IOWA STATE UNIVERSITY
Digital Repository

Graduate Theses and Dissertations | Graduate College

2015

An automated approach to program repair with semantic code search

Yalin Ke
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Ke, Yalin, "An automated approach to program repair with semantic code search" (2015). *Graduate Theses and Dissertations.* Paper 14813.

This Thesis is brought to you for free and open access by the Graduate College at Digital Repository @ Iowa State University. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.

An automated approach to program repair with semantic code search

by

Yalin Ke

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Kathryn T. Stolee, Major Professor

Pavan Aduri

Wei Le

Iowa State University

Ames, Iowa

2015

Copyright © Yalin Ke, 2015. All rights reserved.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my thanks to those people who helped me with conducting research and the writing of this thesis.

First and foremost, I would like to thank my academic advisor, Kathryn T. Stolee, for her guidance, patience and support throughout this research and the writing of this thesis. She has inspired me by her insights and words.

I would also like to thank my committee members for their efforts and contributions to this work: Pavan Aduri, Wei Le.

I would also like to thank my labmates, for their help on solving technical problems when I was coding the project for experiments: Danilo.

ABSTRACT

Every year software companies dedicate numerous developer hours to debugging and fixing defects. Automated program repair has the potential to greatly decrease the costs of debugging. Existing automated repair techniques, such as Genprog, TSPRepair, and AE, show great promise but are not able to repair all bugs. We propose a new automated program repair technique, SearchRepair, which is a complementary program repair technique. We take advantage of existing open source code to find potential fixes based on the assumption that there are correct implementations in open source project code for some defects. The key challenges lie in efficiently finding code semantically similar (but not identical) to defective code and then appropriately integrating that code into the buggy program. The technique we present, SearchRepair, addresses these challenges by (1) encoding a large database of human-written code fragments as SMT constraints on input-output behavior, (2) localizing a given defect to likely-buggy program fragments, (3) dynamically analyzing those buggy fragments to derive input-output pairs that describe likely buggy behavior and that can be encoded as SMT constraints, (4) using state-of-the-art constraint solvers to find fragments in the code database that satisfy those constraints, and (5) validating patches that repair the bug against program test suites.

We evaluate our technique, SearchRepair, on a program repair benchmark set *IntroClass*, which provides 998 buggy programs written by novice students, two test suites for each program, and repair results for existing program repair technique, Genprog, TSPRepair and AE. The two test suites, of which one is written by a human and the other one is automatically generated by a computer, are used to determine if a program is buggy and to evaluate the quality of a repair. We use *instructor test suite* to refer the test suite that is written by a human. And we use *KLEE test suite* to refer the test suite that are generated by the computer. We consider a program as a potential fixable defect if it fails and passes at least one test case in a test suite. Note that extracting input-output behaviors for the semantic code search requires that at least one passed test case so some buggy programs are excluded from our evaluation. There are 778 defects in *IntroClass* based on the instructor test suite and 845 defects in

IntroClass based on the KLEE test suite. We find that when using the instructor test suite, SearchRepair is able to successfully repair 150 of 778 defects, Gengprog is able to fix 287 defects, TSPRepair is able to fix 247 defects, AE is able to fix 159 defects. In total, these 4 techniques are able to fix 310 defects using the instructor test suite and 20 of the 310 defects can only be fixed by SearchRepair. We also find that when using the computer generated test suite, there are 58 unique defects that can only fixed by SearchRepair out of 339 total unique defects that can be fixed by the 4 techniques. These results suggest that SearchRepair is a complementary technique to existing program repair techniques.

CHAPTER 1. INTRODUCTION

Suppose Alex is a freshman in Computer Science who is currently taking the C Programming Language course. Alex's homework is to write a program that can output the smallest of three numbers. In order for students to be able to verify their program's correctness, the instructor provides a test suite, which is listed in Table 1.1, and is supposed to be passed completely by a correct submission. Each test case in the test suite has a set of 3 numbers as input and one number as expected output. Alex writes a program shown in Figure 1.1. When Alex runs the program with the test cases in Table 1.1, he finds that his program only passes test cases t_1, t_2, t_3, but fails test cases t_4 and t_5. It is evening and Alex is at home so he is unable to ask his instructor for help. Alex turns to SearchRepair, which is the automatic program repair tool proposed by this thesis. SearchRepair is able to pinpoint line 12 as the perpetrator to cause Alex's program to fail test cases t_4 and t_5. Line 12 assigns the value of variable c to variable *smallest* regardless of whether c is actually the smallest when variable $a = b$. SearchRepair not only is able to locate the buggy lines, and it also has a built-in code repository that it can use to search for correct code fragments as replacements to the buggy lines. For example, SearchRepair suggests that code snippets shown in Figure 1.2 can be a replacement for lines 7 to 12 of Alex's program in Figure 1.1. SearchRepair then replaces the buggy lines with the replacement to provide a correct program in Figure 1.3 which is able to pass all of the test cases in Table 1.1 provided by the instructor.

Programmers, like Alex, frequently encounter bugs during software development. There are several ways to fix a bug. One is using a debugger to run the program and trying to pinpoint the buggy lines manually. Yet, program debugging is time consuming. A second common way is to seek answers from others by performing web searches or asking colleagues. Yet, online searches may not yield correct results and colleagues my not be available. A third way is to use automated program repair tools. SearchRepair is one of these tools.

```c
#include<stdio.h>

int main()
{
    int a, b, c, smallest;
    printf("Please input 3 numbers");
    scanf(%d%d%d, &a, &b, &c);
    if(a < b && a < c)
    {
        smallest = a;
    }
    else if(b < a && b < c) smallest = b;
    else smallest = c;
    printf(" %d is smallest", smallest);
    return 1;
}
```

Figure 1.1: Program written to output the smallest of three numbers

Table 1.1: Test suite for example assignment

case	input	expected output
t_1	1, 2, 3	1
t_2	3, 2, 1	1
t_3	3, 4, 4	3
t_4	1, 1, 3	1
t_5	2, 2, 5	2

Fixing bugs is not only time consuming for programmers, it also matters a lot to the software industry. Buggy software costs the global economy billions of dollars annually (Research Triangle Institute, 2002). A well known software problem, *Year 2000 problem*, shut down a lot of servers and computers worldwide when January 1st 2000 came. Every year software companies must dedicate considerable developer time (**?**) to manually finding and fixing bugs in their software. However, as the amount of software in the industry increases, manual software repair cannot keep up with the increasing number of bugs (Anvik et al., 2005). Despite their detrimental impact on a company's bottom line, known defects are continuing

```
    if(a <= b && a <= c)
    {
    smallest = a;
    }
    else if(b <= a && b <= c) smallest = b;
    else smallest = c;
```

Figure 1.2: Code snippet fix to program in Figure 1.1

```
#include<stdio.h>

int main()
{
    int a, b, c, smallest;
    printf("Please input 3 numbers");
    if(a <= b && a <= c)
    {
        smallest = a;
    }
    else if(b <= a && b <= c) smallest = b;
    else smallest = c;
    printf(" %d is smallest", smallest);
    return 1;
}
```

Figure 1.3: Program provided by SearchRepair to output the smallest of three numbers

to ship in mature software projects (Liblit et al., 2003). Many defects, including security-critical bugs, remain unaddressed for a very long period of time (Hooimeijer and Weimer, 2007). Security-critical bugs, once exposed, could cost millions of dollars.

At the same time, the expansion of the open-source movement has led to many large, publicly accessible source code databases, such as Github and Bitbucket. Because many programs implement routines, data structures, and designs that have been previously implemented in other software projects (Carzaniga et al., 2010, 2013; Gabel and Su, 2010), we posit that, if a method or component of a software system contains a defect, with high probability, there exists a correct, similar version of that component in some publicly accessible software project.

Since program bugs are so costly and dangerous, and manual program repair is unable to keep up with the pace of the increasing of number of bugs, programmers may wonder if there is a tool that can automatically locate the fault of a program and provide a correct patch replacement. In this thesis, we propose a tool SearchRepair, that can fix these bugs automatically by creating patches from open-source code repositories. The challenge is to automatically find and use existing code to fix bugs. SearchRepair utilizes a semantic search technique proposed by (Stolee et al., 2014), to search over existing open-source code and find correct implementations of faulty code fragments and methods. SearchRepair uses the results to automatically generate patch replacements for software bugs.

SearchRepair has the following four main features.

1. Encodes a large database of human-written code fragments as SMT constraints on their input-output behavior.

2. Localizes a given defect to likely-buggy program fragments and constructs lightweight input-output profiles that characterize the desired behavior of those regions as SMT constraints.

3. Uses state-of-the-art constraint solvers to find fragments in the code database that satisfy those constraints.

4. Validates patches for the bug against program test suites.

To realize these four features, SearchRepair makes several research adaptations to existing techniques.

1. We adapt our previous semantic code search encoding techniques (Stolee et al., 2014) to encode code databases of C fragments. (Stolee et al., 2014) proposed an encoding technique for code databases of Java methods and fragments. We extend that technique for the C language.

2. We adapt spectrum-based fault localization (Jones et al., 2002) to identify candidate regions of faulty code and construct input-output profiles to use as queries for semantic search.

3. We build the infrastructure to perform semantic code search over the SMT-encoded code database, adapt the returned code fragment to the defective context via variable renaming, and validate against provided test suites.

SearchRepair is different from prior repair techniques (Demsky et al., 2006a; Jin et al., 2011; Kim et al., 2013; Le Goues et al., 2012; Nguyen et al., 2013; Qi et al., 2013; Perkins et al., 2009; Weimer et al., 2009, 2013; Wei et al., 2010) because it bridges the gap between correct-by-construction techniques predicated on human-written annotations and generate-and-validate techniques that heuristically create and then test large numbers of candidate repairs. While several techniques in the latter class reuse human-written code from elsewhere in the program or instantiate human-written templates to affect local changes, SearchRepair replaces larger regions of code wholesale with human-written code from other projects. Our assumption is that larger fragments of human-written code that satisfy a partial specification are more likely to satisfy the full, desired, often unwritten specification than are smaller fragments of computer-synthesized code. Our evaluation shows that at least in the context of our experiments, this assumption holds.

We evaluate our technique, SearchRepair, on a program repair benchmark set *IntroClass* (Le Goues et al., 2015), which provides 998 buggy programs written by novice students, two test suites for each program, and repair results for existing program repair techniques GenProg, TSPRepair and AE. The two test suites, of which one is written manually by instructor and the other one is generated by the KLEE algorithm (klee, 2008a), are used to determine if a program is buggy and evaluate the quality of a repair. A program is considered as a potential fixable defect if it fails at least one test case and passes at least one test case in a test suite. The 998 defects provided by *IntroClass* are the union of 778 defects based on the instructor test suite and 845 defects based on the KLEE test suite. We find that when using the instructor test suite, SearchRepair is able to successfully repair 150 of 778 defects, Gengprog is able to fix 287 defects, TSPRepair is able to fix 247 defects, and AE is able to fix 159 defects. In total, these 4 techniques are able to fix 310 defects using the instructor test suite and 20 of the 300 defects can only be fixed by SearchRepair. We also find that 146 of the 150 repairs SearchRepair finds can completely pass an independent test suite, which suggest repairs found by SearchRepair have a very high quality when using the instructor test suite.

We also find that when using the KLEE test suite, there are 58 unique defects that can only be fixed by SearchRepair out of 339 total unique defects that can be fixed by the 4 techniques. These results suggest that SearchRepair is a complementary technique to existing techniques. When using the KLEE

test suite to find repairs, SearchRepair is able to find 186 defects, TSPRepair is able to find 202 defects, AE is able to find 140 defects, and GenProg is able to find 270 defects.

1.1 Contributions

The main contributions of this thesis are:

1. An extension of state-of-the-art semantic code to new primitives and functions, and an implementation for the C programming language, which is detailed in Section 6.3.

2. SearchRepair, a semantic-code-search-based automated program repair approach and its implementation, including an extension of spectrum-based fault localization for use in identifying candidate fragments of defective code. The approach of SearchRepair is detailed in Chapter 5.

3. An in-depth evaluation on a benchmark *IntroClass* comparing SearchRepair to three prior repair techniques showing that SearchRepair can repair some defects that prior techniques cannot. Chapter 7 discusses the comparisons.

1.2 Outline of thesis

The remainder of this thesis is structured as follows. Chapter 2 describes why SearchRepair improve program repair quality. Chapter 3 describes three important concepts (*InroClass* benchmark, Symbolic Execution, Semantic code search), which provides assistance for understanding this thesis. Chapter 4 introduces an illustrative example to fully explain the process of how SearchRepair finds a fix. Chapter 4 also provides several examples to illustrate the features of SearchRepair. Chapter 5 details the SearchRepair approach. Chapter 6 details how SearchRepair is implemented. Chapter 7 evaluates SearchRepair against three other repair techniques, *TSPRepair*, *GenProg*, *AE* on a benchmark set of 998 defects that fail and passes at least one test case. Chapter 7 also discusses the threats to validity. Chapter 8 describes related work in semantic code search and program repair. Finally, Chapter 9 summarizes our contributions and future work. Larget parts of this thesis are published in the International Conference on Automated Software Engineering (Ke et al., 2015).

