Abstract of "Towards Improving the Effectiveness of Automated Program Repair" by Qi Xin, Ph.D., Brown University, May 2018.

Automated program repair (APR) aims to save people time and effort by repairing a faulty program automatically. A significant branch of current APR techniques are search-based: they define a set of modification rules to create a search space of patches first and then search in the space for a correct patch. Current search-based APR techniques face two main problems: (1) the search space problem and (2) the patch overfitting problem. For (1), they define a huge search space of patches to support repairing a large set of bugs, but they are not effective in finding a correct patch within such a huge space. For (2), they are prone to producing a patch that is overfitting. An overfitting, patched program can pass the test suite but does not actually repair the bug.

To address (1), we developed our APR techniques ssFix and sharpFix. ssFix finds and reuses existing code fragments (from a code database) that are syntactically related to the context of a fault to produce patches for its repair. By leveraging such syntax-related code, ssFix potentially creates a search space of patches whose size is reduced but is still likely to contain a correct patch. We evaluated ssFix on 357 bugs in the Defects4J bug dataset. Our results showed that ssFix successfully repaired 20 bugs with valid patches generated and that it outperformed five other repair techniques for Java. sharpFix is an improved version of ssFix. Compared to ssFix, it uses different code search and code reuse methods. For the 357 Defects4J bugs, sharpFix repaired in total 36 bugs with correct patches generated.

To address (2), we developed our patch testing technique DiffTGen which identifies a patched program to be overfitting by first generating new test inputs that uncover semantic differences between the original faulty program and the patched program, then testing the patched program based on the semantic differences, and finally generating test cases. We evaluated DiffTGen and found that it identified 49.4% overfitting patches in our patch dataset and that it can help an APR technique in producing less overfitting patches and more correct ones.

Towards Improving the Effectiveness of Automated Program Repair

by
Qi Xin
B. E., Dalian University of Technology, 2011
M. C. S., Rice University, 2013
Sc. M., Brown University, 2016

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2018

This dissertation by Qi Xin is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Steven P. Reiss, Director

Recommended to the Graduate Council

Date _____ _____

Shriram Krishnamurthi, Reader

Date _____ _____

Alessandro Orso, Reader
(Georgia Institute of Technology)

Date _____ _____

Tim Nelson, Reader

Approved by the Graduate Council

Date _____ _____

Andrew G. Campbell
Dean of the Graduate School

Acknowledgements

I would like to thank my advisor Dr. Steven Reiss for his guidance, advice, support, and help throughout my years at Brown. Steve helped me identify interesting research problems to work on, gave me ideas for solving the problems, encouraged me to develop problem-solving approaches, and discussed with me for refining and improving the approaches. I am also thankful for his time and effort in both reading my papers, proposals, dissertation, and other documents and helping me resolve many low-level technical issues for research experiments. Steve is very nice and always has time for me when I need help. He gives me a great amount of freedom for doing research and does not push me to do things. Working with him, I felt more and more excited about my research work throughout these years.

Many thanks go to Dr. Shriram Krishnamurthi for our discussions about my research projects and for his valuable feedback and suggestions. I am also grateful for the good pieces of advice he gave for solving research problems, reading and writing papers, and being a good researcher. He encouraged me to focus on doing impactful research rather than just creating publications. I will always keep this in mind.

I would also like to thank the other committee members, Dr. Alessandro Orso and Dr. Tim Nelson, for reading my papers, proposal, dissertation and for providing feedback and suggestions on my research work. I would also like to thank the anonymous reviewers for their reviews of the three pieces of thesis work (presented in Chapters 3, 4, & 5), which I submitted to different conferences for publication. I thank Dr. Eugene Charniak and Dr. Erik Sudderth for their feedback and suggestions for the first research project I worked on in my Ph.D. study.

I thank the following departmental staff members for their help: Lauren Clarke, Laura Dobler, Jesse Polhemus, Dawn Reed, Eugenia deGouveia, John Bazik, Donald Johwa, Kathy Billings, Benjamin Nacar, Frank Pari, and Paul Vars. Special thanks go to Lauren Clarke and Frank Pari who helped me a lot.

I thank my friends Chris Tanner, Xiaoxiao Hou, Bin Xu, Jie Zhang, and many others for their help, support, and encouragement.

Finally, I thank my family, especially my parents, for their endless support and love. I dedicate this thesis to my parents: Chibing Xin and Weiwei Guo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's world, software is used everywhere. Software is complex and often contains many faults (or bugs[1]). At the release time in February 2000, Microsoft Windows 2000 had about 35M LOC and contained over 63,000 known faults [4] (about two bugs per 1,000 LOC). Software faults can lead to expensive failures which further lead to significant financial loss. According to the report [209] in 2002, "Estimates of the economic costs of faulty software in the U.S. range in the tens of billions of dollars per year and have been estimated to represent approximately just under 1 percent of the nation's gross domestic product (GDP)". Such economic costs are likely to be higher today.

Debugging is the process that people take to remove bugs in software. Today's debugging process is largely manual: people often create printing statements in their programs or use a debugger (e.g., GDB [8]) to see program states during the failed execution to figure out what goes wrong and then come up with a fix to resolve the failure. Manual debugging as such however is often laborious, time-consuming, and error-prone. According to the report [35] in 2013, software developers spend about 50% of their programming time for debugging. The estimated global cost of debugging per year is 312 billion dollars. Kim and Whitehead [104] found that the median bug-fixing time of two projects ArgoUML (from 01/2002 to 03/2003) and PostgreSQL (from 07/1996 to 11/2000) is about 200 days.

Within the past few decades, many automated techniques that look at making debugging easier have been developed. A branch of such techniques called the *fault localization* techniques look at identifying some parts of the program that are likely to contain the bug. A developer can then save time by only focusing on the fault-located parts of the program for debugging. For example, a branch of such techniques called the *Spectrum-Based Fault Localization* (SBFL) techniques [202] calculate the suspiciousness of the program elements (as lines, statements, branches, or blocks) in a program based on the coverage of those elements in the passing and failing testing runs (against a set of test cases). Intuitively, a program element, for example as a statement, that is covered in more failing runs and less passing runs is more likely to be suspicious. An SBFL technique ranks

[1] In this thesis, we use "bug" and "fault" interchangably.

these program elements by the calculated suspiciousness and presents the ranked elements to the developer for debugging. A developer can then examine the elements in the ranked order to possibly identify the fault quickly for bug-fixing. Though such a technique is expected to make debugging easier by suggesting highly suspicious elements of a program, its actual usefulness is limited. Parnin and Orso [174] compared developers doing debugging tasks with and without a debugging tool that generates ranked suspicious statements using an SBFL technique Tarantula [90]. They found that the ranked suspicious statements only actually help expert developers on easy tasks and do not help developers on hard tasks. They also found that changing the ranks of those statements can cause no significant effects for debugging. In fact, fault localization is only one part of the debugging process. Even given the located fault, a developer still needs to understand how the fault causes the failure and then figure out how to make changes to remove the fault.

To significantly save people time and effort for debugging, within the past decade, researchers look at developing automated program repair (APR) techniques [155] that can automatically repair a fault with a patch generated. A typical APR technique accepts as input a faulty program and a fault-exposing test suite which consists of a set of test cases at least one of which the program failed. The test suite is actually used as a specification that encodes the expected behaviors that a bug-fixed program should have. As output, the APR technique can generate a patch for the faulty program such that the patched program passes the test suite (and thus satisfies the specification).

Many APR techniques have been developed over the past decade, they look at using different approaches: genetic algorithms [116, 74, 219], random search [182], human-written templates [101], existing bug-fixing instances [70, 113], program comparison [205], program synthesis [160, 152, 153, 53, 112, 197], condition synthesis [234, 232], repair templates plus condition synthesis [132], modifications with patch ranking models [134, 190], modifications based on monitored program states [47], code search [97], learned transformations [187, 131], invariants [177], bug reports [128], statistical analysis [95], and non-test-suite specifications [217, 73] for bug repair. A significant fraction of current APR techniques adopt a search-based approach[2] [74, 219, 182, 101, 132, 134, 113, 205, 190]: they define a set of modification rules to generate a space of patches first and then search in the space for *plausible* patches (the patched programs can pass the test suite). Compared to the APR techniques [160, 152, 153, 112] that are synthesis-based, search-based techniques often have better scalability, and have been actually applied to repair bugs for many real projects.

Current search-based APR techniques face two problems: (1) the search space problem and (2) the patch overfitting problem. As pointed out by Long and Rinard in [133], a search-based technique often defines a huge search space of patches to support repairing different types of bugs. However, searching for a correct patch within a huge search space is often difficult. Long and Rinard also found that the search space created by a search-based APR technique, though huge, still often fails to cover a correct patch. For the 69 bugs selected from the GenProg's bug dataset [74], the search space of the state-of-the-art APR technique Prophet contains correct patches for only 19 (27.5%) bugs. Using a 12-hour time budget for repairing one bug, Prophet repaired in total 39 (less than

[2]Such techniques are also called generate-and-validate (g&v) techniques.

60%) bugs with plausible patches generated. In [133], Long and Rinard also found that the search space created by a search-based APR technique often contains many overfitting patches. Using an overfitting patch, one can produce a patched program that passes the test suite but does not actually repair the bug. In Prophet's search space, the number of overfitting patches is often hundreds of times as many as the non-overfitting, correct patches. The existence of such overfitting patches can block the finding of a correct patch. For the 69 bugs, Prophet produced more than 60% of first-found plausible patches that are overfitting and incorrect. Early APR techniques are much more likely to produce overfitting patches. In [183], Qi et al. show that the majority of the patches generated by GenProg [116], AE [219], and RSRepair [182] are incorrect. Patch overfitting is a problem faced by not only the search-based APR techniques but all APR techniques that use a test suite as the correctness criterion for patch validation. A test suite, essentially as a set of input-output examples, cannot completely encode all the expected behaviors a program needs to have. Using a test suite for patch validation, an APR technique cannot avoid producing a patched program that passes the test suite but does not actually repair the bug (such a patched program can somehow make sure it works for the input-output examples used in the test suite but cannot generalize for other examples). In this thesis, we propose new techniques to address the two problems as a way to improve the effectiveness of automated program repair.

1.1 An Overview of Contributions

The contributions of this thesis are reflected in the techniques we developed to address the aforementioned two problems faced by current search-based APR techniques (and other APR techniques in general). To address the search space problem, we developed two APR techniques ssFix and sharpFix which find and reuse existing code fragments from existing programs for bug repair. To address the patch overfitting problem, we developed a patch testing technique DiffTGen which generates new test cases to identify an overfitting patch.

1.1.1 Leveraging Existing Code for Bug Repair

As suggested in [133], one solution to address the search space problem is to build a "targeted" search space of patches whose size is reduced but still contains the correct patches. We developed an APR technique ssFix which looks at finding and reusing existing code fragments from a code database to build such a targeted search space. ssFix is built upon the assumption that existing code fragments (in a large code database) that are similar to the bug context might contain the fix ingredients (the statements/expressions that can be reused to produce the correct patches) for bug repair. After doing fault localization to identify a faulty statement, ssFix first performs syntactic code search to find existing code fragments that are similar to the bug context from a code database that consists of the local faulty program and the non-local programs in a large code repository and then reuses those retrieved code fragments to produce patches for bug repair. To produce patches, ssFix looks at the syntactic differences between a retrieved code fragment and the bug context. For

a code fragment that is similar to the bug context, the syntactic differences are small. By leveraging such code fragments for bug repair, ssFix essentially creates a space of patches whose size is reduced but is still likely to contain the correct patches. More details can be found in Chapter 3. The experiments we conducted demonstrated the effectiveness of ssFix: ssFix repaired 20 of the 357 Defects4J bugs with valid patches generated with the median time of producing a plausible patch being only about 11 minutes. ssFix outperformed five other APR techniques in producing more correct patches with at least comparable (and often less) running time.

1.1.2 Revisiting ssFix for Better Bug Repair

ssFix was built upon the idea of reusing existing code fragments from a code database (that consists of the local faulty program and the non-local programs in a code repository) for bug repair. It implicitly makes the assumption that existing code fragments from the code database often contain the fix ingredients (the statements/expressions needed for producing a correct patch). We conducted an experiment to check how often the assumption holds in practice. ssFix performs syntactic code search to find existing code fragments that are similar to the bug context and then reuses those code fragments to produce patches for bug repair. We evaluated ssFix to see whether it is good at finding useful code fragments for bug repair and whether it is good at reusing those code fragments to produce the correct patches. ssFix largely relies on an existing technique [40] to do fault localization which we found worked poorly in the experiment we performed to evaluate ssFix in Section 3.4. Without working on the real faulty statement, ssFix would fail to produce the correct patch. We conducted experiments to see whether ssFix can do better repair if the fault is accurately located.

