Mauro Pezz, Oliviero Riganelli, and Mauro Santoro, "AutoBlackTest: a tool for automatic black-box testing," pp. 1013-1015 in *Proceedings of the 33rd International Conference on Software Engineering*, (2011).
32. L Martie, T. D. LaToza, and A. van der Hoek, "CodeExchange: supporting reformulation of code queries in context," *Proceedings of the 30th International Conference on Automated Software Engineering*, (2015).
33. Lee Martie and Andre Van der Hoek, "Toward social- technical code search," *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 101-104 (2013).
34. Lee Martie and Andre Van der Hoek, "Sameness: an experiment in coe search," *K mhsamen*, pp. 76-87 (2015).
35. Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu, "Portfolio: finding relevant functions and their usage," *Proceeding of the 33rd International Conference on Software engineering*, (May 2011).
36. Alon Mishne, Sharon Shoham, and Eran Yahav, "Typestate- based semantic code search over partial programs," *SIGPLAN Notices* **47**(10) pp. 997-1016 (2012).
37. Santanu Paul and Atul Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering* **20**(6) pp. 463-475 (June 1994).
38. Andy Podgurski and Lynn Pierce, "Retrieving reusable software by sampling behavior," *ACM Transactions on Software Engineering and Methodology* **2**(3) pp. 286-303 (July 1993).
39. Steven P. Reiss, "A component model for Internet-scale applications," *Proceedings ASE 2005*, pp. 34-43 (November 2005).
40. Steven P. Reiss, "Semantics-based code search," *International Conference on Software Engineering 2009*, pp. 243-253 (May 2009).
41. Steven P. Reiss, "Specifying what to search for," *Proceedings SUITE 2009*, (May 2009).
42. Steven P. Reiss, "Seeking the user interface," *Proceedings of 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 103-114 (2014).
43. Edwin Rodriguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby, "Extending JML for modular specification and verification of multi- threaded programs," *SAnToS Laboratory Tech Report TR2004-10, Kansas State University*, (May 2005).
44. Philip Samuel and Rajib Mall, "Slicing-based test case generation from UML activity diagrams," *SIGSOFT Softw. Eng. Notes* **34**(6) pp. 1-14 (2009).

45. Bunyamin Sisman and Avinash C. Kak, “Assisting code search with automatic query reformulation for bug localization,” *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 309-318 (2013).
46. Kathryn T. Stolee and Sebastian Elbaum, “Toward semantic search via SMT solver,” *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1-4 (2012).
47. Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos, “Solving the Search for Source Code,” *ACM Trans. Softw. Eng. Methodol.* **23**(3) pp. 1-45 (2014).
48. Vijayan Sugumaran and Veda C. Storey, “A semantic-based approach to component retrieval,” *Advances in Information Systems* **34**(3) pp. 8-24 (2003).
49. Watanabe Takuya and Hidehiko Masuhara, “A spontaneous code recommendation tool based on associative search,” *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pp. 17-20 (2011).
50. Stephen W. Thomas, “Mining Unstructured Software Repositories using IR Models,” *Ph.D. Dissertion from Queen,s University, Canada*, (2012).
51. Suresh Thummalapenta and Tao Xie, “PARSEWeb: a programmer assistant for reusing open source code on the web,” *Proceedings ASE,07*, pp. 204-213 (November 2007).
52. Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte, “MSeqGen: object-oriented unit-test generation via mining source code,” pp. 193-202 in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, (2009).
53. 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,” *Proceedings SAC,07*, pp. 866-871 (March 2007).
54. Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang., “Mining succinct and high- coverage API usage patterns from source code,” *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*, pp. 319-328 (May 2013).
55. Shaowei Wang, David Lo, and Lingxiao Jiang, “Code search via topic-enriched dependence graph matching,” *18th Working Conf. on Reverse Engineering*, pp. 119-123 (2011).
56. Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal, “SnipMatch: using source code context to enhance snippet retrieval and parameterization,” *Proceedings of the 25th annual ACM symposium on User Interface Software and Technology*, pp. 219-228 (2012).
57. Haining Yao and Letha Etzkorn, “Towards a semantic- based approach for software reusable component classification and retrieval,” *ACMSE,04*, pp. 110-115 (April 2004).
58. Yunwen Ye and Gerhard Fischer, “Supporting reuse by delivering task relevant and personalized information,” *Proceedings International Conference on Software Engineering,02*, pp. 513-523 (May 2002).
59. Yunwen Ye, “Programming with an intelligent agent,” *IEEE Intelligent Systems* **18**(3) pp. 43-47 (May 2003).