CHAPTER 2. MOTIVATION

The cost of debugging and maintaining software has continued to rise, even while hardware and software costs fall. A 2013 study estimated the global cost of debugging at $312 billion, and software developers spending half their time debugging (Britton et al., 2013). Since there are not enough developer resources to repair all of these defects before deployment, programs ship with both known and unknown bugs (Liblit et al., 2005). In response to this problem, many organizations offers rewards for outside software developers for reporting potential bugs and fixes. For example, Google has offered an average of $500 for each bug outside developers report. Companies receive potential bug reports from those outside developers and then have their developers inspect those reports to fix those bugs, which means more software developers are hired to inspect bug reports. An automated program repair tool may have the potential to save resources for companies like Google. An automated program repair technique is able to detect potential bugs in a program by using some predefined test cases and find potential fixes to those bugs.

The detrimental cost of defective software motivates research in automatically and generically repairing bugs. The results of recent research efforts on this issue have been quite promising (Demsky et al., 2006b; Jin et al., 2011; Kim et al., 2013). One class of techniques constitute *correct-by-construction repair strategies*, which rely on specifications (inferred or provided) to guide sound patch generation (Nguyen et al., 2013; Wei et al., 2010; Demsky et al., 2006b; Weimer, 2006; Jin et al., 2011). This provides confidence that the output is correct. However, such techniques struggle to scale and are usually (though not always) limited to formally specified code.

The other primary class of repair approaches are *generate-and-validate repair techniques*, which use search-based software engineering (Harman, 2007) or predefined repair templates (Perkins et al., 2009; Kim et al., 2013) to generate many candidate repairs for a bug and then validate those repairs using test suites. Test cases are the most common form of validation, the input to such a technique is typically a

program and a set of test cases. Initially-passing tests validate the correct, required behavior, that should be maintained post-repair and initially-failing tests identify the buggy behavior to be repaired.

Prior work has shown that generate-and-validate repair may exhibit poor quality by producing patches that overfit to the specification used during patch generation (Smith et al., 2015; Qi et al., 2015). Intuitively, the repair techniques are only aware of a partial specification for the desired repair. Typically, this partial specification is a set of tests. Since there are many programs that satisfy any given partial specification, the repair techniques can produce any one of them, and with high probability, this repaired program will not adhere to the unwritten, full, desired specification.

We posit that using larger fragments of human-written code that satisfy a partial specification to repair programs is more likely to generalize to the unwritten specification. The intuition behind our argument is that because code is repetitive and often re-implementations routines (Carzaniga et al., 2010, 2013), a human-written routine that fits a partial-specification is more likely to satisfy the unwritten specification than a randomly chosen one of a set of generated routines for the partial specification.

Consider an example of a program that has a bug in a subroutine that sorts an array of integers. No finite set of tests can uniquely define sorting the array, and using a test suite, automated program repair is as likely to produce a sorting routine as it is to produce a different routine that works for the example tests but fails on other, unwritten tests. However, in a large body of human-written code, there are many more sorting routines than other routines that satisfy these tests, so searching for such a human-written routine is more likely to generate to the unwritten specification.

The rest of this paper describes the technique that makes repair via searching for human-written code possible and evaluates our hypotheses that such human-written code can be used to repair defects, and that the quality of the repairs is higher. Chapter 7 shows that this approach does repair more defects, that it repairs some defects that prior techniques cannot repair, and that the resulting repairs pass, on average, 142 out of 145 of the independent tests not used during repair generation.

CHAPTER 3. BACKGROUND

SearchRepair combines two areas of software engineering: automated program repair and semantic code search. In this chapter, we introduce the concepts from automated program repair required to understand the approach and domains used for illustration and evaluation throughout the rest of this work. Our approach to program repair is applied to C programs. Section 3.1 introduces a benchmark set, *IntroClass*, for evaluating program repair techniques. SearchRepair heavily relies on two existing techniques: symbolic execution which is detailed in Section 3.2 and semantic code search which is described at Section 3.3.

3.1 *IntroClass* benchmark for evaluating automated program repair techniques

In this section we describe the *IntroClass* benchmark introduced by (Le Goues et al., 2015). A benchmark is a standard on which multiple techniques could be compared against. *IntroClass* can be used to compare different automatic program repair techniques. *IntroClass* not only provides 998 defects, but it also provides results of three existing techniques: *GenProg*, *TSPRepair* and *AE*. *IntroClass* allows researchers to compare new program repair techniques with these three existing techniques.

All of the programs from the *IntroClass* set are collected from a C programming class at UC Davis. Students taking that class were required to write C programs that satisfy the instructor-provided specifications. There are six assignments, *Median*, *Smallest*, *Syllables*, *Digits*, *CheckSum*, and *Grade*, described in Table 3.1. Students are able to submit several versions for each assignment. Every time a student submits a version for an assignment, that version is tested with two test suites: the instructor test suite and the KLEE test suite. The instructor test suite consists of test cases written by the instructor based the specification of each assignment. The KLEE test suite consists of test cases that are generated by computer based on KLEE algorithm (klee, 2008a). Students are able to receive the testing results

after every submission. Based on the results, students may rethink and redo their implementations, then resubmit.

Our definition for a buggy program is that it must fail and pass at least one instructor test case or fail and pass at least one KLEE test case. By this definition, the *IntroClass* benchmark contains 998 such submitted buggy programs. In addition, *IntroClass* also provides correct programs for each assignment. *IntroClass* also contains programs that fails an entire test suite, which in this thesis are considered unfixable defects for SearchRepair. The reason why SearchRepair considers programs that have no passed test cases as unfixable defects is detailed in Section 3.3.2. SearchRepair only focuses on the 998 buggy programs that fails and passes at least one instructor test case or at lease one KLEE test case. Due to differences between the instructor test suite and the KLEE test suite, some programs may be considered as correct programs for one test suite but not correct programs for the other test suite. Table 3.1 summarizes, *IntroClass* consists of a total of 778 defects using the instructor test suite, and 845 defects using the KLEE test suite.

Table 3.1: The six *IntroClass* benchmark subject programs. The instructor test suite is instructor-written specification based test suite, and the KLEE tests are generated with KLEE to give 100% branch coverage on the instructor written reference implementation. The 998 unique defects are student-submitted versions that fail at least one, and pass at least one of the tests.

IntroClass benchmark						
program	LOC	tests		defects		description
		instructor	KLEE	instructor	KLEE	
median	24	7	6	160	168	median of 3 numbers
syllables	23	6	10	109	115	count vowels
smallest	20	8	8	155	113	min of 4 numbers
grade	19	9	9	226	224	grade from score
checksum	13	6	10	29	49	checksum of a string
digits	15	6	10	91	199	digits of a number
total	114	42	53	778*	845*	
The union of the 778 instructor and 845 KLEE defects is 998 defects						

3.2 Symbolic execution

SearchRepair has a repository which consists of code snippets scraped from existing open source projects, which it uses to find patches for faulty code. SearchRepair extracts all feasible paths of each code snippet and converts each path into SMT constraints. Code snippets along with the SMT constraints of their feasible paths are stored in the repository. SearchRepair utilizes an existing program analysis technique, symbolic execution, to extract feasible paths of a code snippet. This section introduces the details of symbolic execution. The concept of SMT constraints is described in Section 3.3.

Symbolic execution determines which part of a program is possibly executed by some inputs (klee, 2008b). When symbolic execution is performed, a program analyzer interprets each statement by assuming symbolic values instead of getting the actual values for inputs. The main difference with normal execution is that symbolic execution treats inputs as symbolic. It allows inputs to be anything in the domain set by the inputs' data type whereas there are always concrete values for inputs during a normal execution.

To give an idea of how symbolic execution works, consider the program *Difference* in Figure 3.1. This program takes as inputs two numbers and returns the absolute value of the difference between these two numbers. To identify all feasible paths in program *Difference*, symbolic execution does not assign any concrete value to variable a and b. It treats a and b as symbolic over the domain of integers. The symbolic execution constructs paths based on predicates. Line 4 introduces a predicate $a > b$, which can evaluate to true, leading to $c = a - b$ in line 5, or false, which leads to $c = b - a$ in line 7. This predicate creates divergent control flow resulting in two program paths. Figure 3.2 shows the path when predicate $a > b$ evaluates to true. Figure 3.3 shows the path when predicate $a > b$ evaluates to false. In our approach, every multiple path program is interpreted as a disjunction of multiple independent paths.

3.3 Semantic code search

Semantic code search has two parts. One part is encoding source code as Satisfiability Modulo Theory (SMT) constraints, which is used to build a repository. Section 3.3.1 introduces how to encode the pathes of a code snippet into SMT constraints. The other part is searching the repository with an input-output specification. Section 3.3.2 describes how to search the repository.

```
1   int Difference(int a, int b)
2   {
3       int c;
4       if(a > b)
5           c= a - b;
6       else
7           c = b - a;
8       return c;
9   }
```

Figure 3.1: Program that calculates the difference between two numbers

```
1   int a;
2   int b;
3   int c;
4   assert(a > b);
5   c = a - b;
6   return c;
```

Figure 3.2: Path 1 for program in Figure 3.1

Syntactic code search uses syntactic features such as keywords and variable names as the specification. For example, a developer trying to find a method for `String` replacement in C might search for "`C string replace`". By contrast, *semantic* search uses behavioral properties as the specification. For example, the developer may supply several input-output pairs for the desired string replacement function. Semantic code search offers the notable advantage over keyword-based search because a developer can search by example, and need not guess the words that describe the behavior. This makes it particularly amenable to program repair since we can use program behavior to create input-output examples to drive the search.

3.3.1 Encoding source code to build a repository

We adapt and extend prior work (Stolee et al., 2014; Stolee and Elbaum, 2012; Stolee, 2013) on input-output example-based semantic code search in Java. SearchRepair encodes primitives, statements, and library methods in C. Just like any other search engine, SearchRepair builds a repository of source code offline, independent of user queries. The encoding process has two too parts. One part is to use

```
1       int a;
2       int b;
3       int c;
4       assert(a <= b);
5       c = b - a;
6       return c;
```

Figure 3.3: Path 2 for program in Figure 3.1

symbolic execution (Clarke, 1976) to decompose a program into several feasible program paths. The other part is to convert each feasible path into a separate set of SMT constraints.

To give an idea of the encoding process, consider a C method in Figure 3.4, which outputs the max length of two C strings.

```
1  int maxStrlen(char* left, char* right)
2  {
3      int h = strlen(left);
4      int r = strlen(right);
5      if(h > r)
6          return h;
7      else
8          return r;
9  }
```

Figure 3.4: Function that returns the max length of two strings

SearchRepair first converts the C method *maxStrlen* into two paths, as shown in Figure 3.5 and Figure 3.6, by symbolic execution.

Then, for each path, its predicates and statements are all converted into SMT constraints. Each statement in each path is converted to an SMT constraint. For example, line 2 in Figure 3.5 is converted into the constraint `c1` in Figure 3.7. There are two types of SMT constraints in semantic code search: type constraint and value constraint. Constraints `c1,c2,c3` in Figure 3.7 and `c8,c9,c10` in Figure 3.8 declare the variables used in the method. They constrain variable types. Constrains `c4,c5,c6` in Figure 3.7 show the relationship between values of variables. They constrain variable values. Constraint `c6` is the corresponding constraint to the predicate of line 6 in Figure 3.7. Constraint `c4` is the corresponding

```
1    Path 1:
2    int h;
3    int r;
4    h = strlen(left);
5    r = strlen(right);
6    assert(h > r);
7    return h;
```

Figure 3.5: Path 1 for program in Figure 3.4

```
1    Path 2:
2    int h;
3    int r;
4    h = strlen(left);
5    r = strlen(right);
6    assert(h <= r);
7    return r;
```

Figure 3.6: Path 2 for program in Figure 3.4

constraint to the assignment statement of line 4 in Figure 3.7. Figure 3.7 is the corresponding set of SMT constraint to the first path in Figure 3.5. Figure 3.8 is the corresponding set of SMT constraints to the second path in Figure 3.6.

After generating constraints for each code snippet, SearchRepair stores them in a database. The database relates the constraints, the original source code, and the type signature of the snippet. For multi-path programs, they are stored as a disjunction of all the feasible paths represented as SMT constraints.

3.3.2 Searching with an input-output specification

A group of SMT constraints forms an SMT formula expressed in first-order logic and an SMT solver (de Moura and Bjorner, 2008) is used to find an assignment for that SMT formula. To check if a source code snippet satisfies a specified input-output behavior, SearchRepair follows the four steps:

1. Encodes the specified input-output behavior into SMT constraints.

```
c1. (declare-fun h () Int)
c2. (declare-fun r () Int)
c3. (declare-fun returnVal () Int)

c4. (assert (h = (length left))
c5. (assert (r = (length right))
c6. (assert (h > r))
c7. (assert (returnVal = h))
```

Figure 3.7: SMT constraints for the path code in Figure 3.5

```
c8. (declare-fun h () Int)
c9. (declare-fun r () Int)
c10. (declare-fun returnVal () Int)

c11. (assert (h = (length left))
c12. (assert (r = (length right))
c13. (assert (h <= r))
c14. (assert (returnVal = r))
```

Figure 3.8: SMT constraints for the path code in Figure 3.6

2. Encodes the candidate of C snippet from the repository into SMT constraints. This is part of the indexing process and is done offline, which is detailed in Section 3.3.1.