For (1), we found that the idea of reusing existing code for bug repair is promising. We defined the fix ingredient for a simple patch in the context of automated program repair (a patch is simple if all the fixing changes are made within an expression or within a primitive statement which contains no children statements). For our experiments, we looked at 103 Defects4J bugs whose patches are simple from the Defects4J bug dataset (version 0.1.0). We found that the exact fix ingredients and the parameterized fix ingredients (with the non-JDK variable, type, and method names in the fix ingredients parameterized) exist for 50 and 80 of the 103 Defects4J bugs. For (2), we found that ssFix retrieved code fragments that contain the parameterized fix ingredients for 61 bugs within the top-500 retrieved results. After translation, for 38 bugs, the code fragments contain the exact fix ingredients. ssFix reused the retrieved code fragments to produce correct patches for 23 bugs (see Sections 5.3.1 and 5.3.2). For (3), we found that ssFix can produce 24% more correct patches if the statement that contains the fault is known in advance and it can produce 23% more correct patches if the method that contains the fault is known in advance. Based on our experimental observations, we developed a new APR technique sharpFix as an improved version of ssFix. sharpFix improves ssFix's code search by using different code search methods for retrieving code fragments from the local faulty program and from the non-local programs in the code repository. For patch generation, sharpFix goes through the same steps used by ssFix: code translation, code matching (or component matching in [230]), and modification. Each step however is different and improved. Our results show

that sharpFix retrieved code fragments that contain the parameterized fix ingredients for 59 bugs. After translation, for 42 bugs, the code fragments contain the exact fix ingredients. sharpFix reused the retrieved code fragments to produce correct patches for 30 bugs (see Sections 5.4.2 and 5.4.3). For the Defects4J bug dataset, sharpFix repaired in total 36 bugs with correct patches generated. To our knowledge, it outperformed all the existing APR techniques that were evaluated on this dataset in producing the largest number of correct patches so far. More details can be found in Chapter 5.

1.1.3 Generating New Test Cases for Identifying Overfitting Patches

Most of current APR techniques use a test suite as the correctness criterion for patch validation. A test suite consists of a limited set of test cases and cannot fully specify all the expected behaviors that a program should have. A patched program that passes the test suite might not actually repair the bug. This is the patch overfitting problem. To address the problem, our solution is to generate new test cases exposing the differential behaviors between a patched program and the faulty program (such differential behaviors can be related to the patch) and furthermore identify the patch to be overfitting using an oracle.

We developed our patch testing technique DiffTGen which accepts as input a faulty program, a patched program, and an oracle. As output, DiffTGen can generate a test case exposing any overfitting behavior of the patched program. To produce such a test case, DiffTGen first employs a test generator to generate a test input (as a test method containing a sequence of statements for testing) that exercises a syntactic difference between the faulty program and the patched program. DiffTGen next instruments the faulty program and the patched program, executes the two programs with the input to obtain two outputs, and checks whether the outputs are different. If the outputs are different, DiffTGen asks the oracle to tell which output is correct. If the output of the patched program is incorrect, DiffTGen determines the patched program to be overfitting and produces an overfitting-indicative test case if the correct output could be provided by the oracle. Such a test case can be added to the original test suite to make the test suite stronger. Using the augmented test suite, a repair technique can avoid yielding a similar, overfitting patch again. We evaluated DiffTGen on 89 patches generated by four APR techniques for Java. Among the 89 patches, we identified 10 patches to be non-overfitting and the other 79 patches to be possibly overfitting. Our results show that DiffTGen identifies in total 39 (49.4%) overfitting patches and yields the corresponding test cases. We further conducted a repair experiment using an APR technique together with DiffTGen. Our results show that DiffTGen can help an APR technique in producing less overfitting patches and more correct ones. More details can be found in Chapter 4.

The organization of the thesis is as follows. We discuss the related work in Chapter 2. We present the APR technique ssFix in Chapter 3. We present the patch testing technique DiffTGen in Chapter 4. In Chapter 5, we show the experiments we conducted to validate ssFix's built-upon assumption and to evaluate ssFix's code search, code reuse, and repair abilities. In Chapter 5, we also present the APR technique sharpFix which we developed as an improved version of ssFix and

show the experiments we conducted for its evaluation. In Chapter 6, the final chapter, we show our conclusions and discuss future research directions for making APR techniques more effective.

Chapter 2

Related Work

In this chapter, we discuss existing research work in the areas of debugging, automated program repair, patch overfitting, code search, and test-generation-based differential testing that are related to the thesis work.

2.1 Debugging

Debugging is the process of identifying and correcting the root cause of a software failure [246]. As a very simple debugging method, people create printing statements in their programs for printing out relevant program states for debugging. As a general method, people use a debugger (e.g., GDB [8]) to run the program under debug in a controlled condition for identifying and correcting bugs. Using a debugger, a developer can run a program in single steps. In each step, the developer can examine values available at that step to understand what the program is doing. In a debugging step, when the code to be debugged involves a method call, a debugger often allows the developer to either go *into* the call of the method to debug the code there or go *over* the method to see the executing effect of the method call. Instead of debugging a program from the start, a debugger often allows a developer to set up a breakpoint. Then the developer can debug the program from the breakpoint and examine values in single steps thereafter.

Many techniques have been proposed to make debugging easier. Program slicing techniques [221, 213, 56, 78, 233] can identify all the parts of the program that affect the value computed at a certain point. For debugging, given the point where a failure is observed, program slicing techniques can identify all the program parts (e.g., as statements) that contribute to the failure. Then the developer can focus on the identified parts for debugging. Existing program slicing techniques are categorized as *static* or *dynamic*. Static slicing techniques [221, 169, 138, 82] look at the program in the form of the control-flow graph (CFG) or the program dependence graph (PDG) and identify all the nodes in the graph that have control-flow and/or data-flow dependency relationships and/or reachability relationships with the target node. Such static techniques often provide to the developer too many statements and predicates to look at and may not be too helpful. Dynamic slicing techniques [109, 26]

can show only the statements and predicates that affect the target point for a cerain input. There are also existing techniques that combine using static and dynamic slicing [214], use conditions for computing slicing [41], etc. The Whyline tool developed by Ko and Myers [105, 106] allows a user to ask *why did* and *why didn't* questions related to a program's behavior. For a questioned behavior, Whyline builds a chain of runtime actions that caused the behavior (the causality chain) using static [213] and dynamic [251] slicing techniques. The user can further look at the causality chain for debugging.

Delta debugging [245, 248, 49, 247, 50, 166] is the technique that can be used to isolate the minimal program input, user interactions, code changes, program states, etc. that caused a failure. Delta debugging is based on dynamic testing. To work, it needs a testing function. Given some program input for example, the testing function checks whether the input can cause the failure. Delta debugging uses its search algorithm (that is based on binary search) to keep reducing the program input and testing the reduced input. After multiple trials of testing, it can identify the minimal part of the input that caused the failure. According to [246], delta debugging was successfully applied to simplify an input that caused a Mozilla-cannot-print failure. The length of the original input HTML code is 896 lines. Delta debugging successfully identified the `<SELECT>` tag from one line that caused the failure.

A real software project often contains a test suite as a set of test cases. The program passes some of the test cases and fails the others. The failed test cases indicate that the program contains a bug. Fault localization [226] looks at automatically identifying the buggy part of the program that leads to the failure exposed by the failed test cases. A branch of fault localization techniques leverage program slicing to locate faults (in the survey [226], Wong et al. discuss many of such techniques). Another branch of fault localization techniques called the Spectrum-Based Fault Localization (or SBFL) techniques [202] look at leveraging the spectrum (the coverage of program elements such as statements) obtained from running each of the test cases to locate faults. An early SBFL technique proposed by Renieris and Reiss [186] compares the program spectra obtained from a failing run and a passing run that is the most similar to the failing run (according to a distance measure) and identifies program elements that are suspicious of being faulty based on the difference between the two spectra. Going beyond, different SBFL techniques [90, 22, 21, 52, 157, 224] look at using different formula to compute the suspiciousness of program elements based on the spectra obtained from running the test cases. Though different formula were used, their basic ideas are similar: giving more suspiciousness to program elements (e.g., statements) that are covered in more failing runs and less passing runs. The effectiveness of using different formula for fault localization were compared in [22, 21, 157, 24, 111, 224, 93, 236, 235, 110]. Though some formula (e.g., Ochiai [22]) are found to be better than others for locating seeded faults, the study [242] shows that no formula can be consistently better than another for all cases, and a "best" formula does not exist. Another study by Pearson et al. [176] also found that formula is one of the least important factors determining the effectiveness of SBFL techniques. More advanced SBFL techniques look at leveraging additional information such as invariants [181, 191], method call sequences [52, 127], time [240], test case

selection [76] and differentiation [249], execution frequency [20, 117], metamophic relations [228], control-flows [252, 253], etc. There are also other fault localization techniques that are statistical-based [123, 48, 126], machine learning-based [227, 225, 34], data mining-based [158, 43, 44, 250], mutation-based [156, 173], model-based [147, 148], and others [194, 106, 91, 92, 72, 256, 189, 121]. In the survey [226], Wong et al. comprehensively discussed existing fault localization techniques.

Fault localization can identify suspicious program elements (e.g., statements) that are likely to be buggy. By examining the highly suspicious program elements instead of the whole program, a developer can save time and effort on examining bug-irrelevant parts of the program and thus save time on debugging. This is however not practically true. Parnin and Orso conducted a user study [174] comparing developers doing debugging tasks with and without a debugging tool that generates ranked suspicious statements using an SBFL technique Tarantula [90] and found that these located suspicious statements can only help experts on easy debugging tasks and do not help developers on hard tasks and that changing the ranks of those statements can cause no significant effects for debugging. Even with the fault accurately located, a developer still needs to understand how the fault caused the failure given certain inputs and how to modify the original program to remove the fault. Within the past decade, many automated program repair techniques [155] are developed. Fault localization is commonly used by these techniques for identifying some small part(s) of the original program for repair.

2.2 Automated Program Repair

Automated program repair (APR) [155] looks at automatically repairing a faulty program with patches generated. A variety of different approaches to automated program repair have been taken. Arcuri et al. [27, 28] proposed the idea of using Genetic Programming to automatically repair bugs. As one early system, GenProg [220, 116, 74, 114] was developed based on such an idea. It uses mutation and crossover operators to create program mutants at the statement level and uses a genetic algorithm to search for any mutant program that passes the test suite. It is built upon the assumption that a program itself is likely to contain the correct statements used for repairing a bug. Debroy and Wang [57] investigated using an existing fault localization technique Tarantula [90] to locate a fault in a statement and uses a limited set of mutation operations which look at changing the operator of an expression (e.g., `+`) and negating an if/while condition to repair a bug. AE [219] is a variation of GenProg. It uses an adaptive search algorithm and leverages program equivalence analysis to reduce the search space. RSRepair [182] extends GenProg's mutation operators but uses a random search algorithm to repair the program. It also employs test case prioritization techniques [241] to speed up the process of patch validation. After analyzing the generated patches by GenProg, AE, and RSRepair, Qi et al. [183] found that the majority of the patches generated by these techniques are equivalent to single modifications that delete functionality and are not correct. They developed Kali that only does simple functionality deletions and showed that Kali is no worse than the three tools in terms of producing correct patches. One of the reasons why these techniques

can produce incorrect patches is that the test suites that they used for bug repair are weak. Using stronger test suites, however, Qi et al. found that one of the techniques GenProg can produce no more correct patches.

PAR [101] is a pattern-based approach for automated program repair. Instead of applying genetic operations on a suspicious statement as GenProg does, PAR first analyzes the context of the statement and then applies one of the 10 predefined fix templates to generate program variants. In [154], Monperrus points out that the fix templates used by PAR do not address any defect class, and most bugs seem to be fixed by only two of the templates.

To repair a bug, SPR [132] works in stages. It first applies its transformation schemas to produce fixing templates for a suspicious program statement. SPR next performs value search or condition synthesis to yield patches. Prophet [134] is an extension of SPR. It uses a trained probabilistic model to rank the generated patches to possibly avoid reporting an overfitting patch (an overfitting, patched program can pass the test suite but does not actually repair the bug). A further study [133] of Prophet and SPR on 69 defects from the GenProg benchmarks by Long and Rinard shows that (1) correct patches are sparse in the tools' search spaces and the number of overfitting patches is often hundreds of times as many as the non-overfitting, correct patches, therefore it is often difficult to find the correct patches from the tools' search spaces, and (2) increasing the search space would not necessarily enhance the repair performance.

In [45], Chandra et al. proposed the idea of finding angelic values to identify suspicious expressions that need to be fixed in a given scope of a faulty program. An angelic value for an expression is a value the expression needs to take to make the program pass the failed test case. For a passed test case, if the expression takes the angelic value (rather than the value it actually takes), the program can still pass the test case. Chandra et al. looked at using symbolic execution and constraint-solving for finding an angelic value for an expression. If the angelic value can be found, the expression is reported as suspicious. SemFix [160], as a semantics-based approach, uses the idea to do bug repair. It first employs a fault localization technique to obtain a list of suspicious statements. SemFix inspects each statement in the list. Each time, it works on one statement to produce patches. For a suspicious statement, SemFix assumes the defective part of the statement is an expression, so it replaces the expression with a new symbolic variable and performs symbolic execution to yield constraints encoding the expected semantics. SemFix employs component-based synthesis technique to produce a new expression for the introduced symbolic variable so as to make the constraints satisfied. The goal of DirectFix [152] is to produce simple patches that change the program in a minimal way. To do so, DirectFix uses the hard and the soft constraints. The hard constraint encodes the expected semantics of the program and must be satisfied. The soft constraint, however, encodes the structure of the expressions to be modified and should be maximally satisfied. After generating the constraints, DirectFix applies a variant of the component-based synthesis technique [85] where a partial MaxSMT solver is used to produce patches. Angelix [153] is a further improvement over SemFix and DirectFix. Compared to the two previous techniques, it is more scalable and can make multi-line changes. To repair a bug, Angelix replaces the most suspicious expressions with newly introduced

variables and performs a variant of symbolic execution to find angelic values for those variables (i.e., the values for these variables that a correctly patched program would take) along different paths, or the angelic paths. For each test case, Angelix obtains a set of angelic paths. Angelix encodes all the angelic paths obtained from each test case into a constraint and employs component-based synthesis to produce patches while maximally maintaining the structure of the original program. S3 [112] is another synthesis-based repair technique. Given a faulty location (which it identifies using fault localization [23]), S3 identifies the input-output examples for this location as the specification. The input examples are the running values of the variables and methods that are syntactically available at the location (S3 runs the faulty program against the test suite to obtain those values). To obtain the output examples, S3 uses dynamic symbolic execution. One difference between S3 and Angelix lies in the forms of constraints generated: Angelix specifies the actual output should be equal to the expected output for a path condition as the constraint while S3 specifies that no run-time errors (such as assertion errors, array index-out-of-bound errors, etc.) should occur by following a path (i.e., satisfying a path condition). S3 does not use Angelix's constraint form. As explained in the paper [112]: "it usually requires users to instrument the output variables manually". After identifying the input-output examples as the partial specification for the faulty location, S3 uses an enumeration-based technique to do synthesis for patch generation. It uses a relatively sophisticated ranking model to rank patches to avoid patch overfitting [200].

Nopol [234] is another synthesis-based approach. It targets on repairing bugs that are condition-related: it can either repair a buggy if-condition or insert a precondition, as an if-checker, for a statement. Similar to Angelix, Nopol first finds the angelic values for the conditions to be repaired. But different from Angelix, Nopol does not employ symbolic execution to find the values. Rather, it uses value replacement [84] as a simpler approach. After identifying the angelic values, Nopol uses a SMT-based synthesis approach to synthesize specific expressions to yield a patch. relifix [205] targets on regression bugs. It accepts two versions of programs (a regression bug was introduced in the later version) and a test suite as inputs. relifix uses eight mutation operators to modify the later version of program and looks for any patch that is plausible (i.e., the patched program can pass the test suite).

Antoni et al. [53] introduces the syntactic and semantic distances between a program and a patch for it. The developed repair technique Qlose uses SKETCH [201] to produce patches and selects a patch that has the minimum syntactic and semantic distances from the original program.

SearchRepair [97] performs semantic code search to find code fragments that are likely, but not guaranteed, to be semantically correct from the established code database and leverages such code fragments to produce patches. SearchRepair yields constraints for each code fragment via symbolic execution encoding the semantics. For code search, SearchRepair uses constraint-solving technique to find the code fragments that are likely to be useful for repair: they should yield the same output yielded by the buggy fragment for any positive tests but should not produce any output yielded by the buggy fragment for any negative tests. SearchRepair was only shown to work for small C programs. Code Phage [196] does code search to find programs in a code repository that have the

correct semantics and does code transfer to leverage code in the correct programs to fix three types of errors: out-of-bounds, integer overflow, and divide-by-zero errors. QACrashFix [70] proposed by Gao et al. leverages existing fixing instances from Q&A sites to repair crash bugs. To do the repair, the technique uses the crash message to search the Q&A sites for similar crashing issues. For a similar issue, it identifies and extracts the bug and fix code fragments. The technique compares the extracted bug code fragment to the fault location identified from the faulty program and leverages the extracted fix code fragments to do repair. Our repair technique ssFix [230] performs syntactic code search to find existing code fragments (from the local faulty program and the external code repository) that are syntax-related to the bug context and then reuses each retrieved code fragment to produce patches. sharpFix is an improved version of ssFix. It uses different search methods to do code search within different searching scopes: the local faulty program and the external code repository. For patch generation, it performs three steps: candidate translation, code matching, and modification as ssFix does. Each step however is different and improved. More details about ssFix and sharpFix can be found in Chapter 3 and Chapter 5.

The history-driven repair technique HDRepair [113] developed by Le et al. uses a genetic algorithm similar to what GenProg uses. The main differences between the two techniques are two-fold: (1) the history-driven technique uses more mutation operators (12 in total) than GenProg (3 in total) for creating program mutants and the history-driven technique does not use the crossover operator and (2) the technique leverages 3,000 bug-fix instances mined from GitHub for computing the fitness score during the patch search process whereas GenProg computes the fitness score based on the test results.

The study by Tan et al. [206] applies anti-patterns as forbidden repair modifications to search-based repair techniques for improving their performances. They developed seven types of anti-patterns that forbid a repair tool to generate the seven types of non-sensical patches that essentially delete important statements.

ACS [232] performs condition synthesis to produce two types of patches: (1) inserting either an if-throw or if-return statement and (2) changing the condition of an existing if-statement (by either widening or constraining the original if-condition) for bug repair. ACS does test case analysis, document analysis, dependency analysis, and predicate mining to synthesize an if-condition and produce a patch. Given a suspicious statement, JAID [47] monitors the program state at the position of the statement during fault localization to identify suspicious state. It next modifies the program in a way to avoid the suspicious state for bug repair. ELIXIR [190] uses eight types of modifications to produce patches and uses a probabilistic model (a logistic regression model) to sort patches for validation (based on contextual analysis, contextual comparison, and bug-report analysis).

R2Fix [128] leverages bug reports to do repair for three types of bugs: buffer overflows, null dereferences, and memory leaks. Given the bug report, R2Fix first classifies the report as describing one of the three bugs. It next extracts the target file to be modified from the bug report and applies its fix patterns created for the bug type to do repair. Some of the fix patterns require parameters, R2Fix leverages the bug report to extract parameters that are likely to be useful (via name scanning).

Refazer [187] is a technique that accepts a set of bug-fixing instances and works on synthesizing fixing transformations that conform to these instances and can be further used for fixing similar bugs. Refazer has been experimented with fixing bugs for introductory programming assignments. Genesis [131] can learn from transformations from existing patches and generate specific patches based on the transformations to repair three types of bugs: null-pointer, out-of-bounds, and class-cast bugs.

AutoGrader [197] uses program synthesis to automatically provide feedbacks for student's incorrect code. The focus of the technique is generating feedback rather than patches. For that purpose, the approach requires the solution to the incorrect code and the correction rules. MintHint [95] applies statistical analysis in selecting expressions that are likely to be correct for a faulty statement, and then synthesizes repair hints for it.

Many other repair approaches exist that are invariant-based [177], that are deep-learning-based [77], that leverage test-input generation [161], that work on specific types of bugs [42, 164, 192, 199, 195, 165, 204, 88, 222, 211, 54, 25, 139, 68, 216, 243, 129, 163, 69, 100], and that require specifications other than test suites [58, 73, 79, 218, 62]. The program repair website [13] keeps track of publications that are relate to (automated) program repair.

2.3 The Performance of Current APR Techniques

GenProg [116] as an early APR technique was evaluated on a set of 105 real bugs selected from 8 C projects: *fbc*, *gmp*, *gzip*, *libtiff*, *lighttpd*, *php*, *python*, and *wireshark*. As reported in [116], GenProg repaired 55 bugs. It took GenProg about 1.6 hours to repair each of these 55 bugs. Since GenProg's repair uses randomness, it was actually run in 10 trials for repairing a bug. A bug was considered to be repaired if at least one trial was successful. RSRepair [182] was evaluated on 24 bugs selected from the GenProg's dataset (which contains 105 bugs). RSRepair was run in 100 trials for repairing each bug, and it repaired all the 24 bugs. For the same 24 bugs, GenProg was also run in 100 trials and repaired all the bugs. For 23 of the 24 bugs, however, RSRepair was shown to outperform GenProg in using fewer patch trials and test case executions for bug repair. AE [219] was also evaluated on the GenProg's dataset. Since AE is deterministic, it was run in one trial. Results in [219] showed that it repaired 53 bugs. Running in one trial, GenProg only repaired 37 bugs. Qi et al. [183] examined the patches generated by GenProg, RSRepair, and AE and found that the majority of these patches are incorrect (they often introduce security vulnerabilities or delete program's expected functionalities). For the 105 bugs in the GenProg's dataset, GenProg and AE actually produced correct patches for only 2 and 3 bugs respectively. For the 24 selected bugs, RSRepair produced correct patches for only 2 bugs. Kali is a repair technique developed by Qi et al. that deletes functionalities of a program. The experiments in [183] showed that Kali produced as many correct patches as the three techniques did. Long et al. developed SPR [132] and Prophet [134] and evaluated the two techniques on the GenProg's dataset. Out of the 105 bugs in the GenProg's dataset, Long et al. found that the developer patches for 36 bugs were not actually produced for bug repair but for functionality

changes. For the other 69 bugs, SPR and Prophet produced plausible patches for 38 and 39 bugs and correct patches (as the first found plausible patches) for 11 and 15 bugs respectively. For SPR, the average running time of producing a correct patch is 89 minutes. For Prophet, the average running time of producing a correct patch is 138.5 minutes. Long et al. found that patch validation (for applying a patch on the faulty program to get a patched program, compilation, and running test cases) is the most expensive part which takes more than 95% of the technique's running time. SPR and Prophet generated many more correct patches than GenProg, RSRepair, and AE did. Overall SPR and Prophet generated plausible patches for less than 60% of these 69 bugs with more than 60% of the plausible patches being overfitting.

SemFix [160] is an early APR technique that performs program synthesis to produce patches. SemFix was evaluated on 90 bugs from five C projects in the SIR dataset [59]: *Tcas*, *Schedule*, *Schedule2*, *Replace*, and *Grep*. Each SIR program is associated with a test suite containing thousands of test cases. The number of test cases (for each project) used for the repair experiments is limited to 50. Given the small number of test cases for each project, SemFix repaired 48 bugs, and outperformed GenProg which repaired 16 bugs. SemFix was also shown to outperform GenProg on repairing a small number of buggy programs (nine in total) for the GNU core utilities. DirectFix [152] was also evaluated on the SemFix's bug dataset. For some of the bugs, however, DirectFix was given the buggy methods to work on. DirectFix was shown to repair 59% bugs with 56% of those repairs being correct. DirectFix was shown to significantly outperform SemFix in producing more correct patches. Angelix [152] was claimed to be better than SemFix and DirectFix in achieving better scalability. Angelix was evaluated on 82 bugs from five C projects in the GenProg's dataset: *wireshark*, *php*, *gzip*, *gmp*, and *libtiff*. Angelix generated plausible patches for 28 bugs and correct patches for 10 bugs. Angelix's performance was shown be comparable to SPR's performance in repairing the 82 bugs: SPR produced plausible patches for 31 bugs and correct patches for 11 bugs. The average running time for Angelix to produce a patch is 32 minutes.

Many APR techniques for Java were evaluated on the Defects4J bug dataset (version 0.1.0) [94] which contains in total 357 bugs from five real Java projects: *JFreeChart*, *Closure Compiler*, *Commons Math*, *Joda-Time*, and *Commons Lang*. Martinez and Monperrus developed ASTOR [145, 16], a program repair library, which makes available the Java versions for GenProg [116], Kali [183], and MutRepair (the repair technique developed by Debroy and Wang [57]) as JGenProg2, JKali, and JMutRepair. In [143], Martinez et al. evaluated JGenProg (the previous version of JGenProg2), JKali, Nopol (version 2015) on the selected 224 Defects4J bugs from four of the five projects (except *Closure Compiler*). For experiments, they ran each tool to repair each bug with a timeout set as 3 hours. Their results showed that jGenProg, jKali, and Nopol produced plausible patches for 27, 22, and 35 bugs respectively and that jGenProg, jKali, and Nopol produced correct patches for 5, 1, and 5 bugs respectively. The median running times for jGenProg, jKali, and Nopol to repair a bug (with a plausible patch generated) are 61, 18, and 22 minutes respectively. The average running times are 55, 23, and 33 minutes respectively. In 2017, Durieux et al. evaluated Nopol-SMT (the version using SMT solver) on all the 395 bugs from the Defects4J dataset (version 1.1.0).