3. Combines the constraints from step 1 and step 2 and maps variables from input-output to the variables from the C snippet candidates. How to map variables is described in this section. These SMT constraints form an SMT formula expressed in first-order logic

4. Checks if there is an assignment of values that satisfies the SMT formula. An SMT solver is used to find an assignment for that SMT formula. The SMT solver returns sat and the assignment is encapsulated in the satisfiable model if an assignment is found. If an unsat is returned, then this code snippet does not match the input-ouput behavior. Then a new source code snippet is pulled out from the repository for matching util all of the code snippets in the repository are exhausted or a code snippet that matches the input-output behavior is found.

```
char* cat(char* first, char* second){
    char max[20];
    strcpy(max, first);
    strcat(max, second);
    return max;
}
```

Figure 3.9: Program to copy and concatenate two strings

To better understand this idea, suppose a user is searching for code that has the following two inputs:

```
1  *str1 = "sun";
2  *str2 = "moon"
```

and the following outputs:

```
1  *cat = "sunmoon";
```

Semantic code search first transforms this input-output information into SMT constraints.

```
1  c15.  (declare-fun str1 () String)
2  c16. (declare-fun str2 () String)
3  c17. (declare-fun cat () String)
4
5  c18. (assert (str1 = "sun"))
6  c19. (assert (str2 = "moon"))
7
8  c20. (assert (cat = "sunmoon"))
```

Then, it takes snippets out from the repository one by one, maps each input value to a variable in the snippet, and binds the output value to the returned value of the snippet. Consider a code snippet in Figure 3.9 stored in the repository,

the mapping constraints are:

```
c21. (assert ((first = str1) and (second = str2)) and (cat = max))
c22. (assert ((first = str2) and (second = str1)) and (cat = max))
```

Note that only one of the mapping constraints, `c21` or `c22`, can be satisfied at a given time. And only variables with the same type is paired. If a program have two integers as inputs, there is no mapping in this case and that program is skipped. Both constraints `c21` and `c22` do not appear in the same SMT constraint system. Also, the variables that from the input specification only map to the parameter

variables. And the variable from the output specification only maps to the variable in the return statement. In this case, there are two possible mappings.

During the repository indexing process, this snippet in Figure 3.9 has already be transformed into SMT constrains and stored in the database. These constraints are:

```
c23. (declare-fun max() String)
c24. (declare-fun first () String)
c25. (declare-fun second () String)
c26. (assert (max = (first + second)))
c27. (assert (output = max))
```

When the snippet is pulled from the repository, its corresponding SMT constraints `c23` to `c26` are also pulled out for matching. Then semantic code search combines the input-output specification constraints `c15` to `c20`, one of the mapping constraints `c21` or `c22` and the snippet constraints `c23` to `c26` into an SMT formula. The SMT solver then checks if there is an assignment that can satisfy the SMT formula. If an assignment is found, the snippet with that particular mapping satisfies the desired input-output specification. If no assignment is found, semantic code search tries to combine another mappings constraints until a satisfication is found or all mappings has been enumerated and it's determined the code is not a match for the specification.

If the user had searched for a different set of input-output pair, e.g. if the value of the output variable `cat` was *moonfate*, the solver would find that the set of constraints is unsatisfiable, meaning the snippet does not satisfy the input-output specification.

Searching using multiple input-output examples involves several iterations on this process, encoding each input-output example separately and pairing it with each code snippet. In this case, the same mapping constraint must match all input-output examples for a snippet to satisfy the specification. Suppose there are two input-output examples for this case: t_1 and t_2. The code snippet is considered as a potential correct fix only if the mapping constraint `c19` or `c20` satisfies both the input-output specifications of t_1 and t_2.

3.4 Summary

In this chapter, we discuss program repair related concepts: program repair benchmark *IntroClass*, symbolic execution and semantic code search. In the Chapter 5, we explain the approach of SearchRepair. In Chapter 6, we discuss the technical details of how to implement SearchRepair.

CHAPTER 4. ILLUSTRATIVE EXAMPLES

We propose SearchRepair, a tool for automated program repair using semantic code search. In this chapter, we first use an illustration to explain the process SearchRepair uses to locate and fix defects. Then we use several specific examples to show what kinds of buggy programs that SearchRepair can fix. The technical details behind SearchRepair are detailed in Chapter 5.

4.1 An illustration

In this section, we use an example in Figure 4.1 to explain the process of how SearchRepair locates and fixes a buggy program. SearchRepair finds a correct patch fix by following the five steps:

1. Any user who wants to use SearchRepair to fix a buggy program must provide several test cases. Each test case should have at least two parts: input and expected output. Expected output is the result that a program is expected to generate given the input. SearchRepair runs the buggy program with the input to obtain the actual output. The test cases for which the actual output is the same as the expected output are considered passed test cases. Test cases for which the actual output is different than the actual output are considered failed test cases.

2. SearchRepair uses fault localization to locate buggy lines of a program. Based on the test cases from step 1, SearchRepair uses a coverage-based technique to find buggy lines, which is detailed in Section 5.2.

3. After locating the buggy lines, SearchRepair inserts one print statement before the buggy lines and one print statement after the buggy lines. These two statements print the running-time value of each local variable. Then SearchRepair runs the buggy program with the inputs and captures the printed values of these two statements. Variable values form an input-output profile. Variable

values before buggy lines are called *input state*. Variable values after buggy lines are called *output state*. For passed test cases, their input-output profiles should not change even after a correct patch replaces the buggy lines. For failed test cases, some of the variables values of the *Output State* must change after the correct patch replacement.

4. Using only input-output profiles of positive test cases, SearchRepair constructs SMT constraints for the input-output profiles. Then SearchRepair uses semantic code search to find patches in a code repository that can replace the buggy lines. The concepts of SMT constraints and semantic code search are described in Section 5.4. Also the reason why SearchRepair does not use negative input-output profile is explained in Section 5.4. Once SearchRepair finds a potential patch, it renames variables in the patch by the mapping SMT constraint provided by semantic code search and replaces the buggy lines with that patch.

5. After replacing the buggy lines with the patch, SearchRepair re-runs the program on the test suite. If all of the test cases in the test suite pass, the patch is a correct fix, otherwise SearchRepair continues searching the repository to find potential patches, verifying patches until a correct patch is found or all entries in the repository are exhausted.

In this section, we use a code sample selected from a benchmark set *IntroClass* to illustrate the concepts of our approach. A defect in *IntroClass* consists of a buggy program and a test suite. Consider the C program in Figure 4.1 and the program's test suite in Table 4.1. This program takes three numbers as input and outputs the median of the three numbers. Let's use *Median* to denote this program. Line 16 `(c<=b && a>=c)|| (c>=b && a<=c)` is incorrect, when `c` is the median and `a` or `b` is equal to `c`, the program print 0 instead of `c`. When we run *Median* with the test suite in Table 4.1, *Median* passes test cases t_1, t_2 and t_3, but fails test cases t_4 and t_5. Thus we classify passed test cases and failed test cases from the test suite.

Since *Median* has failed test cases, there must be some buggy lines within *Median*. The goal is to find a patch replacement for these buggy lines. Before finding a replacement, SearchRepair needs to locate the buggy lines. SearchRepair derives a coverage-based technique from an existing tool Tarantula (Jones et al., 2002). SearchRepair first calculates the suspiciousness of each line in *Median* by a coverage based technique with the test suite in Table 4.1. Figure 4.2 shows the score of each line's suspiciousness.

Line 16 has the highest score. Due to an implementation limitation that SearchRepair inserts one print statement each before and after buggy lines in order to obtain the input-output specification, SearchRepair is only able to handle buggy lines of complete if-then-else statements and blocks. Inserting print statements around incomplete if-then-else statements and blocks might make either the output specification missing or the input specification missing. Line 16 is a predicate and it is not a complete statement. If SearchRepair inserts print statements before and after line 16, the two print statements are in two exclusive paths of program *Median*, which means SearchRepair cannot extract both input specification and output specification at a given time. Thus SearchRepair considers the sequence of control flow from lines 12 to 17, which are related to that predicate, as a candidate that needs to be replaced. The fault localization technique is detailed in Section 5.2.

Once the bugyy lines are located, SearchRepair executes the program and extracts variable values around the buggy fragments for every test case. We express both input and output state in terms of observed runtime values for the local variables. Consider test case t_1 in Table 4.1, when running *Median* with t_1, the input state is $a = 9, b = 9, c = 9, median = 0$ before line 12, and the output state is $a = 9, b = 9, c = 9, median = 9$ after line 17. We assume that for a passed test case, given the input state, the output state should not change even after the replacement of a correct patch. Test case t_1 is a passed test case, therefore its output state is still $a = 9, b = 9, c = 9, median = 9$ when the buggy lines are replaced with a correct patch. But for failed test case , the output state should change after the replacement of a correct patch. Test case t_4 is a failed test case, therefore after line 16, the output state is $a = 0, b = 2, c = 1, median = 0$, which is incorrect.

SearchRepair then submits the input-output states to a semantic search engine, which converts the input and output states into SMT constraints and searches over a pre-indexed database of code snippets stored as sets of SMT constraints on their behavior. During the searching process, SearchRepair combines each code snippet's corresponding SMT constraints, the mapping SMT constraint between the variables from the code snippet and the variables from the input and output states, and the SMT constraints of the input-output specifications into an SMT formula. In Section 5.4, we discuss how a semantic search engine identifies code snippets that satisfy the SMT formula imposed by the given input-output states. Each returned code snippet is considered as a candidate. SearchRepair then replaces the buggy lines with the candidate and runs the program with the test suite. If the revised program passes all of test cases,

then the candidate is considered as a repair. To replace the buggy code with candidate, SearchRepair modifies the snippets by mapping the variables in the original code context to those in the candidate repair snippet. Consider program *Median* in Figure 4.1, one candidate fix for the buggy lines 12 to 17 is shown in Figure 5.2. The variables' names in candidate fix, x, y, z, m, do not match the variables' names in original buggy code, $a, b, c, median$. The candidate is a correct replacement for repair when we rename it by $x \mapsto a$, $y \mapsto c$, $m \mapsto median$ and $z \mapsto b$. The modified program then passes all of the test cases, as desired. The way to obtain the correct mapping is that SearchRepair enumerates all possible mappings between the variables from input-output states and the variables from the candidate fix. SearchRepair converts each mapping into an SMT constraint and combine it with the input-output SMT constraints and the candidate fix's SMT constraints into an SMT formula. The mapping from the satisfiable SMT formula is considered as a correct mapping.

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5       int a, b, c, median;
6       printf("Please enter 3 numbers separated
            by spaces > ");
7       scanf("%d%d%d", &a, &b, &c);
8       if ((a<=b && a>=c)  (a>=b && a<=c))
9           median = a;
10      else if ((b<=a && b>=c)  (b>=a && b<=c))
11          median = b;
12      else if ((c<=b && a>=c)  (c>=b && a<=c))
13          median = c;
14      printf("%d is the median\n", median);
15      return 0;
16  }
```

Figure 4.1: A student-written program that is intended to print the median of three integers.

```
1   0.0
2   0.0
3   0.0
4   0.0
5   0.0
6   0.5345224838248488
7   0.5345224838248488
8   0.5345224838248488
9   0.0
10  0.7071067811865476
11  0.0
12  1.0
13  0.0
14  0.5345224838248488
15  0.5345224838248488
16  0.0
```

Figure 4.2: Suspiciousness of each line for program in Figure 4.1

Table 4.1: Five test cases for program *Median* in Figure 4.1

	input	output	actual output	pass/fail
t_1	9 9 9	9 is the median	9 is the median	pass
t_2	0 2 3	3 is the median	3 is the median	pass
t_3	0 1 0	0 is the median	0 is the median	pass
t_4	0 2 1	1 is the median	0 is the median	fail
t_5	8 2 6	6 is the median	0 is the median	fail

4.2 Requirements for programs and patch replacement

This section provides some examples that show what kind of buggy programs SearchRepair can fix and what kind of patch replacements SearchRepair can supply for buggy programs.

4.2.1 Requirements for programming language

SearchRepair only supports a portion of the C programming language. The full detail of which part of the C programming language is supported is described in Section 6.3. One step of SearchRepair is to extract the runtime values of local variables to form input and output profiles. For the variables that form input and output profiles, they must be of type integer, float, double, char or char*. Self-defined types are not supported by SearchRepair.

4.2.2 Requirements for test suite

Given a program P with a test suite T, SearchRepair considers P as a correct program if P passed all of the test cases in T. P is considered buggy if there are some test cases in T that P cannot pass. In order to be qualified as a SearchRepair fixable program, the program must have at least one passed test case. The reason why SearchRepair requires a defect to have a passed test case is detailed in Section 5.4. *Median* from Figure 4.1 can be fixed by SearchRepair since *Median* passes test case T_1, T_2, T_3 and fails test case T_4, T_5 in its test suite, which is shown in Table 4.1. However, for *Smallest* in Figure 4.3, even

though it is buggy, SearchRepair is not able to fix it because *Smallest* fails every test case in its test suite shown in Table 4.2.