The results in [60] showed that Nopol-SMT produced plausible patches for 103 bugs. The number of correct patches was not reported (though all the plausible patches were released). HDRepair [113] was evaluated on 90 simple bugs selected from the Defects4J dataset (version 0.1.0) containing 357 real bugs. HDRepair repaired 23 bugs with correct patches generated (HDRepair was run to generate at most 10 plausible patches for repairing a bug). However, for only 13 bugs, HDRepair produced correct patches as the first plausible patches. To repair these bugs, HDRepair was actually given the faulty method to work on (rather than the whole program). It took HDRepair on average 20 minutes to produce a plausible patch. Le et al. also ran experiments to compare HDRepair with PAR (which they re-implemented) and jGenProg and found that PAR and jGenProg produced correct patches for 4 and 1 bugs respectively. ACS [232] was evaluated on the 224 Defects4J bugs from four of the five projects (except *Closure Compiler*). For bug repair, ACS was given a 30-minute timeout. ACS repaired in total 23 bugs with plausible patches generated. Among the 23 patches, 18 are correct. ACS was shown to be efficient (the median time of generating a patch is about 5.5 minutes) and relatively effective in producing non-overfitting patches (the non-overfitting rate is 18/23=78.3%). ACS was shown to significantly outperform jGenProg, Nopol, PAR, and HDRepair in producing more correct patches and being more effective at producing non-overfitting patches. S3 was evaluated on two bug datasets: Dataset-1 consists of 52 bugs from the IntroClass dataset [115] (all the 52 bugs are related to one small program called *smallest*). Dataset-2 consists of 100 real bugs from 62 Java projects (the developer patches for these 100 bugs are simple: each patch makes a fixing change that is fewer than 5 lines of code). S3 was shown to produce correct patches for 22 bugs from Dataset-1 and 20 bugs from Dataset-2. S3 was shown to produce no overfitting patches. Angelix however was shown to produce correct patches for only 4 bugs from Dataset-1 and 6 bugs from Dataset-2. More than 50% of Angelix's generated plausible patches were shown to be overfitting. The running time took by S3 and Angelix for producing a patch was not reported. JAID [47] was evaluated on the selected 138 bugs from the 357 Defects4J bugs. JAID repaired 31 bugs with plausible patches generated. Among the 31 bugs, it repaired 25 bugs with correct patches. However, not all the correct patches are the first-found plausible patches. For 9 bugs, JAID actually produced the correct patches as the first-found plausible ones. JAID was shown to be slow: the average running time it took to repair a bug is about 120 minutes. The median running time is about 3 hours. ELIXIR [190] was evaluated on the 224 Defects4J bugs from four of the five projects (except *Closure Compiler*). ELIXIR repaired 41 bugs with plausible patches generated. Among the 41 bugs, it repaired 26 bugs with correct patches (as the first-found plausible patches) generated. The median/average running time ELIXIR took to produce a plausible patch for bug repair however was not reported, though the maximum running time for repair a bug was reported to be 90 minutes. For bug repair, ELIXIR used bug reports to rank the generated patches. Without using bug reports, ELIXIR was shown to be able to produce correct patches for 20 bugs. In [190], the authors showed that the number of overlapped correct patches generated by ELIXIR and ACS is only 4 (ACS produced in total 18 correct patches). ELIXIR was also evaluated on 127 bugs sampled from the *Bugs.jar* dataset. It produced plausible patches for 39 bugs. Among these bugs,

it produced correct patches for 22 bugs.

Our technique ssFix was evaluated on all the 357 bugs in the Defects4J dataset (version 0.1.0). The code repository used is the Merobase repository [83] containing 58,219 projects. Our results show that ssFix repaired 60 bugs with plausible patches generated. Among the 60 bugs, ssFix produced valid patches (as the first-found plausible patches) for 20 bugs. A patch was considered valid if it is plausible and does not break any expected behaviors of the original faulty program. Among the 20 bugs, ssFix produced correct patches (as the first-found plausible patches) for 15 bugs. The median time ssFix took to produce a plausible patch is only 10.7 minutes. For technique comparison, we ran jGenProg, jKali, Nopol (version 2015), HDRepair, and ACS on the same 357 bugs. We found ssFix outperformed these techniques in producing many more valid and correct patches. In terms of the median running time, ssFix was comparable to jKali and Nopol, and was faster than jGenProg, HDRepair, and ACS. More details can be found in Chapter 3. In Chapter 5, we show an experiment for which ran ssFix on the same 357 bugs using the DARPA MUSE repository [55] containing 66,341 projects. Compared to the experiment we did in Chapter 3, ssFix looked at using more candidates (100 more, or two times as many) for repairing a fault-located statement. Our results in Chapter 5 show that ssFix repaired 69 bugs with plausible patches generated. For 22 of the 69 bugs, it produced correct patches (as the first-found plausible patches). The median time of producing a plausible patch is about 10 minutes. We also evaluated sharpFix on the same 357 bugs. sharpFix repaired 89 bugs with plausible patches. For 36 of the 89 bugs, it produced correct patches (as the first-found plausible patches). The median time of producing a plausible patch is 11.3 minutes.

2.4 Patch Overfitting

Most APR techniques use test suites as the correctness criterion to evaluate their generated patches. A generated, patched program can pass the test suite but may not actually repair the bug: the patch often breaks some expected functional behaviors that the original, unpatched program has. Such a patched program, or the patch, is called *overfitting* because it overfits the test suite used for bug repair but fails to generalize to be a correct patch (which fully repairs the bug and does not break any expected behaviors of the original, unpatched program).

Existing studies [200, 183] investigated the quality of the patches generated by early APR techniques [116, 219, 182] and found that they are prone to producing overfitting patches. The study by Smith et al. [200] looked at two APR techniques: GenProg [116] and RSRepair [182] and investigated whether they are likely to produce overfitting patches for repairing small programs [115]. To measure the overfittingness of an APR technique, they used a held-out test suite. They found that the two techniques often generate patched programs that overfit the original test suite and do not pass the held-out test suite. For programs to be repaired that are closer to be correct, the two repair techniques are more likely to produce patches that break their expected behaviors. The study by Qi et al. [183] investigated the generated patches for real bugs by three APR techniques: GenProg

[116], AE [219], and RSRepair [182]. They found that the majority of their generated patches are incorrect. In fact, most of the generated patches are equivalent to functionality deletion and are harmful (because they delete a program's expected behaviors). Qi et al. found that the tests used by such tools are weak: they do not directly check the correctness of outputs of the patched programs but instead check for the "weaker properties" (for example, they check whether the patched program produces 0 as the exit code). Using stronger test suites, Qi et al. found that these techniques produced no patches at all.

To address the overfitting problem, Tan et al. [206] look at using seven types of anti-patterns as the forbidden repair modifications to avoid generating an overfitting patch (which for example deletes an exit call). Existing techniques [244, 229, 238, 130] look at generating new tests or test cases to possibly identify and avoid overfitting patches. The work by Yu et al. [244], as an initial effort, investigated the possibility of combining a repair technique with a test case generator to improve the ability of the repair technique in producing less overfitting and more correct patches. They found however such a combination does not effectively help in addressing the overfitting problem. One possible limitation is that they look at generating test cases for the buggy program. A test case that is generated as such does not always specify the expected behavior a program should have and can thus be problematic. Our technique DiffTGen [229] looks at generating new test inputs that expose the differential behaviors of the faulty, unpatched program and the patched program. Assuming both programs are deterministic, such differential behaviors are actually caused by the patch. DiffTGen asks an oracle (a human for example) to judge the correctness of the two behaviors and determines the patch to be overfitting if the patched program's behavior is not correct. If the oracle could provide an expected behavior (as an output) for the patched program, DiffTGen may produce a test case. Such a test case exposes the overfitting behavior of the patched program. Using such a test case, a repair technique can avoid generating a similar overfitting patch again. The technique Opad [238] developed by Yang et al. has a similar idea and can be used to identify overfitting patched programs that either crash or have memory-safety issues. Compared to DiffTGen, Opad uses automatic oracles (for memory-safety oracles, Opad uses Valgrind [19]) but is limited to identifying only two types of overfitting patches. The technique proposed in [130, 231] can automatically estimate the correctness of a patch. To do that, the technique first generates a set of new test cases without actually using an oracle. To do so, it employs an existing test generator Randoop [170] to generate a set of test inputs. Using each generated test input, it runs the faulty program to obtain an execution trace. It compares the new execution trace to the execution traces it obtained by running the faulty program against the original test suite to estimate the passing/failing status for the new trace. By running the faulty program and the patched program against each generated test case and comparing their executions, the technique estimates the correctness of the patch. The estimation is based on what follows: For a passing-status test case, if the execution of the patched program is much different from the execution of the faulty program, the patch is estimated as incorrect. For a failing-status test case, if the execution of the patched program is not much different from the execution of the faulty program, the patch is estimated as incorrect. Otherwise, the patch is estimated as correct.

Experiments from [130, 231] show that the technique successfully identified 56.3% incorrect patches. In theory, the technique can be inaccurate at identifying a correct patch as incorrect.

As another way of addressing the overfitting problem, many current APR techniques use a patch ranking model to possibly rank an non-overfitting patch before any overfitting patches to be reported. SPR [132] uses heuristics (based on eight rules) to prioritize the generated patches. Prophet [134], as an improvement of SPR, uses a probabilistic model trained from a large set of existing human-patches to rank its generated patches. To train such a model, Prophet extracts as features (1) the modification type, (2) the pair of statement types (for the unpatched and the patched statements), and (3) the pair of value characteristics which capture how variables and constants are used in the original program and in the patched program. In the patch synthesizing process, SemFix [160], DirectFix [152], and Angelix [153] look at synthesizing simple patches for the original faulty program that are possibly non-overfitting. QLOSE [53] tries to synthesize a patch to produce a patched program that can pass the test suite and are syntactically and semantically similar to the original faulty program. In [53], the authors defined two types of distances: the syntactic distance and the semantic distance. For syntactic distance, they look at the distances (either boolean-based or size-based) for all the changed expressions between the faulty program and the patched program. For semantic distance, they compare the execution traces between the faulty program and the patched program for each test input. An execution trace is a sequence of program configurations. A program configuration is made of a program location and a mapping from variables associated with that location to the values they hold. For calculating the semantic distance, QLOSE looks at each pair of configurations between the two execution traces. ACS [232] leverages dependency analysis, document analysis, and predicate mining to sort variables and predicates for synthesizing conditions to produce patches. The ranking model used by our technique ssFix (and also by sharpFix) looks at the types and sizes of the generated patches for patch sorting. A patch generated by deletion has a lower rank than a patch generated by non-deletion. A complex patch has a lower rank than a simple patch. JAID [47] ranks patches by the suspiciousness of the monitor states, the fixing actions, and the patch simplicity. S3 [112] uses a relatively sophisticated patch ranking method that is based on six ranking functions measuring the syntactic similarity between the unpatched and the patched code, the locality of used variables and constants, the number of formula-satisfying instances, output coverage, and anti-patterns. Similar to Prophet, ELIXIR [190] uses a trained probabilistic model to rank patches based on the syntactic distance between the bug and the repair code, the contextual similarity, the frequency of the used objects and variables in the context, and the bug report. MintHint [95] performs statistical correlation analysis to rank the expressions to be used for producing repair hints.

The proness of producing non-overfitting patches, i.e., *non-overfittingness*, is one measure for evaluating the effectiveness of an APR technique. But it is not the only measure. Consider a dummy APR technique X that generates no patches at all. The non-overfittingness for X is 1. But such a technique is useless. Another measure for evaluating an APR technique's effectiveness is *repairability*: the ability of finding a plausible patch for a bug (a plausible, patched program

can pass the test suite). A good APR technique should have both high repairability and high non-overfittingness.

Currently, a common way to evaluate the non-overfittingness of an APR technique is by comparing the patches it generated to the developer patches available from the bug dataset. To be considered as *correct*, a patch needs to be semantically equivalent to the developer patch. Defining a correct patch this way can lead to an underestimate of a repair technique's ability in producing correct, non-overfitting patches. For example, ssFix generated a "valid" patch by changing the declared type of a variable from *int* to *float* to avoid the precision loss. The developer patch changes the type from *int* to *double*. Because the patch generated by ssFix is not semantically equivalent to the developer patch, it is not considered as correct.

2.5 Code Search

Today's code repositories contain a huge set of software projects. In 2017, GitHub claims to have 67 million projects (as the GitHub repositories) [7]. It is quite possible for one to search the code repositories for relevant code to make his/her programming task easier. For example, instead of writing code from scratch for software development, one may consider finding and reusing existing code from a relevant project within the large code repositories. Existing code search engines like Open Hub [12], Krugle [10], and SearchCode [14] allow a user to use keywords to specify the code he/she wants, and they work on finding relevant code fragments containing those keywords. Current code repository hosting platforms: GitHub [6], SourceForge [15], and BitBucket [3] also enable keyword search for finding software projects and code files that a user wants. Sourcerer [124, 29] is a software infrastructure that can be used for code search and mining. For code search, it is possible to use Sourcerer to find code that contains certain syntactic structure (e.g., a switch statement and three loops), that contains certain keywords, that is of certain size (in LOC), etc. using different ranking schemes (e.g., a matched method name is assigned larger weight than a matched package name). S^6 [185] allows a user to specify the semantics of the program he/she wants using keywords, signatures, test cases, contracts, and security constraints. S^6 starts with keyword search to obtain an initial set of code results. It applies transformations to these code results and then finds those that can satisfy the static specifications such as the class/method signature. For each such code result, it generates dependent code for checking whether the code result can satisfy the dynamic specification, i.e., the test cases. S^6 finally reports to the user any code results that can satisfy the dynamic specification. CodeGenie [120] is a test-driven code search technique. Given the user-provided test cases and a missing program feature exposed by the test cases as a missing method, CodeGenie extracts information about missing method (e.g., the interface of the missing method) from the test cases, produces the query, and performs code search to find code for the missing method. CodeGenie can further integrate the retrieved code into the project, run the test cases, and show the results to the user. Portfolio [150] accepts the user-provided keywords as input and performs code search to find functions that not only match these keywords and but also are used in a way

that is possibly relevant to these keywords using the PageRank and the spreading activation network (SAN) algorithms. Exemplar [149] accepts the user-provided keywords as input and looks at the application documentation, API documentation, and dataflow of the API calls used in applications to locate relevant applications as the search results. Code search techniques that accept user-provided keywords as the search query often suffer from vocabulary mismatch problem. Recent code search techniques look at addressing the problem in different ways. Bajracharya et al. [30] investigated the possibility of using words from semantics-relevant code to improve the search results. In their research, they look for semantics-relevant code that has similar API usage. CodeHow [137] accepts user-provided keywords and identifies relevant APIs based on matching the keywords with an API name and its documentation. CodeHow uses the relevant APIs retrieved to create an expanded query, and performs code search using the expanded query for finding code results. CoCaBu [198] leverages the posts from Q&A sites for query expansion and then performs code search with the expanded query to find better search results. Many other techniques exist [142, 135, 119, 237, 162] that do query expansion/reformulation to improve code search results.

A branch of code search techniques look at finding code as API usage examples. PARSEWeb [210] allows a user to provide the source and destination types as the query, and it works on finding method call sequences that take the source type of object as input and produce a destination type of object. Prospector [140] leverages code examples mined from real code to help synthesize the API usage code as a way to produce the expected API type from the query API type. SNIFF [46] can also recommend API usage code to the user. It only allows the user to use free-form English words to describe the API he/she wants to use as the query. SNIFF matches the query against existing code fragments in a code database. SNIFF annotates each of the code fragments in the code database with the documents of the APIs used in the code fragment for indexing. This way, SNIFF's code search leverages the API documentation to effectively identify code fragments containing the API usage that user expects. Each code fragment retrieved may contain irrelevant code pieces that are very specific to their programs, SNIFF performs the type-based intersection of these code fragments to identify their common interesting part to be presented to the user. MAPO is a technique [255] that mines frequent API usage patterns for recommendation. Still other code search techniques exist [80, 36, 159, 136] that look at code completion. For example, the technique Strathcona [80] accepts the developer's unfinished code at hand and works on recommending relevant code for completion based on the class inheritance, method calls, and types of the partial code.

Recently, code search has been used for bug repair. The key is to find existing code pieces (from the faulty program itself, from the earlier versions of the faulty program, or from existing programs in a large code repository) that are, or are likely to be semantically correct. Then it might be possible to leverage such code to produce correct patches for bug repair. There are some existing techniques that look at doing code search for bug repair. QACrashFix [70] targets on repairing crash bugs. It uses the crash message as the query to search for similar crash issues from a Q&A site (e.g., Stack Overflow [18]). For each similar issue, it extracts and leverages the buggy and fixed code pieces associated with the issue to generate patches. Code Phage [196] does code search to find correct

code to be transferred to the faulty program for eliminating three types of errors: out-of-bounds access, integer overflow, and divide-by-zero errors. Code Phage's code search is based on program execution. For code search, it needs two inputs: one triggers the error and the other does not. Given the two inputs, Code Phage finds applications from a code database that can successfully process the two inputs. SearchRepair [97] looks for code fragments within a large code repository that is likely (but not guaranteed) to be semantically correct. Using the test cases, SearchRepair performs fault localization to identify a faulty code fragment from the faulty program and generates the input-output profile for this buggy code fragment as the specification. SearchRepair extracts code fragments in comparable sizes from the code repository. For each code fragment, it does symbolic execution to yield constraints for encoding the semantics of the code fragment. For code search, SearchRepair does constraint-solving (by employing a SMT constraint-solver) to find code fragments that can satisfy the input-output profile generated for the buggy code fragment as the specification. SearchRepair further produces patches for the buggy code fragment using the found code fragments that satisfy the input-output profile. In fact, SearchRepair can only find code fragments that are likely to be semantically correct: For test cases that the original program passes, the input-output profile specifies the expected output should be generated. However, for test cases that the original program fails, the input-output only specifies the bad output should be avoided but does not specify what should be the expected output. In general, symbolic execution has limited expressive power, and constraint-solving can be expensive. Due to such reasons, SearchRepair was only shown to be able to repair small, IntroClass-level programs. Compared to Code Phage and SearchRepair, ssFix's syntactic code search (Section 3.3.2 in Chapter 3) is more lightweight. It extracts two types of tokens from the buggy code fragment (identified by faulty localization) and any candidate code fragment (extracted from the code database) and performs token-matching using a Boolean model and a vector space model to find candidate code fragments that are syntactically similar to the buggy code fragment. ssFix looks at using such candidate code fragments to produce patches for the buggy code fragment for bug repair. sharpFix's code search (Section 5.4.2 in Chapter 5) improves ssFix's code search by using different code search methods for local code search (within the local faulty program) and global code search (within the external code repository) and then merges the results.

Prior to our work, the problem of how to do code search for finding useful code for bug repair had not been well addressed. Code clone detection/search techniques are a very related branch of code search techniques for solving the problem. Such techniques look at finding similar code fragments as clones within or across software projects. Code clones are commonly categorized into four types as the exact clones (Type-1): identical except for any differences in whitespace and comments; the renamed/parameterized clones (Type-2): identical except for any naming differences; the near-miss clones (Type-3): similar with modifications of statement changed, added, or deleted in addition to any naming differences; and the semantic clones (Type-4): functionally similar but syntactically non-similar. In fact, code clone detection/search techniques [188, 184] can be leveraged for finding existing code fragments that are clones to the buggy code fragment. A repair technique can then

leverage the syntactic differences between the buggy code fragment and its cloned code fragment to produce patches. Current clone detection techniques can be categorized as token-based [96, 122], tree-based [239, 32, 86], graph-based [107, 125], metrics-based [86], and others [102, 87]. Many of such techniques are not well suitable for finding code from a large code database for bug repair because (1) they are used for detection but not search, so they are not scalable for repository-level code search and (2) they often set up a high "threshold" for identifying or searching for clones that are too similar to the buggy code fragment. For a buggy code fragment that is of certain size and is unique, it is not likely to find a code fragment from a code database that is too similar to it. To address (1), there are already existing techniques [99, 98, 118] that are used for clone search. To address (2), there are existing techniques [66, 81, 107, 125, 102] that can identify code fragments that are semantically or functionally similar. Dynamic techniques like [87] however are still too expensive. Recently, there emerge techniques like [103] that can search for functionally similar code in a static way. It might be worth trying these techniques to retrieve code fragments to be used for bug repair. Often times, the difference between a retrieved code fragment and a buggy code fragment does not imply any bug-fix. To effectively identify a bug-fixing difference, it is possible to leverage machine learning techniques to train a model based on any buggy code fragment and a code fragment is similar to it but contains the fix code and then use the model to identify code fragment that is useful for repair.

2.6 Generating New Tests for Differential Testing

Our patch testing technique DiffTGen (Chapter 4) identifies a patch to be overfitting through generating new test cases. Given the original faulty program and the patched program, it works on generating new test inputs that expose any semantic differences between the faulty program and the patched program. Assuming the two programs are deterministic, it is the patch that actually causes the semantic differences exposed by the input. Based on the semantic differences, DiffTGen asks an oracle to judge the correctness of the differential semantics and may further generate a new test case showing the patch is overfitting if the exposed semantics of the patched program is incorrect.

DiffTGen is related to existing techniques that generate new tests to expose differential semantics between the two programs. Evans and Saviora [63] proposed to generate new test cases for the original program and for the patched program and do cross-running for differential testing, i.e., to run the original program against the test cases generated for the patched program and to run the patched program against the test cases generated for the original program. The way DiffTGen works is to generate a target program first with certain statements marked as the coverage goals. Next it employs an existing test generator to generate test inputs that satisfy at least one of the coverage goals. Such test inputs are interesting and are likely to expose differential semantics between the original and the patched programs. Next DiffTGen runs the two programs with each of such test inputs generated to see whether any actual differential behaviors of the two programs can be detected. DiffTGen is related to TESTGEN [108], DiffGen [207] and BERT [167, 89] in producing

a target program first and then employing an external test generator to generate test inputs for the target program. Different from DiffTGen, TESTGEN and DiffGen encode the value comparisons in the target program as conditions and use the corresponding branches as the coverage goals. For example, they can generate code in the target program as `if (x!=y) {s}` and use the statement `s` as the coverage goal. If the test generator can generate a test input that exercises `s` upon execution, then the test input is guaranteed to expose the semantic difference as the different values taken by `x` and `y`. DiffTGen and BERT employ a test generator to generate test inputs that exercise any changed code between the original program and the patched program. Next they dynamically run the two programs on the generate input to identify semantic differences. Such a target program produced by BERT is the patched program. BERT employs a test generator to generate test inputs that cover any changes made in the patched program as the coverage goals. The target program produced by DiffTGen is not simply the patched program with the changes marked as the coverage goals. When a change is made on an if-condition, DiffTGen can synthesize a new if-condition with the guarded statement marked as the coverage goal. A generated test input that can cover the statement guarded by the synthesized if-condition would expose differential branch-taking behaviors. Compared to DiffTGen, TESTGEN, DiffGen, and BERT are used for identifying regressions. DiffTGen however could identify not only regressions but also a patch's other overfitting behaviors. The three techniques only report to the user any differential behaviors detected. DiffTGen does so but in addition generates actual test cases. The three techniques were tested on modified programs where the modifications were randomly seeded or human-made. DiffTGen was tested in the context of automated program repair.

There are other regression, differential or patch testing techniques that are based on symbolic execution. DiSE [179] combines static program analysis and directed symbolic execution to find inputs exercising a modification. The differential symbolic execution technique [178] uses method summaries to characterize program semantic behaviors. With the support of a theorem prover, it compares two method summaries to identify semantic differences. eXpress [208] combines dynamic symbolic execution (DSE) and path pruning to generate tests revealing program behavioral differences. KATCH [141] starts with an existing test input that has the "best" potential to cover a modification based on a defined reaching distance. Based on this input, KATCH uses either symbolic execution or definition switching to generate new inputs to cover the modified code. The shadow technique [39, 171] uses concolic execution to find test inputs uncovering the semantic differences between two programs. For each if-condition after a change point in the original program, the technique tries to find test inputs to force the original and the patched programs to have different branch-taking behaviors if the concrete executions do not reveal such behaviors. To do so, the shadow technique generates a constraint for such differing behaviors and then initiates a bounded symbolic execution to further explore any new semantic differences going forward. DiffTGen's synthesized if-statement has a similar idea, but is only applied to the changed if-condition, not all the if-conditions affected by a change. Compared to DiffTGen, the shadow technique that leverages symbolic execution can potentially capture more differing, branch-taking behaviors. However, it

is also more expensive. According to [171], it may take a few hours to finish. Compared to the above testing techniques, DiffTGen is designed and has been evaluated in the context of automated program repair. It performs differential testing, but goes one step further in producing test cases. DiffTGen is more lightweight and has been shown to work fast. Again, it can not only identify regressions but a patch's other overfitting behaviors.

DiffTGen currently uses EvoSuite [65] to generate test inputs as the test methods. Within each test method, EvoSuite uses an evolutionary algorithm to build up a sequence of method calls as the testing statements. It uses mutation and crossover operations to evolve the test methods to make them satisfy the highest coverage criterion for the unit-under-test. It is however also possible to use many other existing test generation techniques to achieve test input generation such as Randoop [170], JPF [215], SPF [175], CUTE/JCUTE [193], Korat [33], PEX [212], KLEE [38], DART [71], and CREST [37].

Chapter 3

Leveraging Syntax-Related Code for Automated Program Repair

In this chapter, we present our automated program repair (APR) technique ssFix which leverages existing code (from a code database) that is syntax-related to the context of a bug to produce patches for its repair. Given a faulty program and a fault-exposing test suite, ssFix does fault localization to identify suspicious statements that are likely to be faulty. For each such statement, ssFix identifies a code chunk (or target chunk) including the statement and its local context. ssFix works on the target chunk to produce patches. To do so, it first performs syntactic code search to find candidate code chunks that are syntax-related, i.e., structurally similar and conceptually related, to the target chunk from a code database (or codebase) consisting of the local faulty program and an external code repository. ssFix assumes the correct fix contained in the candidate chunks, and it leverages each candidate chunk it retrieved to produce patches for the target chunk. To do so, ssFix first translates the candidate chunk by unifying the names used in the candidate chunk with those in the target chunk, then matches the chunk components (expressions and statements) in the translated candidate chunk and the target chunk, and then produces patches for the target chunk based on the matched and unmatched components. ssFix finally validates the patched programs it generated against the test suite and reports the first patch whose corresponding patched program can pass the test suite.

We evaluated ssFix on 357 bugs in the Defects4J bug dataset. Our results show that ssFix successfully repaired 20 bugs with valid patches generated using a code repository containing 58,219 projects and that it outperformed five other APR techniques for Java.

3.1 Introduction

A significant fraction of current APR techniques adopt a search-based approach [74, 219, 101, 182, 132, 134, 205, 113]: they define a set of modification rules to generate a space of patches first and

then search in the space for patches that are correct. The search space is often very huge which makes searching for a correct patch difficult. Long and Rinard [133] investigated the search spaces of two state-of-the-art search-based APR techniques, SPR [132] and Prophet [134], and found that (1) their search spaces, though huge, often fail to contain a correct patch and (2) the plausible-but-incorrect patches in their search spaces are often hundreds of times as many as the correct patches. The large amount of such incorrect patches can easily block the finding of a correct one.