```
1   /**/
2
3   #include <stdio.h>
4   #include <math.h>
5
6   int main() {
7       int a, b, c, smallest;
8       printf("Please enter 3 numbers separated by spaces > ");
9       scanf("%d%d%d", &a, &b, &c);
10      if (a < b && a < c)
11          smallest = a;
12      else if (b < a && b < c))
13          smallest = b;
14      else if (c < a && c < b))
15          smallest = c;
16      printf("%d is the smallest\n", smallest);
17      return 0;
18  }
```

Figure 4.3: A student-written program that is intended to print the smallest of three integers

4.2.3 Requirements for fault locations and patches

The buggy lines can be anywhere in a method body part. SearchRepair can repair buggy programs regardless of the location of the bug. Buggy lines can be at the beginning of a program or at the end of a program. Buggy lines can be just one single line or multiple lines. Buggy lines can be in a conditional block or inside a loop. *Median* in Figure 4.1 is an example with buggy lines in a conditional block, which exists on line 12. *Syllables* in Figure 4.4 is an example that has buggy lines in a loop. The goal of *Syllables* is to find the number of syllables in a string by counting the vowels, but lines 13 to 16 only do not count the vowel *i*. Table 4.3 shows the two test cases for *Syllables*. Test case t_2 is a failed test case since *Syllables* does not count the vowel *i*. Thus, SearchRepair only uses test case t_1 for searching patches. The input-output specifications, which is the running time values of local variables before line

Table 4.2: Three test cases for program *Smallest* in Figure 4.3

	input	output	actual output	pass/fail
t_1	9 9 10	0 is the smallest	9 is the smallest	fail
t_2	2 2 3	0 is the median	2 is the smallest	fail
t_3	1 1 2	0 is the median	1 is the smallest	fail

13 and after line 16 is shown in Table 4.4. There are two input-output specifications for test case t_1 since there are two iterations for test case t_1. Then SearchRepair uses semantic code search to find patches. Here is a correct patch with a mapping from $a \mapsto string$, $i \mapsto i$ and $count \mapsto num$:

```
1 if (a[i] == 'a'  a[i] == 'e'  a[i] == 'o'  a[i] == 'u'  a[i] == 'y'  a[i] == 'i')
2 {
3     num++;
4 }
```

Table 4.3: Two test cases for program syllables in Figure 4.4

Test	Input	Expected Output	Actual Output	Result
t_1	“ab”	“The number of syllables is 1.”	“The number of syllables is 1.”	passed
t_2	“bi”	“The number of syllables is 1.”	“The number of syllables is 0.”	failed

Table 4.4: The input-output specification for program in 4.4

loop iteration	input specification	output specification
1st iteration	string:“ab”, count:0, i:0	string:“ab”, count:1, i:1
2nd iteration	string:“ab”, count:1, i:1	string:“ab”, count:1, i:1

There are some limitations for patches due to implementation issues. SearchRepair relies on SMT solver to find fixes and SMT solver is only able to deal with predefined types and functions. Patches

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <stdlib.h>
4   #define sizeA 22
5   int main () {
6   char string[sizeA];
7   int i;
8   int count = 0;
9   printf("Please enter a string > ");
10  fgets(string,sizeA-1,stdin);
11
12  for (i = 0; i <= strlen(string); i++){
13      if (string[i] == 'a'  string[i] == 'e'  string[i] == 'o'  string[i] == 'u'  string
            [i] == 'y')
14      {
15          count++;
16      }
17  }
18  printf("The number of syllables is %d.\n", count);
19  return 0;
20  }
```

Figure 4.4: A student-written program that is intended to print the number of syllables in a string

Table 4.5: Data types and lib functions that can be with patch replacements

data types	char, int, float, double, char*, int*, float*, double*, char[]
c lib	strcmp, strlen, strncmp, strcpy, strncpy, toupper, tolower, isdigit islower, isupper, strcat, strncat

can contain variables of specific types listed in Table 4.5. Patches can contain declaration statements, assignment statements, and conditional statements. Patches can not contain return statements. Patches can also contain methods (Table 4.5) from C ctype and cstring library. Patches cannot have loops because SearchRepair does not index loops. Indexing loops is left for future work.

CHAPTER 5. APPROACH

In Section 4.1, we use an illustrative example to briefly describe what happens during every step. In this chapter, we describe the techincal approach of SearchRepair.

5.1 Buggy programs and test suite

Given a program P and a test suite T, SearchRepair first identifies which test cases in T pass and which test cases fail. SearchRepair considers P as a correct program if P passes T completely. SearchRepair considers P as a potential fixable program if P passes T partially. SearchRepair considers P as an unfixable program if P fails T completely.

5.2 Fault localization

We adapt the Tarantula (Jones et al., 2002) technique for fault localization to identify the buggy lines for SearchRepair to try to repair. Tarantula is a well-known example of a class of techniques that implement spectrum-based fault localization. Tarantula uses coverage information provided by a set of passing and failing test cases to compute suspiciousness scores that characterize the likelihood that a given line or piece of code is responsible for the failing test cases. Given a program P and a test suite T, Tarantula executes each test case $t \in T$ and records whether it passed or failed. It then uses the number of passing and failing test cases on which each statement is executed to compute a *suspiciousness* score. The suspiciousness of statement s, is calculated as:

$$suspiciousness(s) = \sqrt{\frac{failed(s)}{total\_failed} \times \frac{failed(s)}{failed(s) + passed(s)}}$$

where $suspiciousness(s)$ denotes the score of the suspiciousness of statement s. The higher the value of $suspiciousness(s)$, the more likely s is buggy. $failed(s)$ denotes the number of failed test cases that

execute s. $total\_failed$ denotes the total number of failed test cases, including both failed test cases that does not execute s and failed test cases that execute s. $passed(s)$ denotes the number of passed test cases that execute s. There are two principles for choosing a formula to calculate suspiciousness:

- The more failed test cases executing statement s, the more suspicious statement s is. This is why $\frac{failed(s)}{total\_failed}$ is a component in the formula.
- The more passed test cases executing statement s, the less suspicious statement s is. This is why our formula has a second component $\frac{failed(s)}{failed(s)+passed(s)}$

The use of this formula can vary in the way the two components are weighted; in SearchRepair, we weight them equally.

Given an initial suspiciousness score for each line in a program, we *calculate* the weight for code fragments that might be replaced by SearchRepair. We do this in the following way:

1. Use the input program and test suite to compute an initial suspiciousness score for each line of code in the program. For the example in Figure 4.1, this initial computation assigns the highest weight to the condition and block on lines 16 and 17 (indeed, the culprit), lower weights to the other predicates in the sequence of if-else clauses, and 0 weights to the declaration and final return.
2. Let *pivot* denote the line with the highest computed suspiciousness score. In our example, this is line 16, which should read `c >=b && c <= a` instead of `c >=b && c >= a`.
3. If *pivot* corresponds to a guard, such as controlling an if block or loop, SearchRepair considers replacing the entire block controlled by the predicate and preceding related control-flow. In Figure 4.1, even though line 16 is suspicious, lines 12–17 are marked for replacement. If *pivot* is not a predicate, SearchRepair takes a window of code around *pivot* as the fragment to be replaced. In the experiments described below, we heuristically set the size of the window to no more than 5 lines.
4. If multiple lines share the biggest suspiciousness, repeat step 3 for each line. SearchRepair will consider each of them as independent candidate sites for repair.

Our experimental results in Chapter 7 show that this approach can accurately locate buggy lines for a non-trivial number of defects in our dataset.

5.3 Obtaining input-output profiles

The fault localization procedure described in the previous subsection identifies candidate code fragments as sites for possible repairs. For each such identified code fragment, the next step is to extract variables, their dynamic values from the source code and its execution on the input test cases. The high-level goal is to collect, for each variable, its **name**, **type**, and *value* both before and after the execution of the candidate buggy code fragment. In the example from Figure 4.1, it requires that we identify the values of `a,b,c,` and `median` before and after the execution of the block comprising lines 12–17. These two collections of values are denoted as the *input state* and *output state* of the program on each execution. We use a straightforward, unoptimized logging procedure to acquire these values in our experiments.

Table 5.1 shows input-output profiles of our *Median* program for the test cases shown in Table 4.1. Since the test cases t_1, t_2, and t_3 pass on the buggy program, the associated input-output profiles are positive examples. For the other test cases, t_4 and t_5, the associated input-output examples are negative specifications since the test cases fail.

Table 5.1: An input-output profile for program *Median* in Figure 4.1 with test suite in Table 4.1.

test	input	input state	output state
t_1	9 9 9	a:9:int b:9:int c:9:int median:0:int	a:9:int b:9:int c:9:int median:9:int
t_2	0 2 3	a:0:int b:2:int c:3:int median:0:int	a:0:int b:2:int c:3:int median:2:int
t_3	0 1 0	a:0:int b:1:int c:0:int median:0:int	a:0:int b:1:int c:0:int median:0:int
t_4	2 0 1	a:0:int b:2:int c:1:int median:0:int	a:0:int b:2:int c:1:int median:0:int
t_5	2 8 6	a:2:int b:8:int c:6:int median:0:int	a:2:int b:8:int c:6:int median:0:int

5.4 Semantic searching with input-output profiles

After obtaining input-output profiles that characterize the potentially-defective code fragment, we use those profiles to search a repository of code snippets for candidate repairs. The repository consists of code snippets encoded as sets of SMT constraints. Chapter 3 described how to build and search over

```
1          if((x <= y && x >= z) || (x >= y && x <=z))
2              m = x;
3          else if((y <= x && y >= z) || (y >= x && y <= z))
4              m = y;
5          else
6              m = z;
```

```
1          path 1: Assert((x <= y && x >= z) || (x >= y && x <=z))
2                  m = x
3          path 2: Assert((y <= x && y >= z) || (y >= x && y <= z))
4                  m = y
5          path 3: Assert((z < x && z > y) || (z < y && z < x))
6                  m = z
```

Figure 5.1: A C snippet and the multiple paths that can be generated from it.

such a repository of Java snippets, as described in previous work (Stolee et al., 2014, 2015). In this section, we will discuss the details of our extension to this technique in order to search over snippets of C code and repair defects.

There are three steps to build a semantic code search repository:

1. Collect candidate source code fragments to encode and store in the database. We encode these fragments at a level of granularity that approximates the level at which we attempt replacements to repair code. We capture entire blocks of statements surrounded by predicates (such as the body of if-then checks or while loops) as well as sequences of statements with a maximum window size of five.

2. Statically enumerate paths from the snippet. To illustrate, consider the C snippet in Figure 5.1 that comprises of a disjunction between three paths. Let p_1, p_2, p_3 denote the three paths. As described in Chapter 3 and Section 5.2, we currently support loops in the buggy code but not in the repair code.

3. Encode every path into SMT constraints as described in prior work (Stolee et al., 2015) and references in Chapter 3. Then, store each path's constraints into the repository as an entry.

The first research implementation challenge was to adapt the Java technique (Stolee et al., 2014) to the C language. This required novel encoding of `char *`. We natively encode a number of string and arithmetic library functions not previously encoded in the Java-based semantic search.

```
1     if((x <= y && x >= z)  (x >= y && x <=z))
2         median = x;
3     else if((y <=  x&& y >= z)  (y >= x && y <= z))
4         median = y;
5     else if((z >= x  z <= y)  (z <=x && z >= y))
6         median = z;
```

Figure 5.2: Candidate fix for lines 12–17 in the program in Figure 4.1.

```
1     if ((a <= b && a >= c)  (a >= b && a <= c)) {
2         median = a;
3     } else if ((b <= a && b >= c)  (b >= a && b <= c)) {
4         median = b;
5     } else if ((c <= b && a <= c)  (c >= b && a <= c)) {
6         median = c;
7     }
```

Figure 5.3: A partial fix to program in Figure 4.1 to replace lines 12–17, but test t_4 still fails

When searching, we define a code snippet s in the repository as a *match*, or potential patch, for a candidate faulty code region if for each pair of *input state* and *output state* corresponding to passed test cases, at least one path in s satisfies the specification. Note that SearchRepair requires a fixable defect to have at least one passed test case. There are two reasons for this. One is that SearchRepair utilizes passed test cases to localize buggy lines in a defect. Section 5.2 describes how SearchRepair localizes buggy lines. The absence of passed test cases makes SearchRepair unable to pinpoint the location of buggy lines. The other reason is that the input-output specifications are false for failed test cases. When using false input-output specifications to search a repository, SearchRepair looks for snippets that do not satisfy the input-output specification. The number of assignments that falsify a propositional logic formula is very big. Thus, SearchRepair produces a lot of false positives when there are only failed test cases. Better utilizing the failed test cases is left for future work.

To illustrate the matching process, consider a single input-output example that could describe code that computes the median of three numbers: {`int i = 2`, `int j = 3`, `int k = 4`} (*input*) and {`int med = 3`} (*output*). Such an input-output query can be translated into constraints as:

$$(\exists i, j, k, med : \texttt{int}) \wedge \\ (i = 2) \wedge (j = 3) \wedge (k = 4) \wedge (med = 3) \qquad (Q_{io})$$

Because we cannot assume consistent variable names, for each considered candidate database fragment, SearchRepair includes constraints encoding all possible mappings between the inputs and outputs of the profile and the candidate. Consider the example in Figure 5.1, whose input variables are `x`, `y`, and `z`. In some instances, including in this example, we can preemptively identify the fragment's output variable, if it is assigned last on every path (`m`, in this example; were this not the case, the number of possible mappings, and thus the mapping constraint, would be larger). The mapping constraints between the example input-output pair and this candidate fragment is:

$$\begin{aligned} &(med = m) \wedge \\ &(((i=x) \wedge (j=y) \wedge (k=z)) \vee ((i=x) \wedge (j=z) \wedge (k=y)) \vee \\ &((i=y) \wedge (j=x) \wedge (k=z)) \vee ((i=y) \wedge (j=z) \wedge (k=x)) \vee \\ &((i=z) \wedge (j=x) \wedge (k=y)) \vee ((i=z) \wedge (j=y) \wedge (k=x))) \end{aligned} \qquad (\mathcal{M}_{io})$$

Thus, for each input-output pair *io*, and for each path *n*, in the fragment, the query to the SMT solver is:

$$\gamma \wedge \phi_n \wedge Q_{io} \wedge \mathcal{M}_{io} \qquad (search)$$

If at least one these queries is satisfiable for a particular fragment, the associated path *n* satisfies the constraints imposed by the input-output pair in question. Only one path per input-output pair needs to be satisfiable for the entire fragment to be considered a candidate patch. When the query is satisfiable, the SMT solver produces a satisfiable model, which provides a suitable binding between input-output and fragment variables consistent with the constraint $\mathcal{M}_{io}$.

Extending this procedure to the multiple examples included in the profile of a candidate buggy region requires the separate encoding of each input-output pair. We define a code fragment as a *match*, or potential patch for a candidate faulty code region, if for each input state and output state pair corresponding to passing test cases, at least one path satisfies the specification (Section 5.5 describes other types of matches, such as partial matches). SearchRepair currently queries the SMT solver once per input-output example in a profile and requires variable mappings to be consistent between each example for a satisfying fragment to be considered a match.

5.5 Evaluating a patch

A candidate patch from the search results is considered a potential *match*. The next step is to use the fragment to replace the buggy lines. When a match is identified during the search, the SMT solver produces a satisfiable model detailing how the variables in the input-output profiles from the program state are mapped to the variables in the patch code. Using these mappings, we perform variable renaming on the patch code to match the scoping and variables in scope in the buggy lines. Then, we re-run the test suite to determine if the code causes all tests to pass.

Based on this execution, a patch is classified as a *true fix*, a *partial fix*, or a *non-fix*.

True fix: a patched program that passes all of the test cases, including both previously passed test cases and previously failed test cases. For example, the snippets in Figure 5.2 is a *true fix* for the program *Median* in our running example from Figure 4.1 since all test cases in Table 4.1 match when lines 12-17 are replaced with the fix code.

Partial fix: a patched program that passes all of the passed test cases, and passes a portion of the failed test cases. For example, the snippet in Figure 5.3 is a *partial fix* to *Median* since it only passes the second failed test case (i.e., t_5) in Table 4.1 and still fails t_4.

Non-fix: a patched program that does not pass any of the failed test cases. For example, the repository may have the same snippet as the buggy lines that passes all of the positive test cases but behaves just as the original buggy program.

The process of searching and evaluating a patch continues while a true fix is not found. As soon as a true fix is found, SearchRepair stops. In the absence of such a match, SearchRepair can continue to iterate until either the list of candidate matches is exhausted or a **true fix** is found.

CHAPTER 6. IMPLEMENTATION

In this chapter, we present details of the implementation for SearchRepair. We design and implement SearchRepair by a strict Model-View-Controller (MVC) model. Figure 6.1 is the MVC structure of how SearchRepair repairs programs. This chapter discusses how we implement the model layer, view layer and controller layer. The view layer and model layer are briefly described in Section 6.1 and Section 6.2, respectively. In Section 6.3, we discuss the controller layer.

6.1 View layer

The view layer interacts with programmers who want to use SearchRepair to repair a defect. The user interface of SearchRepair has three parts: defect, test suite and fix. Programmers input the defect code in the main function of a C source file, input the values of inputs in a text file and input the expected values of outputs in a text file. SearchRepair then take these three files as input and search its repository for fixes. When there is a fix, SearchRepair produces a correct program by replacing the buggy lines with that fix. When a fix is found, SearchRepair stops searching.

6.2 Model layer

The Model layer is a MySQL table, in which each entry consists of source code for a C method, paths, constraints, variables, and formal parameters. This is used for semantic code search. Table 6.1 shows the structure of a table and an example of entries. For example, for the method *big* in Table 6.1, it returns the larger number of two integers. It has two paths, each path is converted to a set of SMT constraints. Its formal parameters and the variables in the method body, `a` and `b`, are also stored.

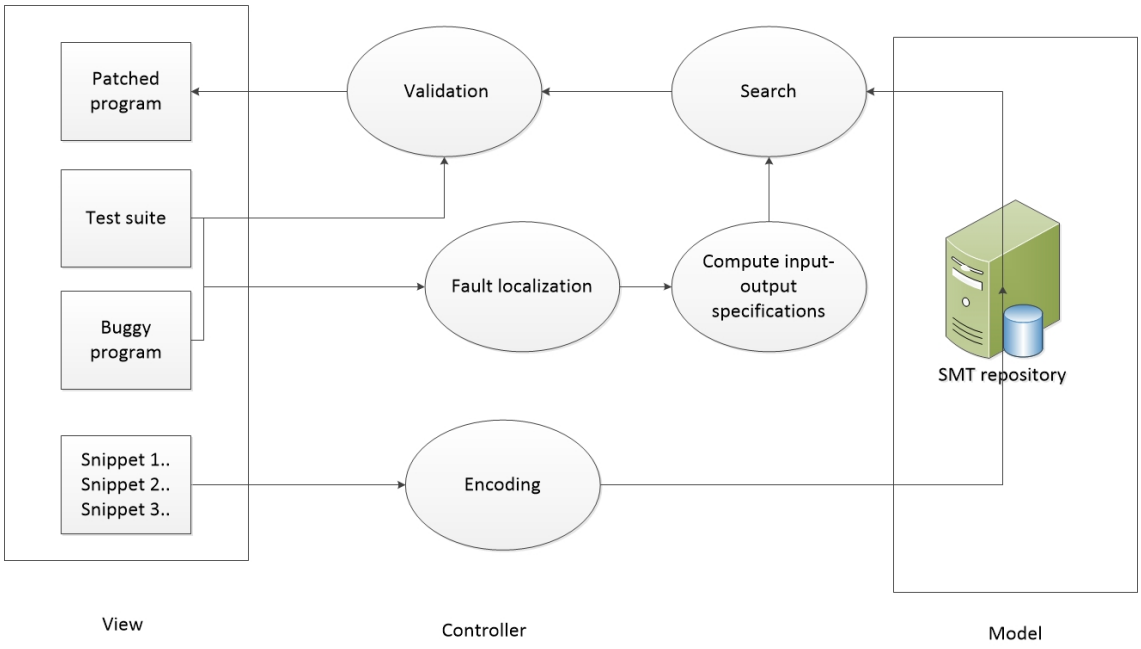

Figure 6.1: Workflow of SearchRepair

Table 6.1: An example of two entries in the repository of SearchRepair

method Name	source code	paths	constraints	variables	formals
big	int big(int a, int b) if (a < b) return b; else return a;	path 1: a < b; return b path 2: a >= b; return a	path 1: assert(< a b) & output = b path 2: assert (>= a b) & output= a	a:int b:int	a:int b:int
small	int small(int a, int b) if (a < b) return a; else return b;	path 1: a < b; return a path 2: a >= b; return b	path 1: assert(< a b) & output = a path 2: assert(>= a b) & output = b	a:int b:int	a:int b:int

6.3 Controller layer

There are five modules in controller layer: encoding, fault localization, computing input-output specifications, search and validation. Encoding is the use of symbolic execution to collect feasible paths for a C multiple path program and converts each path into SMT constraints. SearchRepair incorporates an existing tool developed by Claire Le Gous, which can be downloaded at https://bitbucket.org/clegoues/autobugfix-symex. Searching is used to check the satisfaction of specified input-output

behavior and code snippets in the database. SearchRepair also incorporates Z3 (de Moura and Bjorner, 2008), an existing SMT solver, to solve the matching. Next, we talk about the details of encoding, fault localization and validation implementations.

6.3.1 C encoding implementation

Stolee et al. (2014) provide details on coding a subset of the Java language. Our approach extends their work to the C language. We encode a subset of the C language.

Data types: five built-in types of C are supported (character, integer, double, float, char*(a character pointer)) and one composite data type (char array). Table 6.2 shows the six built-in types' counterparts in the SMT solver. Integers and booleans are built-in types for SMT theories, but strings and characters are not. Stolee (2013) created a way to interpret string by SMT. In their way, each character is assigned an integer value. A string s is defined by its length and by every i^{th} character of the string for $0 \leq i < length(s)$. Two strings are equivalent when both have the same length and contain the same characters. SearchRepair adopts stolee, *et al.*'s way to interpret C strings.

Table 6.2: C built-in types' counterparts in SMT solver

C type	**SMT counterpart**
int a	declare-fun a () Int
char a	declare-fun a () Int
float a	declare-fun a () Real
double a	declare fun a() Real
char* a	declare-fun a () Intpointer
char a[10]	declare-fun a () String

Note that SearchRepair does not distinguish between integer and character. Integer and character are both interpreted as Int. SearchRepair also does not distinguish between float and double, which are both interpreted as Real. These may cause SearchRepair to produce false positives when code shares the same structure with the true fix but with different types of variables. In verification process, SearchRepair replaces buggy lines with the candidate fixes and re-run the program with the test suite. If there is such

Table 6.3: C statements and their counterparts in SMT

statement type	**statement**	**counterpart in SMT solver**
declaration	double a	declare-fun a () Real
assignment	double a = b	assert(= a b)
conditional	if(a < b) c = 1	assert(= c (ite (< a b) 1 0))

a false positive, SearchRepair is not able to compile the program because of type error. Thus these false positives can be eliminated by the patch verification process. The C language has two implicit data types, char*(interpreted as a C string) and boolean type, which are also supported. For type char*, its interpretation depends on the context in order to determine if it should be interpreted as a character pointer or a C string. Type char*(C string) is interpreted in the same way as char array. C does not have a explicit boolean type but interprets implicitly that the statement (a < b) is a boolean value, where true is non-zero and false is zero. SearchRepair uses 1 to denote true and 0 to denote false.

Statements: Declaration statements, assignment statements, and conditional statements are supported by SearchRepair. Return statement is not supported by SearchRepair due to the limitation that SearchRepair inserts print statement after the buggy lines to obtain the input-output specification 6.3.3. Programs exit when executing return statements. If there is a return statement in the buggy lines, the print statement inserted after the buggy lines is not being executed. Table 6.3 lists every statement's counterpart in SMT solver constraints.

Operators: Table 6.4 shows supported C operators and their counterparts in SMT.

Table 6.4: C operators and their counterparts in SMT

C operators	+	-	*	/	%	a++	b–	+=	-=	*=	/=	%=	<	>	==	<=	>=
SMT	+	-	*	/	mod	a+1	b-1	+=	-=	*=	/=	%=	<	>	==	<=	>=

Libraries: SearchRepair supports library functions *isDigit, isLower, isUpper, toUpper, toLower, strLen, strcpy, strncpy, strcmp, strncmp, strcat, strncat*. Here are encoding details of these functions:

Encoding isDigit(int c)

```
1    Function : int value = isDigit(int c)
2    Description: check if character c is digit
```

```
3       Encoding: assert(= value
4                        (ite(and (< c 58) (> c 47)) 1 0)
5                        )
```

Encoding isUpper(int c)

```
1       Function : int value = isUpper(int c)
2       Description: check if character c is in upper case
3       Encoding: assert(= value
4                        (ite(and (< c 91) (> c 64)) 1 0)
5                        )
```

Encoding isLower(int c)

```
1       Function : int value = isLower(int c)
2       Description: check if character c is in lower case
3       Encoding: assert(= value
4                        (ite(and (< c 123) (> c 96)) 1 0)
5                        )
```

Encoding toUpper(int c)

```
1       Function : int value = isLower(int c)
2       Description: transform a character to its upper case
3       Encoding: assert(= value
4                        (ite(and (< c 123) (> c 96)) c-32 c)
5                        )
```

Encoding toLower(int c)

```
1       Function : int value = isLower(int c)
2       Description: transform a character to its lower case
3       Encoding: assert(= value
4                        (ite(and (< c 91) (> c 64)) c+32 c)
5                        )
```

Encoding strlen(char* str)

```
1       Function : int value = strlen(char* str)
2       Description: returns the number of characters in a string
3       Encoding: assert(= value
4                        (length str)
5                        )
```

Encoding strcpy(char* dest, char* source)

```
Function : char* str = strcpy(char* dest, char* source)
Description: copy a string from source to  dest
Encoding: assert(= dest source)
```

Encoding strncpy(char* dest, char* source, int n)

```
Function : char* str = strncpy(char* dest, char* source, int n)
Description: copy n characters from source to  dest
Encoding: assert(= (length dest) (ite (> (length source) n)
                          n (length source)))
          assert(forall (index int) (ite (and
                    (> index 0) (<= index (length dest)))
                    (= (charof source index) (charof dest index)) true))
```