To address the problem, the study [133] suggests leveraging repair information beyond the test suite to create a search space that is likely to contain a correct patch and is *targeted* so that the correct patch could be effectively identified. One idea is to leverage existing code fragments to produce effective patches. We call the code fragments that contain the correct forms of expressions, statements, etc. and can be used for generating a correct patch the *repair* code fragments. GenProg assumes the faulty program itself contains the repair code fragments at the statement level for patch generation. The study by Barr et al. [31] demonstrated the feasibility of this assumption. If the repair code fragments may exist in the local program, they may also exist elsewhere in many non-local programs. The study by Sumi et al. [203] supports this assumption. They found up to 69% of the repair code fragments (in the form of code lines) can be obtained (possibly with identifier renaming) from either the local program or the non-local programs. The study is based on the UCI dataset [168] containing 13,000 Java projects. We believe it is more likely to find the repair code fragments for bug repair in smaller granularity (e.g., at the expression level) and from a larger code database (e.g., GitHub, which is huge and is still rapidly growing).

A repair code fragment could possibly exist in the faulty program itself and/or in non-local programs. The question is how to find and leverage such a code fragment to produce patches. One idea is to use semantic code search, i.e., finding code fragments that are likely to be semantically correct. However, semantic code search is often expensive and it may fail to find many repair code fragments that do not represent the correct implementation (they may contain more functional features than the correct implementation does, they may use different data types or side-effect processing mechanisms, etc.) but can be leveraged to produce a correct patch. Code Phage (CP) [196] and SearchRepair [97] are two repair techniques that use semantic code search. CP's code search relies on code execution, and it can only find code that can process the given inputs. SearchRepair uses symbolic execution to encode program semantics as constraints. Symbolic execution, however, has limited expressive power for program semantics. SearchRepair's code search is based on constraint solving which is undecidable in general and is often expensive. Currently SearchRepair was only shown to work for small C programs.

If semantic code search is still limited, the natural question would be: does syntactic code search work? The answer is yes. We propose a novel APR technique ssFix which performs syntactic code search to find and leverage existing code fragments from a codebase (which consists of the local faulty program and an external code repository) to produce patches for bug repair. We assume a repair code fragment that can be effectively leveraged for bug repair to be syntax-related (i.e., structurally similar and conceptually related) to the fault-located part of the faulty program. Intuitively, such a

repair code fragment is likely to implement a coding task similar to what is implemented in the faulty code fragment (e.g., both as iterating a list of data items to look for certain values having similar names) and implements it correctly. Compared to SearchRepair and CP, ssFix is not directly targeted at finding code fragments that are semantically correct. In fact, code fragments that contain the fix may not represent the correct implementation: they may contain more or less functional features than the correct implementation does, they may use different data types or side-effect processing mechanisms, etc. So instead, ssFix uses a lightweight syntactic code search (based on a Boolean model and a TF-IDF vector space model) to find syntax-related code fragments where a repair code fragment is likely to exist. Given such a fault-related code fragment (as a candidate code chunk), ssFix translates the code chunk by unifying the identifier names in it with those in the faulty code fragment (the target code chunk), matches the components (statements and expressions) between the two chunks, and produces patches for the target chunk based on the syntactic differences that exist between the matched and unmatched components. For a candidate chunk that is syntax-related to the target chunk, the syntactic differences are small, and the search space is largely reduced. Through experiments, we demonstrated the feasibility of the assumption on which ssFix is built and the effectiveness of ssFix for bug repair.

We evaluated ssFix on all the 357 bugs in the Defects4J dataset. Our results show that ssFix successfully repaired 20 bugs with valid patches generated. The median time for producing a patch is about 11 minutes. Compared to five other repair techniques for Java: jGenProg [145] (available at [16]) which is a Java version of GenProg [116, 74], jKali [145] (available at [16]) which is a Java version of Kali [183], Nopol (version 2015) [234] (available at [17]), HDRepair [113] (available at [9]), and ACS [232] (available at [1]), our results show that ssFix has a better repair performance. ssFix is currently available at *https://github.com/qixin5/ssFix*.

3.2 Overview

In this section, we show an overview of ssFix and explain how it works with an example. ssFix accepts as input a faulty program, a fault-exposing test suite, and a codebase consisting of the faulty program and a code repository (we used the Merobase repository [83] which contains 58,219 projects and about 2.5 million Java source files). As output, ssFix either produces a patched program that passes the test suite or nothing if it cannot find one within a given time budget. Figure 3.1 is an overview of ssFix's repair process. ssFix goes through four stages to repair a bug: *fault localization*, *code search*, *patch generation*, and *patch validation*.

We use an example to go through the four stages. The faulty method as shown in Figure 3.2 is from a faulty program (bug id: *L21*) in the Defects4J bug dataset. It accepts as parameters two calendar objects `cal1` and `cal2` and checks whether they represent the same time. The fault is at Line 8 where the 12-hour calendar field `Calendar.HOUR` is used for comparing two local hours. Given two calendar objects whose hour fields are different (e.g., one is 4 and the other is 16) but all the other fields are identical, the faulty program may treat them as identical although they represent different

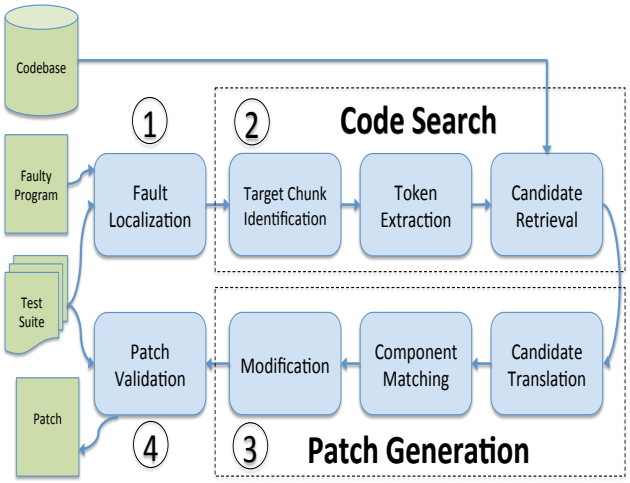

Figure 3.1: An Overview of ssFix

```
1 public static boolean isSameLocalTime(Calendar cal1, Calendar cal2) {
2 if (cal1 == null || cal2 == null){
3   throw new IllegalArgumentException(``The date must not be null'');
4 }
5 return(cal1.get(Calendar.MILLISECOND)==cal2.get(Calendar.MILLISECOND) &&
6        cal1.get(Calendar.SECOND)==cal2.get(Calendar.SECOND) &&
7        cal1.get(Calendar.MINUTE)==cal2.get(Calendar.MINUTE) &&
8        cal1.get(Calendar.HOUR)==cal2.get(Calendar.HOUR) &&
9        cal1.get(Calendar.DAY_OF_YEAR)==cal2.get(Calendar.DAY_OF_YEAR) &&
10       cal1.get(Calendar.YEAR)==cal2.get(Calendar.YEAR) &&
11       cal1.get(Calendar.ERA)==cal2.get(Calendar.ERA) &&
12       cal1.getClass()==cal2.getClass());
13 }
```

Figure 3.2: The faulty method of L21 (the faulty expression is shown in red)

times (one is early morning and one is late afternoon). For bug repair, the following modification should be made

```
- cal1.get(Calendar.HOUR) == cal2.get(Calendar.HOUR)
+ cal1.get(Calendar.HOUR_OF_DAY) == cal2.get(Calendar.HOUR_OF_DAY)
```

where the 24-hour calendar field `Calendar.HOUR_OF_DAY` is used. The modification is relatively simple, but none of the repair tools that ssFix is compared to succeeded for this bug. By leveraging existing code that is syntax-related to the bug context, ssFix successfully repaired the bug with the correct patch generated.

3.2.1 Fault Localization

In the first stage, ssFix does fault localization to identify a list of suspicious statements in the program that are likely to be faulty. The statements in the list are ranked from the most suspicious to the least. For bug repair, ssFix goes through the list: each time it looks at one statement (the *target* statement) and works on generating patches for a local code area (as a code chunk) including the statement (we will explain how to generate such a code chunk later). Currently, ssFix can only produce patches that make local changes (i.e., within the local code chunk) in the faulty program, though this may involve modifying more than one statement.

ssFix employs the fault localization technique GZoltar (version 0.1.1) [40] to identify a list of suspicious statements in the program that are likely to be faulty. The statements in the list are ranked by their suspiciousness (measured by scores) from high to low. If the testing process of the faulty program (i.e., the process of running the faulty program against the test suite) did not print any stack trace, ssFix simply uses the list of statements yielded by GZoltar as the fault localization result. Otherwise, ssFix first produces a list of statements from the printed stack traces by following each stack trace and adding to the list each statement from the fault program that is on the stack trace. ssFix then produces the fault localization result by appending the list of suspicious statements yielded by GZoltar to the list of statements it produced from the stack traces (GZoltar does not use the stack trace information to compute a statement's suspiciousness. To repair a failure which causes the stack trace to be printed, we assume the statements from the stack trace are more suspicious than the other statements in the faulty program). The faulty return statement starting at Line 5 in Figure 3.2 is ranked No. 2 in the list of suspicious statements identified by ssFix.

3.2.2 Code Search

Given a target statement identified as suspicious, ssFix goes through three steps to find syntax-related code fragments from the codebase: *target chunk identification*, *token extraction*, and *candidate retrieval*. As the first step, ssFix generates a code chunk *tchunk* including the statement itself and possibly its context. ssFix then searches for code fragments in the codebase as *cchunk*s that are syntax-related, i.e., structurally similar and conceptually related, to *tchunk*. A *tchunk* to be used as the query for code search should not be too small (e.g., including only a simple statement

```
1 GregorianCalendar calEnd = new GregorianCalendar();
2 calEnd.setTimeInMillis(end.getTime());
3 if (calStart.get(Calendar.HOUR_OF_DAY)==calEnd.get(Calendar.HOUR_OF_DAY)
4  && calStart.get(Calendar.MINUTE)==calEnd.get(Calendar.MINUTE)
5  && calStart.get(Calendar.SECOND)==calEnd.get(Calendar.SECOND)
6  && calStart.get(Calendar.MILLISECOND)==calEnd.get(Calendar.MILLISECOND)
7  && calStart.get(Calendar.HOUR_OF_DAY)==0
8  && calStart.get(Calendar.MINUTE)==0
9  && calStart.get(Calendar.SECOND)==0
10 && calStart.get(Calendar.MILLISECOND)==0
11 && start.before(end))
12    return true;
```

Figure 3.3: A candidate code chunk retrieved from the Merobase repository (the fix expression is in purple). The chunk's enclosing method *isAllDay* checks whether the two time values obtained by *start.getTime()* (not shown) and *end.getTime()* both as milliseconds represent the starting time of two days (from 00:00 of one day to 00:00 of the next day). The full class name of the chunk is *org.compiere.util.TimeUtil*.

as `return x`) because it does not include enough context. On the other hand, it should also not be large. The study by Gabel and Su [67] shows that a code fragment with more than 40 tokens can be too unique in general to have similar code fragments retrieved for code search at the repository level. Based on this result, we develop a simple chunk generation algorithm (Algorithm 1 in Section 3.3.2) to generate a *tchunk* including the target statement and its local context if the statement is not too large (to determine its size, we use a threshold based on the LOC of the statement). For our example, ssFix uses this algorithm to produce a *tchunk* with only the return statement included.

As the second and third steps in code search, ssFix extracts the structural k-gram tokens and the conceptual tokens from *tchunk* and invokes the Apache Lucene search engine [2] to do a document search to obtain a list of indexed code fragments (treated as documents) from the codebase. The retrieved list of code fragments (which we call the *candidate* code chunks, or *cchunk*s) are ranked from the ones that are the most syntax-related to *tchunk* to the least (measured by the scores computed by Lucene's default TF-IDF model from high to low). Later, ssFix goes through the list and leverages each *cchunk* to produce independent patches for *tchunk*. More details can be found in Section 3.3.2. The retrieved *cchunk* shown in Figure 3.3 is what ssFix later uses to produce a correct patch for *tchunk*. This *cchunk* is ranked No. 6 among all the retrieved chunks.

3.2.3 Patch Generation

ssFix leverages a candidate chunk *cchunk* to produce patches for *tchunk* in three steps: *candidate translation*, *component matching*, and *modification*. *tchunk* and *cchunk* may use different identifier names for variables, fields, types, and methods that are syntactically (and semantically) related. For example, the two chunks in Figure 3.2 and in Figure 3.3 use different names: `cal2` and `calEnd` for a related variable. As the first step, ssFix translates *cchunk* (if it was retrieved from a non-local program) by unifying the identifier names in *cchunk* with those that are syntactically related in

tchunk. Without such a translation, ssFix would often fail to directly use statements and expressions from *cchunk* to produce patches for *tchunk*: the patched program could simply fail to compile for using unrecognized names. We developed an heuristic algorithm (Algorithm 2 in Section 3.3.3) which ssFix uses to match variables, fields, types, and methods between *tchunk* and *cchunk* based on how they are used in the two chunks. ssFix then renames the variables, fields, types, and methods in *cchunk* to their matched counterparts in *tchunk* to achieve the translation. For our example, ssFix determines `calStart` to match `cal1` and `calEnd` to match `cal2` based on pattern-matched expressions like the following three pairs.

```
cal1.get(Calendar.MILLISECOND) == cal2.get(Calendar.MILLISECOND)
calStart.get(Calendar.MILLISECOND) == calEnd.get(Calendar.MILLISECOND)

cal1.get(Calendar.SECOND) == cal2.get(Calendar.SECOND)
calStart.get(Calendar.SECOND) == calEnd.get(Calendar.SECOND)

cal1.get(Calendar.MINUTE) == cal2.get(Calendar.MINUTE)
calStart.get(Calendar.MINUTE) == calEnd.get(Calendar.MINUTE)
```

ssFix creates a translated version of *cchunk* as *rcchunk* by renaming the two variables `calStart` and `calEnd` to their respective matched ones `cal1` and `cal2` in *tchunk*.