Encoding strcmp(char* str1, char* str2)

```
Function : int str = strcmp(char* str1, char* str2)
Description: compares two strings
Encoding:
          assert(= str (forall (index int) (ite (and
                    (> index 0) (<= index (length str1)))
                    (ite (= (charof str1 index) (charof str2 index)) 0
                    (ite (> (charof str1 index) (charof str2 index)) 1 -1)))
```

Encoding strncmp(char* str1, char* str2, int n)

```
Function : int str = strncmp(char* str1, char* str2, int n)
Description: compares two strings with the first n characters
Encoding: assert (= min (ite ( > (length str1) (length str2))
                    (length str2) (length str1)))
          assert(= str (forall (index int) (ite (and
                    (> index 0) (<= index (ite (> n min) min n) ))
                    (ite (= (charof str1 index) (charof str2 index)) 0
                    (ite (> (charof str1 index) (charof str2 index)) 1 -1)))
```

Encoding strcat(char* str1, char* str2)

```
Function : char* dest = strcat(char* str1, char* str2)
Description: concatenate one string to another
Encoding: assert (= (length dest) (+ (length str1) (length str2)))
          assert(forall (index int) (ite (and
                    (> index (length str1)) (<= index (length dest)) )
                    (= (charof dest index) (charof str2 (- index (length str1))) true)
                         )
          assert(forall (index int) (ite (and
```

```
8                          (<= index (length str1)) (> index 0))
9                          (= (charof dest index) (charof str1 index) ) true))
```

Encoding strncat(char* str1, char* str2, int n)

```
1    Function : char* dest = strcat(char* str1, char* str2, int n)
2    Description: concatenate n characters from one string to another string
3    Encoding: assert (= min (ite ( > (length str2) n)
4                                 n (length str2)))
5              assert (= (length dest) (+ (length str1) min))
6              assert(forall (index int) (ite (and
7                        (> index (length str1)) (<= index (length dest))
8                        (= (charof dest index) (charof str2 (- index (length dest))) ) )
9              assert(forall (index int) (ite (and
10                       (<= index (length str1)) (> index 0) )
11                       (= (charof dest index) (charof str1 index) ) true))
```

6.3.2 Fault localization implementation

In Chapter 5, we discussed the approach based on Tarantula (Jones et al., 2002) to localize faults in a buggy program. Here, we are going to present the technique details. SearchRepair uses a tool, *Gcov*, to calculate the number of executions of each statement in a C program. *Gcov* is distributed within *GCC*. Consider the C source file in Figure 6.2 with two inputs in Table 6.5. The three steps that are used to calculate execution times of each statement are

1. Run command $gcc - fprofile - arcs - ftest - coveragetest.c$ to compile test.c file.

2. Run command $./test32$"and"$./test23$ to execute with inputs.

3. Run command $gcovtest.c$ to get the execution times. The result is shown in Figure 6.3. The numbers at the start of each line is the number of times that line is being executed. If there is no number at the start of that line, it means that line is trivial or not executed. Lines with the symbol '-' are descriptions of Gcov result file or the lines that are not within the method body. Lines with '#' in Figure 6.3 indicate the lines associated with them are within the method body but are not executed during running time.

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   int main(int argv, char** args)
4   {
5       int b = atoi(args[1]);
6       int c = atoi(args[2]);
7       if(b < 2){
8            c = c + 3;
9       }
10      printf("%d\t%d\n",  b, c);
11      return 1;
12  }
```

Figure 6.2: A C program that increases the second input by 3 if the first input is less than 2

Table 6.5: Two inputs for program in Figure 6.2

Input 1	Input 2
1, 2	2, 3

6.3.3 Computing input-output specifications, searching and validation

After fault localization, SearchRepair inserts two print statements into the buggy program. One is inserted before the buggy lines. The other is inserted after the buggy lines. Each print statement prints the runtime values of existing local variables of the buggy program. If there are two local variables, *var*1 of integer type and *var*2 of double type, when a buggy program executes to its buggy lines. SearchRepair then inserts the following statements before the buggy lines:

```
print("input: %d, %f", var1, var2);
```

If there is any new local variable being defined in buggy lines, SearchRepair adds that local variable into the print statement inserted after the buggy lines. For example, variable *var*3 of char type is defined in the buggy lines. The print statement inserted after the buggy lines is:

```
print("input: %d, %f, %c", var1, var2, var3);
```

SearchRepair then runs the buggy program with the test suite and captures the prints in order to obtain the input-output specifications.

```
1          -:    0:Source:test.c
2          -:    0:Graph:test.gcno
3          -:    0:Data:test.gcda
4          -:    0:Runs:2
5          -:    0:Programs:1
6          -:    1:#include<stdio.h>
7          -:    2:#include<stdlib.h>
8          -:    3:int main(int argv, char** args)
9          -:    4:{
10         -:    5:    //  int a = atoi(args[0]);
11         2:    6:    int b = atoi(args[1]);
12         2:    7:    int c = atoi(args[2]);
13         2:    8:    if(b < 2){
14     #####:    9:         b = 3;
15     #####:   10:    }
16         2:   11:    printf("%d\t%d\n",  b, c);
17         2:   12:    return 1;
18         -:   13:}
```

Figure 6.3: The sum of execution times for each line of the program in Figure 6.2 on the two inputs in Table 6.5

SearchRepair relies an existing tool Z3 de Moura and Bjorner (2008) to conduct semantic code search. SearchRepair converts input-output specification into SMT constraints. Section 3.3 describes how to convert input-output specifications into SMT constraints. SearchRepair also constructs mapping constraints by mapping the variables from input-output specifications and the variables from the code snippet for the repository. SearchRepair supplies input-output SMT constraints, one mapping constraint and the snippet SMT constraints to Z3 and lets Z3 decide if there is a satisfiable assignment.

If a satisfiable assignment is found, SearchRepair renames the variables in the snippet by the mapping constraint from the searching process. The renamed code snippet is called a patch in this thesis. Then SearchRepair replaces the buggy lines with this patch, recompiles the buggy program and re-runs the buggy program with the test suite. If all test cases are passed, that patch is a correct fix.

CHAPTER 7. EVALUATION

This chapter describes SearchRepair's evaluation, including a comparison of effectiveness and quality with respect to three prior tools, GenProg (Weimer et al., 2009), TSPRepair (Qi et al., 2013), and AE (Weimer et al., 2013). We explore five questions:

1. **RQ1: Is SearchRepair effective enough that it can fix a non-trival number of defects?**
2. **RQ2: Is SearchRepair able to discover correct patches from open source projects?**
3. **RQ3: Does SearchRepair produce quality patches that fixes it found can pass an independent test suite?**
4. **RQ4: Which test suite is more suitable for SearchRepair, human written or computer generated?**
5. **RQ5: Does SearchRepair fix defects fast enough that it can find fixes within minutes?**

Section 7.1 describes our experimental set up. Sections 7.2 evaluates SearchRepair's effectiveness at producing repairs. Section 7.3 explores SearchRepair's feasibility. Section 7.4 evaluates the quality of the repairs produced by SearchRepair. Section 7.5 explores how different test suites affect SearchRepair. Section 7.6 shows the results of the performance of SearchRepair. Section 7.7 discusses the threats to validity. Sectioon 7.8 is a summary of this chapter.

7.1 Experimental setup

We base our evaluation on *IntroClass*, a benchmark set constructed of student-written C programs containing defects intended for evaluating automatic program repair research (Le Goues et al., 2015). *IntroClass* is available for download (introclass, 2008). This benchmark dataset is used to address the

research questions: RQ1 to RQ5. How *IntroClass* benchmark was generated is detailed in Section 3.1. *IntroClass* provides 998 versions of programs submitted by students for six small C programming assignments in an introductory undergraduate course.

We built three repositories of code snippets for SearchRepair. These three repositories are used by semantic code search to find patches for buggy programs. The source code used for building each repository is different. One repository, denoted by Rsp(linux), contains only source code extracted from the linux kernel. The second repository contains only source code from *IntroClass*, which is denoted by Rsp(Intro). We use a scraper to scrape code from all solutions in *IntroClass*, including those correct solutions. If the scraper happens to capture the code from the correct solutions, the repository may contain code from the correct solutions in *IntroClass*. Note that both *IntroClass* and linux kernel have a large size, the scraper does not scrape every line of code. The scraper captures several lines of code from each file in *IntroClass* and linux kernel. Recall that we design SearchRepair based on the assumption that given a defect in a project, there are correct re-implementations in other projects for that defect. We want to simulate that situation. In *IntroClass*, when a student submitted a buggy solution, another student may have submitted a solution that has the correct code to fix that defect. Here is how we design the third repository, denoted by Rsp(others). Rsp(others)'s source code also comes from *IntroClass*. Rsp(others) is different depending on which students program is being fixed. For example, say we have three students, Alex, Barb, and Chris. If youre fixing one of Alexs program, then Rsp(others) contains snippets from the buggy programs from Barb and Chris. Similarly if Barbs program is being fixed, Rsp(others) contains snippets from buggy versions of Alex and Chriss programs. Table 7.1 shows the details of each repository.

Table 7.1: The number of code snippets and the number of lines, predicates, and parameters per snippet in each repository

Repository	Snippets	Lines per snippet	Predicates per snippet	Parameters per snippet
Rsp(linux)	240	1.8	0.1	3
Rsp(Intro)	130	3	0.7	4.2
Rsp(others)	160	3.1	0.67	4

7.2 Repair effectiveness

This section addresses research question *RQ1: is SearchRepair effective enough that it can fix a non-trivial number of defects?* Table 7.2 shows how effective each of the four repair techniques, SearchRepair, GenProg, TSPRepair, and AE, are at producing patches that pass all of the tests supplied to the repair technique. (Section 7.4 evaluates the quality of the patches.) SearchRepair repairs 187 (22.3%) of the 845 defective student programs, compared to GenProg's 258 (30.4%), TSPRepair's 235 (30.3%), and AE's 142 (15.0%) when using the KLEE test suite. When using the instructor test suite, SearchRepair repairs 150 (19.2%) of the 778 defective student programs, compared to GenProg's 272 (34.4%), TSPRepair's 235 (32.3%), and AE's 128 (15.9%).

Table 7.2: Number of defects fixed by each technique with the KLEE test suite. The repository is Rsp(others). The total column specifies the total number of defects, whereas the total row specifies the total number of repaired defects.

program	SearchRepair	AE	GenProg	TSPRepair	total
median	90	15	72	34	145
syllables	2	5	5	7	115
smallest	71	88	113	113	113
grade	5	2	2	3	224
checksum	19	1	3	1	49
digits	0	29	63	63	199
total	187	142	258	235	845

We can see that SearchRepair did well on `median` and `smallest`, fixed several defects on `grade`, and produced some but not many repairs on `syllables`. There are two assignments for which SearchRepair was unable to produce any repairs, `checksum` and `digits`. The `checksum` assignment is challenging for all the techniques except GenProg. GenProg has the ability to combine multiple repairs over the course of an evolutionary search, and `checksum` functionality may require multi-edit repairs beyond what can be provided even at SearchRepair's higher granularity. However, Section 7.3 describes how by using a

Table 7.3: Number of defects fixed by each technique with the instructor test suite. The repository is Rsp(others). The total column specifies the total number of defects, whereas the total row specifies the total number of repaired defects.

program	SearchRepair	AE	GenProg	TSPRepair	total
median	68	58	108	93	168
syllables	4	11	19	14	109
smallest	73	71	120	119	155
grade	5	2	2	2	226
checksum	0	0	8	0	29
digits	0	17	30	19	91
total	150	159	287	247	778

database of code snippets from the Linux kernel, SearchRepair was able to repair 18 of the `checksum` defects. The SearchRepair performed poorly on the `digits` assignment because this assignment requires modeling of I/O operations beyond the capability of our constraint encoder. Extending the semantic search technique to encode and model such operations increases SearchRepair's ability to handle a wider array of program constructs and thus, defects. We are encouraged by SearchRepair's success given the subset of C language constructs and operations it currently supports.

SearchRepair is complementary to the other repair techniques. When using the instructor test suite, it can repair 20 defects that the other three techniques do not repair. Figure 7.1 shows the Venn diagram describing the breakdown of which techniques repaired which defects, when using the instructor test suite. There are 310 total unique defects the tools were able to repair. Of these, 20 (6.5%) are unique to SearchRepair, 160 (51.6%) can be repaired by at least one other technique but not by SearchRepair, and 130 (41.9%) can be repaired by SearchRepair and at least one other technique. These results suggest that SearchRepair may be interestingly orthogonal to these other previously-proposed generate-and-validate techniques. For example, SearchRepair and AE do not repair any defects in common. This suggests that, although SearchRepair does not repair *more* defects than the previous tools, it repairs *different* defects.

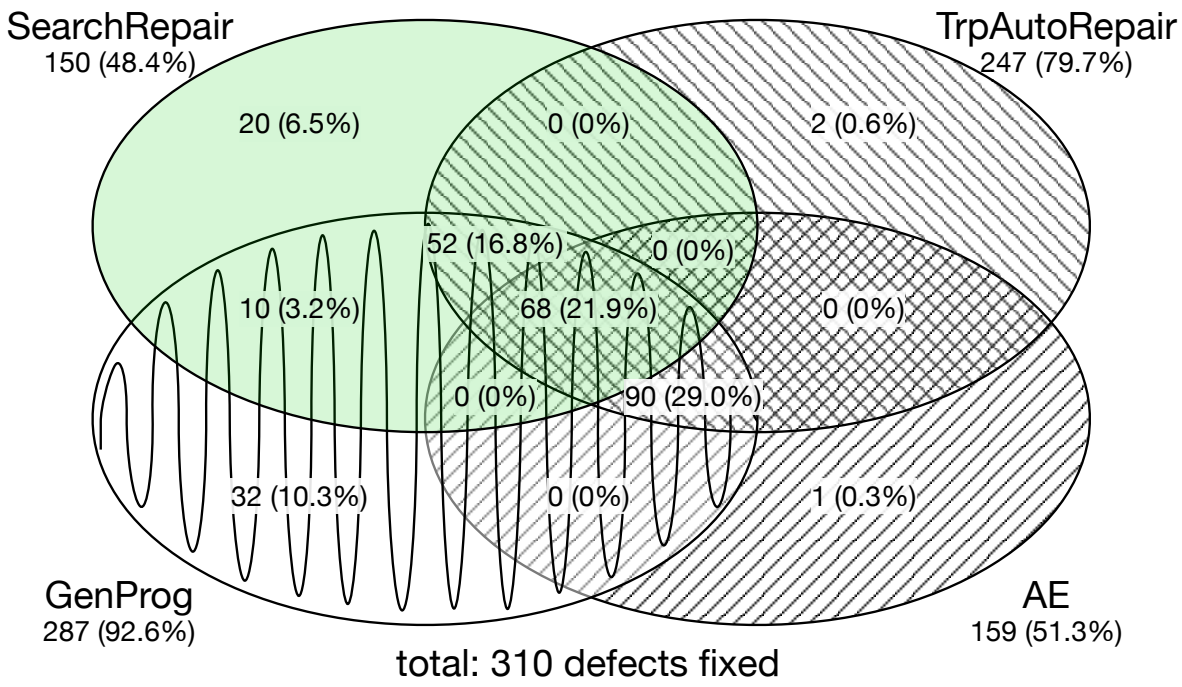

Figure 7.1: SearchRepair is the only tool that can repair 20 (6.5%) of the 310 defects fixed by the four repair techniques. The other three repair tools can together repair 160 (51.6%) defects that SearchRepair cannot. The remaining 130 (41.9%) of the defects can be repaired by SearchRepair and at least one other tool. And the repository used by SearchRepair is Rsp(others). The test suite used by SearchRepair is the instructor test suite.(Not shown in the diagram is that 35 (11.3%) of the defects can be repaired by both GenProg and TSPRepair, and that 0 (0.0%) of the defects can be repaired by both SearchRepair and AE.)

Table 7.4 shows the number of defects fixed only by SearchRepair when using the KLEE test suite. It shows that there are 339 total unique defects SearchRepair and other three techniques can fix, of which 58 of them can only be fixed by SearchRepair.

Unlike previous techniques, SearchRepair can also automatically identify *partial fixes* that address some but not all of the defective behavior. Some of our dataset programs only failed one test case, rendering partial fixes moot. As we can see from Table 7.5 and Table 7.6.

Table 7.4: Number of defects fixed by SearchRepair only and by all of GenProg, TSPRepair, AE and SearchRepair. The repository used by SearchRepair is Rsp(others). The test suite is the KLEE test suite.

Program	**SearchRepair only**	**Fixed by ALL**
median	35	116
syllables	0	6
smallest	0	113
grade	5	8
checksum	18	33
digits	0	63
total	58	339

Table 7.5: The number of *complete fix*, *partial fix*, and *no fix*, across all program types when using the instructor test suite

	Total	Complete Fix	Partial Fix	No Fix
median	168	68	0	100
syllables	109	4	0	105
smallest	155	73	0	82
grade	226	5	4	217
checksum	29	0	0	19
digits	91	0	0	91

7.3 Feasibility of SearchRepair

This section addresses *RQ2: is SearchRepair able to discover correct patches from open source projects?* Table 7.7 shows the results of running SearchRepair on *IntroClass* with these three repositories using the KLEE test suite. When using Rsp(linux), SearchRepair is able to find 18 fixes for the overall 840 defects. It shows that SearchRepair is able to find the correct implementation for bug from open source projects. We also notice that the results for Rsp(Intro) and Rsp(others) are more promising because 187

Table 7.6: The number of *complete fix*, *partial fix*, and *no fix*, across all program types when using the KLEE test suite

	Total	Complete fix	Partial fix	No fix
median	145	90	2	53
syllables	115	2	0	113
smallest	113	71	0	42
grade	222	5	4	213
checksum	49	18	0	31
digits	196	0	0	196

out of 840 defects can be fixed. This implies that the reason why the fix rate of SearchRepair searching Rsp(linux) is so small is because the linux kernel might not contain a lot of correct implementations for defects in *IntroClass*. Another reason might be that SearchRepair only scrapes a small portion of linux source code. SearchRepair might not scrape the implementations into Rsp(linux). The results for Rsp(Intro) and Rsp(others) are the same thought these two repositories are different. This might be due to the high similarity among the submissions in *IntroClass*. Both repositories contains similar implementions for defects in *IntroClass*. Thus, we posit that if SearchRepair is able to scrape enough volume of source code from open source projects, its fix rate would go up.

7.4 Repair quality

This section addresses research question *RQ3: does SearchRepair produce quality patches that fixes it found can pass an independent test suite?* Although it is important to be able to produce a repair, it is just as important to know how good a repair is. We want to know given an independent test suite, how many test cases can be passed by that repair. A low quality repair would frustrate users to not to adopt that technique. We introduce two measures to evaluate the quality of a new repair technique. One is that given an independent test suite, how many repairs can pass all of the test cases in the independent test suite. The other measure is that given an independent test suite, what is the average number of test cases of the test suite a repair can pass.

Table 7.7: Number of defects fixed by SearchRepair using different repositories, when using the KLEE test suite

program	Rsp(linux)	Rsp(Intro)	Rsp(others)
median	0(145)	90(145)	90(145)
syllable	0(115)	2(115)	2(115)
smallest	0(113)	71(113)	71(113)
grade	0(224)	5(224)	5(224)
checksum	18(49)	18(49)	18(49)
digits	0(199)	0(199)	0(199)
total	18(845)	187(845)	187(845)

We conduct two experiments, one which uses the instructor test suite to find repairs and uses the KLEE test suite as independent test suite. We then use the second, independent test suite that is not used by the repair process to compare the quality of the repairs. If a repair passes more of the independent tests, then it generalizes better to the full specification of the program, and is thus of higher quality. Table 7.8 shows the results, from which we can see SearchRepair has a very high quality on *Median*, *Smallest* and *Grade*. 146 of the 150 repairs can completely pass the independent test suite.

Table 7.8: Overfitting evaluation when using the instructor test suite

	versions pass all of extra	average passing test cases
median	68(68)	6(6)
smallest	73(73)	8(8)
checksum	0(0)	0 (6)
syllables	0(4)	0 (9)
digits	0(0)	0 (6)
grade	5(5)	9 (9)
total	146/150	

The other experiment is using the KLEE test suite to find repairs and the instructor test suite as the independent test suite. Table 7.9 shows the results from which we can see SearchRepair has a very high quality on *Median*, *Smallest* and *Grade*. 120 of the 187 repairs can completely pass the independent test suite.

Table 7.9: Overfitting evaluation when using the KLEE test suite

	versions pass all of extra	average passing test cases
median	86(90)	6.9(7)
smallest	11(71)	2.0(8)
checksum	18(18)	6 (6)
syllables	0(2)	0 (9)
digits	0(0)	0 (6)
grade	5(5)	9 (9)
total	120/187	

7.5 The impact of test suite

The section addresses *RQ4: which test suite is more suitable for SearchRepair, human written or computer generated? IntroClass* has two types of test suites. One type of test suite is written by humans, based on the specification of program assignments. The other type of test suite is generated by computers based on the KLEE algorithm to achieve a full coverage of every branch of program. In this section, we compare the performance of SearchRepair when using different test suites. The repository we use is Rsp(others) since Rsp(Linux) does not contain enough correct reimplementations and Rsp(Intro) provides the same results as Rsp(others) does.

Table 7.3 shows the number of defects fixed by SearchRepair, GenProg, TSPRepair and AE based on the instructor test suite. Table 7.2 shows the number of defects fixed by SearchRepair, GenProg, TSPRepair and AE based on the KLEE test suite with the repository Rsp(others). SearchRepair can outperform GenProg on *Median* and *checksum* assignments when using the KLEE test suite. When using the instructor test suite, SearchRepair fixes less defects than GenProg does on every program

assignment. Note that there is a big discrepancy for every technique between the number of defect fixes on the KLEE test suite and the instructor test suite. For example, GenProg can fix 24 more defects using the instructor test suite than using the KLEE test suite. However, we notice that SearchRepair can fix 32 more defects using the KLEE test suite than using the instructor test suite. From the result, we can see that then number of fixes found by SearchRepair may vary on different test suites. However, *IntroClass* only provides two test suites. An experiment with large number of test suites is future work for us.

7.6 Repair speed

The section addresses research question *RQ5: does SearchRepair fix defects fast enough that it can find fixes within minutes?* It is important to be able to produce a good repair, however, it is as important to know how fast a repair can be generated. We want to know that given an independent test suite and a set of buggy programs, the average time of finding a fix for each buggy program. A low speed of finding a fix would frustrate users to not use that technique. We conduct two experiments, one is to use the instructor test suite as the independent test suite. The other one is to use the KLEE test suite as the independent test suite. Both experiments are run on Mac OS with a CPU of 2.0GHz.

Table 7.10: Running time when using the instructor test suite. The repository is Rsp(others)

	number of defects	fixes found	overall time	avg time per fix	avg time per defect
median	168	68	3.1h	2.8min	1.1min
smallest	155	73	5.6h	4.6min	2.2min
checksum	29	0	8.7h	NA	18min
syllables	109	4	13h	3.3h	7.7min
digits	91	0	0.1h	NA	0.01min
grade	226	5	2.5h	30min	0.7min
total	778	150	33.0h	24min	2.3min

From results (Table 7.10 and Table 7.11), we see that it takes 2 to 3 minutes for SearchRepair to process a defect. The reason why SearchRepair processes *digits* so fast is that the fault localization technique does not apply to *digits* programs. Thus, SearchRepair does not conduct searching process and

Table 7.11: Running time when using the KLEE test suite. The repository is Rsp(others)

	number of defects	fixes found	overall time	avg time per fix	avg time per defect
median	145	90	3.3h	2.2min	1.4min
smallest	113	71	5.6h	3.2min	4.3min
checksum	49	18	13.3h	NA	14min
syllables	115	4	14h	NA	7.4min
digits	199	0	0.1h	NA	0.01min
grade	224	5	2.5h	30min	0.7min
total	840	148	40.8h	15min	2.9min

verification process for *digits* programs. Because *checksum* and *syllables* programs have loop inside, it takes much longer for SearchRepair to process them. Section 6.3 in Chapter 6 shows that SearchRepair generates might more than input-output specifications with one input for programs with loops. Thus, the searching process needs to match more input-output specifications for programs with loops. This decreases the searching speed of SearchRepair on programs with loops.

7.7 Threats to validity