The translated chunk *rcchunk* may not represent the correct patch but may contain the correct forms of components (expressions and statements) to be used in *tchunk* or indirectly suggest a faulty statement in *tchunk* to be deleted for producing a correct patch. Instead of replacing *tchunk* with *rcchunk* at the chunk level for patch generation, ssFix matches components that are syntactically related between the two chunks and produces patches based on the syntactic differences that exist between the matched components and unmatched components. Specifically, ssFix uses a modified version of the tree matching algorithm used by ChangeDistiller [64] to do component matching, and it modifies *tchunk* to produce patches using three types of operations: *replacement*, *insertion*, and *deletion*. More details can be found in Section 3.3.3. For our example, ssFix found the following pair of components (and 26 others) from *tchunk* and *rcchunk* to match.

```
cal1.get(Calendar.HOUR) == cal2.get(Calendar.HOUR)
cal1.get(Calendar.HOUR_OF_DAY) == cal2.get(Calendar.HOUR_OF_DAY)
```

In *tchunk*, it then replaces the first component with the second (from *rcchunk*) to produce the correct patch.

3.2.4 Patch Validation

For each *cchunk*, ssFix produces a set of patches. It filters away patches that are syntactically redundant (for such patches, the corresponding patched programs generated are syntactically equivalent) and patches that have been tested earlier (generated by other *cchunk*s). ssFix next sorts the filtered patches based on the modification types and the modification sizes to make a correct patch

Table 3.1: The Four Metrics Associated with a Statement for Fault Localization

Count	Description
$n_{00}(s)$	the number of passing runs where s is not covered
$n_{01}(s)$	the number of failing runs where s is not covered
$n_{10}(s)$	the number of passing runs where s is covered
$n_{11}(s)$	the number of failing runs where s is covered

likely to be found before an overfitting patch (such a patched program can pass the test suite but does not actually repair the bug). More details can be found in Section 3.3.4. ssFix reports the first patched program that passes the test suite. If no such program can be found, ssFix looks at the next *cchunk* from the retrieved list and repeats the patch generation and patch validation processes. For our example, ssFix successfully found the correct patch after validating 202 individual patched programs that failed the testing (the majority of those simply failed the fault-exposing test case). It took ssFix less than seven minutes to find this patch.

3.3 Methodology

In this section, we elaborate on the four stages that ssFix takes to do bug repair.

3.3.1 Fault Localization

ssFix uses the approach described in Section 3.2.1 to do fault localization to identify a list of suspicious statements in the faulty program that are likely to be buggy. ssFix assumes the bug to be contained in one of these statements, and in the later stages, it looks at each suspicious statement in the list to produce patches independently. (To repair a complex bug, a repair technique may need to look at multiple statements within multiple different methods of the faulty program and make changes for all of them to produce a patch. Currently ssFix cannot do that type of repair, though it is possible for ssFix to make multiple changes for more than one statement within a small local code chunk, see Section 3.3.3.)

For fault localization, ssFix uses an existing spectrum-based fault localization technique GZoltar [40] (version 0.1.1). GZoltar calculates the suspiciousness of a statement (i.e., a score representing the likelihood of the statement being faulty) based on the coverages of the statement in the testing runs of the faulty program (in a testing run, GZoltar runs the faulty program against a test case). More specifically, to do fault localization, GZoltar first instruments the faulty program and then runs the instrumented faulty program against the test suite. After doing so, for each statement s, it obtains four counts: $n_{00}(s)$, $n_{01}(s)$, $n_{10}(s)$, and $n_{11}(s)$ as described in Table 3.1. $n_{00}(s)$ is the number of all passing runs where s is not covered (in a passing run, the faulty program passes the test case). $n_{01}(s)$ is the number of all failing runs where s is not covered (in a failing run, the faulty program fails the test case). $n_{10}(s)$ is the number of all passing runs where s is covered. $n_{11}(s)$ is the number of all the failing runs where s is covered. Based on the four counts, GZoltar uses the

Ochiai formula [22] as shown below to calculate a suspiciousness score $susp(s)$ which ranges from 0 to 1 inclusive with 1 being the most suspicious and 0 being not suspicious at all. (GZoltar does not calculate a suspiciousness score for a statement s if there is no failing run or s is not covered by any testing run. This makes sure the denominator of the formula is not zero.)

$$susp(s) = \frac{n_{11}(s)}{\sqrt{(n_{11}(s) + n_{01}(s)) \times (n_{11}(s) + n_{10}(s))}}$$

Intuitively, a statement is more suspicious than another statement if it is covered in more failing runs but less passing runs.

3.3.2 Code Search

The code search stage of ssFix starts with a target statement s identified as suspicious in the first stage. ssFix generates a local code chunk *tchunk* including s itself and possibly the local context of s. ssFix then extracts the structural and conceptual tokens from the text of *tchunk*. ssFix treats the extracted tokens as a vector of terms and uses Lucene's Boolean Model and its TF-IDF vector space model to find candidate code chunks *cchunk*s that are syntax-related to *tchunk* from the codebase.

Chunk Generation

Algorithm 1 Generating a Local Target Code Chunk

```
Input: s, th                                   ▷ s: target statement, th: LOC (we use 6)
Output: tchunk                                 ▷ A target code chunk
 1: function ChunkGen(s,th)
 2:   tchunk ← {s}
 3:   if getSize(tchunk) ≥ th then return tchunk
                                               ▷ getSize returns the LOC of a code chunk
 4:   s0 ← get the parent statement of s
 5:   if s0 exists then
 6:     tchunk0 ← {s0}
 7:     if getSize(tchunk0) ≤ th then return tchunk0
 8:   s1 ← get the statement before s in its block
 9:   s2 ← get the statement after s in its block
10:   if both s1 and s2 exist then
11:     tchunk1 ← {s1, s, s2}
12:     if getSize(tchunk1) ≤ th then return tchunk1
13:   else if s1 exists but s2 does not exist then
14:     tchunk2 ← {s1, s}
15:     if getSize(tchunk2) ≤ th then return tchunk2
16:   else if s1 does not exist but s2 exists then
17:     tchunk3 ← {s, s2}
18:     if getSize(tchunk3) ≤ th then return tchunk3
19:   else
20:     return tchunk
```

A *tchunk* with some context of s included could provide information about what s intends to do with the semantics potentially common to a large amount of existing code fragments in the codebase. Although it is often necessary to include some context of s (especially when s is too simple as `return x` for example), it can be a bad idea to include a large context (e.g., a whole method that implements multiple tasks). As the study [67] shows, for repository code search, significant syntactic redundancies were observed for code containing only up to 40 tokens (or 5-7 lines approximately).

A larger code fragment is likely to be too unique. Based on this observation, we developed a simple algorithm *chunkgen* (Algorithm 1) which generates a *tchunk* including s possibly with its local context: if the size of s is equal to or larger than a chunk-size threshold th (6 LOC), ssFix simply produces a *tchunk* including s itself (Lines 2-3 in Algorithm 1). Otherwise, ssFix produces a *tchunk* including either the enclosing parent statement of s up to the declared method (not inclusive), if any exists (Lines 5-7), or a maximum of s and its two neighboring statements (Lines 8-18) as long as the size of *tchunk* is no larger than th.

The way ssFix generates *cchunk*s is similar: For each Java source file in the codebase, ssFix looks at every method defined in every class defined in the file. It extracts the following code fragments within the method as *cchunk*s: (1) every compound statement which contains children statements and (2) every sequential three statements within each code block (e.g., a body block of a for-statement). (Note that for any compound statement which has a non-block single statement as its body, ssFix will create a new block as the body containing the statement. Also note that if a code block contains no more than three statements, all the statements are then included in the chunk). ssFix produces a *cchunk* using (1) and (2) to cover the two cases it produces a *tchunk* using the target statement's parent statement and the target statement itself plus its neighboring statements. Note that ssFix does not use any chunk-size threshold to produce a *cchunk*. This makes ssFix be able to find a *cchunk* that is smaller or larger than *tchunk* (for statement deletion and insertion).

Token Extraction

Given either *tchunk* or *cchunk*, ssFix extracts the structural k-gram tokens and the conceptual tokens from the text of the chunk. For every generated *cchunk* in the codebase, ssFix employs Lucene [2] to create an index for the extracted tokens to facilitate code search. Given *tchunk*, ssFix searches in the codebase for *cchunk*s that have "similar" tokens using Lucene's Boolean model and its TF-IDF vector space model.

Extracting the structural k-gram tokens: ssFix first tokenizes the text of a chunk and gets a list of tokens. To mask names, number constants, and literals that are program specific, ssFix symbolizes different types of tokens: ssFix uses the symbol v for non-JDK variables and fields, t for non-JDK & non-primitive types, m for non-JDK methods, lb for boolean literals (`true` or `false`), ln for number constants, and ls for string literals that contain whitespace characters (e.g., as an exceptional message). ssFix does not symbolize JDK tokens, primitive types, character literals, or string literals that do not contain whitespace characters since they are often semantics-indicative. We call the symbolized tokens the *code pattern tokens* and we call the string of these tokens concatenated by single spaces the *code pattern*. ssFix next splits the list of code pattern tokens into sub-lists by curly brackets and semicolons to avoid generating k-grams that are not very interesting (e.g., a k-gram that starts at the end of one statement but ends at the start of another). Finally, ssFix concatenates (with no space in between) every sequential k (we set k=5) tokens within every sub-list of tokens to get the structural k-gram tokens. (Note that if a sub-list contains less than k tokens, ssFix would produce a less-than-k-gram token.) As an example, for the expression

shown at Line 6 in Figure 3.2, ssFix produced in total 13 structural tokens (as 5-grams in angle brackets) as shown below.

```
The Expression:
cal1.get(Calendar.SECOND)==cal2.get(Calendar.SECOND)

The Structural Tokens (13 in total):
<$v$.get(Calendar>, <.get(Calendar.>, <get(Calendar.SECOND>,
<(Calendar.SECOND)>, <Calendar.SECOND)==>, <.SECOND)==$v$>,
<SECOND)==$v$.>, <)==$v$.get>, <==$v$.get(>,
<$v$.get(Calendar>, <.get(Calendar.>, <get(Calendar.SECOND>,
<(Calendar.SECOND)>
```

To produce such structural tokens, ssFix splits the expression into a list of tokens, symbolizes the tokens by changing `cal1` and `cal2` to `$v$` (note that ssFix does not symbolize `get`, `Calendar`, and `SECOND` since they are from the JDK library), and produces a list of 13 5-gram tokens.

Extracting the conceptual tokens: Two chunks that are conceptually related often use common tokens such as "time", "iterator", or "buffer". ssFix extracts such conceptual tokens as follows: ssFix first tokenizes the text of a chunk and gets a list of tokens containing *Java identifiers* only. For any token that is camel-case or contains underscores or numbers, ssFix splits the token into smaller tokens and appends them to the list. ssFix finally changes each token in the list into lower-case. For each lower-case token in the list, ssFix creates a stemmed token using the Porter Stemming algorithm [180]. If the stemmed token is different from the original token, ssFix inserts it into the list after the original token. Finally, ssFix eliminates any tokens whose string lengths are less than 3 or greater than 32 as well as the stop words and the Java keywords. For example, the list of conceptual tokens for the expression shown at Line 6 in Figure 3.2 is {"cal1", "cal", "calendar", "second", "cal2", "cal", "calendar", "second"} (note that "get" is a stop word that is eliminated).

Candidate Retrieval

For candidate retrieval, ssFix invokes Lucene's query search with the query tokens being the extracted tokens from *tchunk*[1]. It uses Lucene's default TF-IDF vector space model which uses *Lucene's Practical Scoring Function* shown below (as defined in [11]) to retrieve *cchunk*s,

$$overlap(tts, cts) \cdot qnorm(tts) \cdot \sum_{t \in tts} (tf(t \in cts) \cdot idf(t)^2 \cdot getBoost(t) \cdot norm(t, cts))$$

where t is a token (either a structural or a conceptual token), tts is the token list of *tchunk*, cts is the token list of *cchunk*, $overlap(tts, cts)$ is a score factor based on how many tokens in tts are found in cts, $tf(t \in cts)$ is the token frequency of t that appears in cts, $idf(t)$ is the inverse document frequency of t. $qnorm$, $getBoost$, and $norm$ are the normalization and boosting functions defined in

[1]It is also possible to invoke Lucene's query search twice using the structural tokens and the conceptual tokens independently and then merge the results, but we did not experiment this.

Lucene's default model. The retrieval process ignores any *cchunk* whose number of matched tokens (the tokens that are matched with those in *tchunk*) is less than $n/8$ where n is the total number of tokens in *tchunk*. To do so, ssFix uses Lucene's Boolean model.

For each *tchunk*, ssFix obtains a list of *cchunk*s that have the highest relatedness scores ranked from high to low. Currently, it only looks at the top 100 (at most) *cchunk*s that are not syntactically redundant for bug repair.