IntroClass is composed of short, simple programs and our results may not generalize to more complex programs. We mitigate this threat to validity by evaluating on a large number of such programs, tested by carefully-considered test suites, programmed by actual novices making real mistakes. The experimental setting is appropriate for the constraints of the semantic search engine research prototype for C. The implementation of new constraint encodings for C language constructs is labor-intensive, and we are confident that adding new such constructs will grow SearchRepair's expressive power. For example, approximation and unrolling can both extend support to more powerfully encoding loops.

Code search is limited to strings, integers, booleans, floats, and chars. Thus, the constraints that we use to encode candidate repairs is also limited to these data types.

For most of our experiments, the repository is constructed of closely related programs, in the interest of supporting a scalable evaluation. While we removed a given student's correct answer from

the repository (to avoid giving the search the correct answer), the artificial nature of the repository construction is a threat to the generality of our results. Our success on the one case study assignment using snippets scraped from the Linux kernel supports our hypothesis that the approach can generalize to broader datasets.

The results of SearchRepair varies as the test suite changes. SearchRepair is able to find 32 more fixes when using the KLEE test suite than using the instructor test suite. The impact of test suites need more research.

We assume the test cases are sound and complete. That is, we assume the tests fail when there is a bug and pass when there is not one. While in general this assumption is flawed, for the scope of the evaluation, this assumption is reasonable. Applying SearchRepair to a broader context will likely require us to consider partial fixes when complete fixes cannot be found. In that case, the partial fix may be the product of the test suite, and not the code repository being searched over.

7.8 Summary

In this chapter, we explore five research questions of SearchRepair.

1. **Effectiveness: Is SearchRepair effective enough that it can fix a non-trivial number of defects?** Our results show that SearchRepair is able to fix 19.2% of the overall defects when using the instructor test suite, which is better than the fix rate 15.9% of an existing technique AE when using the instructor test suite.

2. **Is SearchRepair able to discover correct implementations from open source project?** When using the linux kernel as the source project and the KLEE test suite, SearchRepair is able to find 18 fixes. This number is not large, but it does implies SearchRepair has the potential to discover correct implementations from open source project.

3. **Does SearchRepair has good repair quality that fixes it found can pass an independent test suite?**. Our results show that SearchRepair does provides repair of good quality: 98% of the found fixes can completely pass an independent test suite.

4. **Which test suite is more suitable for SearchRepair, human written or computer generated?** Our experiments show that the the computer generated test suite by KLEE algorithm is better than the human written test suite. When using the KLEE test suite, SearchRepair is able to fix 187 out of 845 bugs, while SearchRepair is able to fix 150 out of 778 bugs when using the instructor test suite. But *IntroClass* only provides two test suites, more work should be done with a large number of test suites. This is future work.

5. **Does SearchRepair fix defects fast enough that it can find fixes within minutes?** Our results show that the time for SearchRepair to find a fix varies. For some programs without loops, it takes several minutes to find a fix. For other programs with loops, it takes tens of minutes to find a fix. How to optimize the speed for finding fix for defects with loops is left for future work.

6. **Does SearchRepair fix defects fast enough that it can find fixes within minutes?** Our results show that the time for SearchRepair to find a fix varies. For some programs, it takes several minutes to find a fix. However, for other programs, it takes tens of minutes to find a fix.

CHAPTER 8. RELATED WORK

Code redundancy. Prior work has found that much of software is both syntactically and semantically redundant (Barr et al., 2014; Carzaniga et al., 2010, 2013; Gabel and Su, 2010). A study of 6,000 software projects (over 420 million lines of code) found that large portions of most software projects are syntactically redundant (Gabel and Su, 2010). Semantically, many methods can be reconstructed by composing other methods in the same project (Carzaniga et al., 2010, 2013). These findings are consistent with researchers' observations of code clones (Kapser and Godfrey, 2008). Intuitively, software projects repeatedly reuse the same common building block data structures and methods as other projects, and often reimplement this functionality. This suggests that if a project contains a method with a bug, other software projects are likely to implement a bug-free version of the same or similar functionality that may be used to produce a repair.

Semantic Code Search. Recently keyword-based searches have begun to incorporate semantic information for finding working code examples from the Web (Keivanloo et al., 2014) or reformulating queries for concept localization (Haiduc et al., 2013). The search approach we use for program repair depends on a more structured specification. Prior semantic code search has used formal specifications (Ghezzi and Mocci, 2010; Penix and Alexander, 1999; Zaremski and Wing, 1997) and test cases (Podgurski and Pierce, 1993; Reiss, 2009). Formal specifications allow precise sound matching but must be written by hand, which is difficult and error-prone. Test cases are more lightweight but, prior to our work, required the code to be executed and could not identify partial matches.

Code Synthesis. Code can be synthesized using input-output examples, written in a domain-specific language (Gulwani et al., 2011), using predefined components (Jha et al., 2010), or based on a high-level behavioral description (Autili et al., 2007). Recent approaches use context from a debugger to show where in a program synthesis should occur (Galenson et al., 2014). While effective in that domain, synthesis-based approaches are limited by the solver's ability to enumerate and test all possible

combinations of program constructs. In our use of solvers for semantic search, we instead encode, search over, and return existing code.

Program repair. Automated program repair is concerned with automatically bringing an implementation more in line with its specification, typically by producing a patch that addresses a defect as exposed by a specification or a test case. Interest in this field has expanded substantially over the past decade to include at least twenty projects since 2009 that involve some form of repair (e.g., AE (Weimer et al., 2013), AFix (Jin et al., 2011), ARC (Bradbury and Jalbert, 2010), Arcuri and Yao (Arcuri and Yao, 2008), ARMOR (Carzaniga et al., 2013), and AutoFix-E (Wei et al., 2010; Pei et al., 2014), Axis (Liu and Zhang, 2012), BugFix (Jeffrey et al., 2009), CASC (Wilkerson et al., 2012), ClearView (Perkins et al., 2009), Coker and Hafiz (Coker and Hafiz, 2013), Debroy and Wong (W. Eric and Wong, 2010), FINCH (Orlov and Sipper, 2011), GenProg (Le Goues et al., 2012), Gopinath et al. (Gopinath et al., 2011), Jolt (Carbin et al., 2011), Juzi (Elkarablieh and Khurshid, 2008), PACHIKA (Dallmeier et al., 2009), PAR (Kim et al., 2013), SemFix (Nguyen et al., 2013), Sidiroglou and Keromytis (Sidiroglou and Keromytis, 2005), TSPRepair (Qi et al., 2013), etc.). Chapter 3 has discussed the differences between generate-and-validate and correct-by-construction repair. The approach behind SearchRepair bridges the gap between these two repair approaches, in a similar vein as SemFix (Nguyen et al., 2013), which uses learned constraints to guide component-based synthesis of repair code. SearchRepair is more general in the types of defects it can repair than SemFix, which specifically targets defective predicates and assignments. SearchRepair can also use databases of human-written code, broadening the granularity of the changes that can be found and applied to beyond one line. This may impact readability, maintainability, or generalizability of the resulting code, as we observed in our overfitting metrics.

CHAPTER 9. CONCLUSION AND FUTURE WORK

As the volume of modern open source projects increases, it is possible that a piece of code with a defect has been reimplemented correctly elsewhere. We take advantage of this observation by using advances in semantic code search — searching for code based on a behavioral specification as opposed to a keyword description — to automatically repair defective programs. We implemented our approach in SearchRepair, a technique that uses static analysis to build a searchable database of open-source code snippets that describes snippet behavior as a set of SMT constraints, and dynamic analysis to identify candidate faulty regions in a program and construct characteristic input-output behavior profiles. SearchRepair uses an SMT constraint solver to search over the database of code snippets for potential repairs, maps candidate repair fragments to a buggy context, and validates the candidate repair using test cases.

We evaluate SearchRepair on a program repair benchmark *IntroClass*, which consists of 998 defects. Our evaluation proves that SearchRepair is feasible and it can find 20 implementations from the linux kernel for defects in *IntroClass*. Our evaluation shows that SearchRepair is able to fix 145 out of 843 defects when using the instructor test suite. SearchRepair is also able to fix 187 out of 840 test defects when using the KLEE test suite. SearchRepair can fix more defects than AE which proves that SearchRepair is more effective.

When using the instructor test suite to find repairs, 142 of the 145 repairs can completely pass the KLEE test suite. When using the KLEE test suite to find repairs, 120 of the 187 repairs can completely pass the instructor test suite, which suggests that SearchRepair has a good quality.

SearchRepair is completely automated. We can use the KLEE algorithm to generate test suites. Fault localization, searching,and patch generation are all performed by programs automatically.

Overall, SearchRepair's results strongly suggest more research is warranted in semantic-search-based repair and that such approaches may produce patches that drastically outperform its counterpart techniques in terms of generalizing to program specifications.

9.1 Future work

Though our experiments show that SearchRepair does fix some buggy programs in the benchmark set *IntroClass*, SearchRepair is still a preliminary technique that needs improvement.

1. **Being more generic**. As discussed in Section 7.7 and Chapter 6, SearchRepair only supports a small portion of the C programming language. We need to extend SearchRepair to support more data types, especially user defined data types. We also need to extend SearchRepair to support indexing loops. We also need to extend SearchRepair to support multiple programming languages, like Java and Python.

2. **Be faster**. Our experiments show that it takes SearchRepair several minutes to process a defect, which in real life may be intolerable to users. We need to evaluate how well users can tolerate such a delay and find ways to optimize the speed.

3. **Improve fault localization technique**. We use a coverage based technique to pinpoint buggy lines, which has a poor performance when the buggy lines are not within a conditional block. We need to find a better fault localization technique that does not solely rely on coverage.

BIBLIOGRAPHY

Anvik, J., Hiew, L., and Murphy, G. C. (2005). Coping with an open bug repository. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39.

Arcuri, A. and Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168.

Autili, M., Inverardi, P., Navarra, A., and Tivoli, M. (2007). SYNTHESIS: A tool for automatically assembling correct and distributed component-based systems. In *International Conference on Software Engineering*, pages 784–787.

Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. (2014). The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China.

Bradbury, J. S. and Jalbert, K. (2010). Automatic repair of concurrency bugs. In *International Symposium on Search Based Software Engineering - Fast Abstracts*, pages 1–2.

Britton, T., Jeng, L., Carver, G., Cheak, P., and Katzenellenbogen, T. (2013). Reversible debugging software. *University of Cambridge*.

Carbin, M., Misailovic, S., Kling, M., and Rinard, M. C. (2011). Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*.

Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., and Pezzè, M. (2013). Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 782–791, San Francisco, CA, USA.

Carzaniga, A., Gorla, A., Perino, N., and Pezzè, M. (2010). Automatic workarounds for web applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 237–246, Santa Fe, New Mexico, USA.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215 – 222.

Coker, Z. and Hafiz, M. (2013). Program transformations to fix C integers. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 792–801, San Francisco, CA, USA.

Dallmeier, V., Zeller, A., and Meyer, B. (2009). Generating fixes from object behavior anomalies. In *Automated Software Engineering*, pages 550–554.

de Moura, L. M. and Bjorner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340.

Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., and Rinard, M. (2006a). Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 233–244. ACM.

Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., and Rinard, M. (2006b). Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 233–243, Portland, ME, USA.

Elkarablieh, B. and Khurshid, S. (2008). Juzi: A tool for repairing complex data structures. In *International Conference on Software Engineering*, pages 855–858.

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

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

Ghezzi, C. and Mocci, A. (2010). Behavior model based component search: an initial assessment. In *ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*.

Gopinath, D., Malik, M. Z., and Khurshid, S. (2011). Specification-based program repair using sat. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 173–188. Springer.

Gulwani, S., Korthikanti, V. A., and Tiwari, A. (2011). Synthesizing geometry constructions. In *Conference on Programming language design and implementation*.

Haiduc, S., De Rosa, G., Bavota, G., Oliveto, R., De Lucia, A., and Marcus, A. (2013). Query quality prediction and reformulation for source code search: The refoqus tool. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1307–1310, Piscataway, NJ, USA. IEEE Press.

Harman, M. (2007). The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357.

Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. In *Automated Software Engineering*, pages 34–43.

introclass (2008). Klee. http://repairbenchmarks.cs.umass.edu/IntroClass/.

Jeffrey, D., Feng, M., Gupta, N., and Gupta, R. (2009). BugFix: A learning-based tool to assist developers in fixing bugs. In *International Conference on Program Comprehension*.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, pages 215–224.

Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. (2011). Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 389–400, San Jose, CA, USA.

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM.

Kapser, C. J. and Godfrey, M. W. (2008). "Cloning considered harmful" Considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692.

Ke, Y., Stolee, K., Goues, C. L., and Brun, Y. (2015). Repairing programs with semantic code search. In *International Conference on Automated Software Engineering*. IEEE/ACM.

Keivanloo, I., Rilling, J., and Zou, Y. (2014). Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 664–675, New York, NY, USA. ACM.

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA.

klee (2008a). Klee. https://en.wikipedia.org/wiki/Klee%27s_measure_problem.

klee (2008b). Symbolic. https://en.wikipedia.org/wiki/Symbolic_execution.

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

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

Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. (2003). Bug isolation via remote program sampling. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, San Diego, CA, USA.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM.

Liu, P. and Zhang, C. (2012). Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, pages 299–309.

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781.

Orlov, M. and Sipper, M. (2011). Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192.

Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., and Zeller, A. (2014). Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)*, 40(5):427–449.

Penix, J. and Alexander, P. (1999). Efficient specification-based component retrieval. *Automated Software Engineering*, 6.

Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. (2009). Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*.

Podgurski, A. and Pierce, L. (1993). Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering Methodology*, 2.

Qi, Y., Mao, X., and Lei, Y. (2013). Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 180–189. IEEE.

Qi, Z., Long, F., Achour, S., , and Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Technical Report MIT-CSAIL-TR-2015-003, MIT Computer Science and Artificial Intelligence Laboratory.

Reiss, S. P. (2009). Semantics-based code search. In *Proceedings of the International Conference on Software Engineering*, pages 243–253.

Research Triangle Institute (2002). The economic impacts of inadequate infrastructure for software testing.

Sidiroglou, S. and Keromytis, A. D. (2005). Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49.

Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. (2015). Is the cure worse than the disease? A large-scale analysis of overfitting in automated program repair. Technical Report UM-CS-2015-007, University of Massachusetts School of Computer Science.

Stolee, K. T. (2013). Solving the Search for Source Code. PhD Thesis, University of Nebraska–Lincoln.

Stolee, K. T. and Elbaum, S. (2012). Toward semantic search via SMT solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 25:1–25:4.

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

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

W. Eric, V. D. and Wong (2010). Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, Paris, France.

Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. (2010). Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72.

Weimer, W. (2006). Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190.

Weimer, W., Fry, Z. P., and Forrest, S. (2013). Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 364–374, Vancouver, BC, Canada.

Weiß, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Workshop on Mining Software Repositories*.

Wilkerson, J. L., Tauritz, D. R., and Bridges, J. M. (2012). Multi-objective coevolutionary automated software correction. In *Genetic and Evolutionary Computation Conference*, pages 1229–1236.

Zaremski, A. M. and Wing, J. M. (1997). Specification matching of software components. *ACM Transactions on Software Engineering Methodology*, 6.