3.3.3 Patch Generation

In this stage, ssFix leverages a candidate chunk *cchunk* to produce patches for *tchunk* in three steps: *candidate translation*, *component matching*, and *modification*.

Candidate Translation

Algorithm 2 Creating an Identifier Mapping

Input: *tchunk*, *cchunk*
Output: *imap*[*idBind* → *idBind*] ▷ *idBind* is an identifier binding
1: *imap*[*idBind* → *idBind*]← empty
2: *cmap*[(*idBind*, *idBind*)→ *int*]← empty
3: *tcompts*, *ccompts* ← get the lists of non-trivial components from *tchunk* & *cchunk* (components visited in pre-order in the ASTs of *tchunk* & *cchunk*)
4: ▷ Non-trivial components do not include number constants, literals, or identifiers
5: *matched_compts* ← match components (one-to-one) between *tcompts* and *ccompts* by code pattern equality
6: **for all** (*tcompt*, *ccompt*)∈ *matched_compts* **do**
7: *tptokens*, *cptokens* ← get the code pattern tokens of *tcompt*, *ccompt*
8: ▷ *tptokens* & *cptokens* are two lists having identical elements
9: **for all** (*tptoken*, *cptoken*)∈ (*tptokens*, *cptokens*) at every list index **do**
10: **if** *tptoken* and *cptoken* are both identifier symbols **then**
11: *tidbind* ← get the identifier binding of *tptoken*
12: *cidbind* ← get the identifier binding of *cptoken*
13: **if** (*cidbind*, *tidbind*) is an entry in *cmap* **then**
14: *c* ← *cmap*.get(*cidbind*, *tidbind*)
15: *cmap*.add((*cidbind*, *tidbind*), *c* + 1)
16: **else**
17: *cmap*.add((*cidbind*, *tidbind*), 1)
18: **for all** *cidbind* from *cmap* **do**
19: *tidbind* ← get the mapped identifier with the max value of *c* (tie breaking by the Levenshtein Similarity between identifier strings)
20: *imap*.add(*cidbind*, *tidbind*)
21: **return** *imap*

A candidate chunk *cchunk* and the target chunk *tchunk* may use different identifier names for variables, fields, types, and methods that are syntactically and semantically related, especially when they are not from the same program (for example, the two chunks in Figure 3.2 and in Figure 3.3 use different names: `cal2` and `calEnd` for a related variable). We developed an heuristic algorithm shown in Algorithm 2 to map variable, field, type, and method identifiers appeared in *cchunk* to those in *tchunk* that are syntactically related (and may thus be semantically related) based on matching the code patterns of their contexts. (The code pattern used here is identical to what we defined in Section 3.3.2 but with all non-JDK identifiers, number constants, and literals symbolized to increase matching flexibility). Given a *cchunk* that is not from the local, faulty program (where *tchunk* is from), ssFix uses the algorithm to match their identifiers and renames every identifier in *cchunk* (which has a match) as its matched identifier in *tchunk* to get a translated version of

cchunk as *rcchunk*. Since a *cchunk* and a *tchunk* from the same faulty program use identifier names consistently, ssFix does not create a translated version for such a *cchunk*.

Algorithm 2 accepts as input *tchunk* and *cchunk*. It outputs an identifier mapping *imap* that maps an identifier that appears in *cchunk*, as a reference binding (or a binding), to an identifier that appears in *tchunk*, also as a binding. (ssFix matches and renames all identifiers that have the same binding consistently.) The algorithm starts by collecting in pre-order a list of components (statements and expressions) in the tree structure of the chunk (for either *tchunk* or *cchunk*) that are non-trivial, i.e., that are not number constants, literals (*boolean*, *null*, *character*, and *string* literals), or identifiers (Line 3). These components represent all the contexts of all the identifiers in the chunk. The algorithm then matches the components (Line 5) by comparing their code patterns. One component in *tchunk* can match at most one component in *cchunk*, and one component in *cchunk* can match at most one component in *tchunk*. Two components can match if and only if their code patterns (as two strings) are identical. For every matched components whose code patterns are identical (and thus share an identical list of code pattern tokens), the algorithm obtains the two lists of code pattern tokens (Line 7). At every index where the two code pattern tokens are both identifiers, the algorithm gets the identifier bindings, matches them, and saves this match with a count in a map *cmap* (Lines 9-17). For example, ssFix finds the following two components to match. It then creates two matches between `cal1` and `calStart` and between `cal2` and `calEnd` with a count saved for each.

```
cal1.get(Calendar.MILLISECOND) == cal2.get(Calendar.MILLISECOND)
calStart.get(Calendar.MILLISECOND) == calEnd.get(Calendar.MILLISECOND)
```

Finally, the algorithm iterates *cmap*, for each identifier binding in *cchunk* (*cidbind*), it finds its matched identifier binding in *tchunk* (*tidbind*) with the maximum matching count. If there are more than one such matched *tidbind*s, the algorithm breaks the ties by comparing the string similarity of the identifier bindings (Lines 18-20).

Component Matching

ssFix matches components between *tchunk* and *rcchunk* to identify their syntactic differences at the component level. Later it leverages the syntactic differences that exist between the matched and unmatched components to produce patches for *tchunk*. ssFix extends the tree matching algorithm of ChangeDistiller (as shown in Figure 3.4 copied from Fig 9. in [64]) to do component matching based on the component types, structures, and contents. The original algorithm performs tree matching at the statement level and is used for code evolutionary analysis. The algorithm used by ssFix follows its basic idea to match leaf nodes first (using the $match_1$ function as shown in Figure 3.5 copied from Section 3.4 in [64]) and then inner nodes (using the $match_2$ function as shown in Figure 3.6 copied from Section 3.4 in [64]) in a bottom-up way. We make changes to the original algorithm on the definitions of leaf and inner nodes, node compatibility, and node similarity.

Specifically, we define a *leaf* node to be either a simple statement which has no children statements or an expression that is not a number constant, a literal, or an identifier. We define an *inner* node

1: **Input**: trees T_1, T_2
2: **Result**: final matching set: M_{final}
3: $M_{\text{final}} \leftarrow \phi$, $M_{\text{tmp}} \leftarrow \phi$
4: Mark all nodes in T_1 and T_2 "unmatched"
5: **for all** leaf $x \in T_1$ and leaf $y \in T_2$ **do**
6: **if** $match_1(x, y)$ **then**
7: $M_{\text{tmp}} \leftarrow M_{\text{tmp}} \cup (x, y, sim_{2g}(v(x), v(y)))$
8: **end if**
9: **end for**
10: Sort M_{tmp} into descending order, according to the leaf-pair-similarity
11: **for all** leaf-pair-similarity $(x, y, sim_{2g}(v(x), v(y))) \in M_{\text{tmp}}$ **do**
12: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
13: Remove all leaf-pairs from M_{tmp} that contain x or y
14: Mark x and y "matched"
15: **end for**
16: Proceed post-order on trees T_1 and T_2:
17: **for all** unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ **do**
18: **if** $match_2(x, y)$ (incl. dynamic threshold and inner node similarity weighting) **then**
19: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
20: Mark x and y "matched"
21: **end if**
22: **end for**

Figure 3.4: ChangeDistiller's Tree Matching Algorithm

$$match_1(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise,} \end{cases}$$

Figure 3.5: The $match_1$ function used in Figure 3.4

$$match_2(x, y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise,} \end{cases}$$

Figure 3.6: The $match_2$ function used in Figure 3.4

Table 3.2: Compatibility Rules for Certain Types of Leaf Nodes (two leaf nodes that have the same component type may need to satisfy the specified rule for matching)

Component Type	**Rule**
ArrayAccess	Compatible array types
ArrayCreation	Compatible array types
ClassInstanceCreation	Compatible class types
InfixExpression	Same operator
PostfixExpression	Same operator
PrefixExpression	Same operator
MethodInvocation	Same method name
Assignment	Same assignment operator

[a] An array type is incompatible with a non-array type.
[b] Two array types are compatible iff (a) the array dimensions are equal and (b) the element types are equal.
[c] Two class types are compatible if they are equal.

to be a compound statement that has children statements.

ChangeDistiller uses the two functions $match_1$ and $match_2$ (as shown in Figure 3.5 and Figure 3.6) to match leaf nodes and inner nodes respectively. For node matching, the two functions both look at node compatibility (the l functions used in $match_1$ and $match_2$) and node similarity (the sim_{2g} functions used in $match_1$ and $match_2$ and the overlap similarity $\frac{|common(x,y)|}{max(|x|,|y|)}$ used in $match_2$). We give a new definition for the node compatibility (the l functions) as follows.

1. A leaf node is not compatible with an inner node.

2. Two leaf nodes are compatible if (a) their node types are equal (e.g., both as return statements) and (b) they follow the node-type-specific rules as shown in Table 3.2.

3. Two inner nodes are compatible if their node types are equal or they are both loop statements (for, while, or do statements).

ChangeDistiller uses different similarity metrics in $match_1$ and $match_2$ to measure node similarities.

- For leaf nodes, the similarity metric used in $match_1$ looks at their bigram string similarity (the sim_{2g} function). Two leaf nodes can only match if the similarity score is above some threshold f. In [64], f is set as 0.6.

- For inner nodes, the similarity metric used in $match_2$ looks at the overlap similarity of their children nodes and the bigram string similarity of their conditions (as either if-conditions or loop conditions). A threshold t is used for the overlap similarity and the threshold f is used for the bigram string similarity. In [64], f is 0.6, and different values (0.6, 0.8, and 0.4) are used for t under different conditions.

We make changes to the similarity metrics used in $match_1$ and $match_2$:

```
if (x) { s0; }                //Buggy Statement
if (y) { s1; } else { s2; } //Candidate Statement
```

```
1. if (y) { s1; } else { s2; }
2. if (y) { s0; }
3. if (x) { s1; }
4. if (x) { s0; } else { s2; }
5. if (x && y) { s0; }
6. if (x || y) { s0; }
```

Figure 3.7: The Six Patches Generated for Two Matched If-Statements

1. We decrease the values of f and t, and we use the same, decreased value of t for all cases.

2. We ignore the bigram string similarity part for the similarity metric in $match_2$.

ChangeDistiller was designed to match nodes that are highly similar for evolutionary analysis. In our context, we decrease the thresholds f and t to allow components that are syntactically related but are not highly similar to match, and we use the same decreased value of t for all cases for inner node matching. Currently ssFix uses 0.2 for f and 0.4 for t and it works reasonably well with these thresholds for our experiments. We do not consider the similarity of two conditions (as if-conditions or loop conditions) as a factor to match two compound statements (as two inner nodes) because a bug could make one condition dissimilar to the other. In such case, we still allow the two statements to match as long as they have similar children according to the overlap similarity used in the $match_2$ function in [64] so that the faulty condition has a chance of being repaired.

Modification

In the final step of patch generation, ssFix modifies *tchunk* based on the matched and unmatched components between *tchunk* and *rcchunk* to yield an initial set of patches using three types of modifications: *replacement*, *insertion*, and *deletion*. We next discuss each in turn.

Replacement: For every matched components (*tcpt*, *ccpt*) where *tcpt* is a component from *tchunk* and *ccpt* is a component from *cchunk*, ssFix replaces *tcpt* with *ccpt* and the sub-components of *tcpt* with the sub-components of *ccpt* to produce patches (it does not actually do the replacement if the component to be replaced is syntactically identical to the one as the replacement). Specifically, ssFix first replaces *tcpt* with *ccpt* to produce a patch if *tcpt* is not syntactically identical to *ccpt*. ssFix may do more replacements on the sub-components of *tcpt* and *ccpt* based on their types following the rules we created in Table 3.3. (Recall that if *tcpt* matches *ccpt*, either they have the same component type or they are both loop statements.) For each row in Table 3.3, there is more than one rule. ssFix follows the rules to produce patches independently: each time, it follows one rule to produce one patch (it would not produce a patch if the replacement makes no actual syntactic changes.) For example, ssFix produces six patches for the two if statements shown in Figure 3.7 (suppose `x`, `y`, `s0`, `s1` and `s2` are all different). Note that ssFix may make multiple changes using one replacement. For

Table 3.3: Sub-Component Replacement Rules for Certain Types of Matched Components

Component	**Rule**
If Statements	1. Replace condition 2. Replace then-branch 3. Replace else-branch 4. Combine conditions with && 5. Combine conditions with ||
For Statements	1. Replace initializers 2. Replace condition 3. Replace updaters 4. Replace initializers, condition, & updaters 5. Replace body
Loop Statements (not both as for-statements)	1. Replace condition 2. Replace body
Switch Statements	1. Replace expression 2. Replace body
Try Statements	1. Replace try-body 2. Replace catch-clauses 3. Replace finally-body
Synchronized Statements	1. Replace synchronized expression 2. Replace body
Return Statements (with *boolean* returned expressions)	1. Combine the expressions with && 2. Combine the expressions with ||
Catch Clauses	1. Replace caught exception 2. Replace body
Assignments/Infix Expressions	1. Replace left-hand side 2. Replace operator 3. Replace right-hand side
Method Calls/Super Method Calls	1. Replace caller expression 2. Replace method name 3. Replace arguments*
Constructor Calls (i.e., this(...))	1. Replace arguments*
Super Constructor Calls	1. Replace caller expression 2. Replace arguments*
Prefix/Postfix Expressions	1. Replace operator 2. Replace operand

* ssFix may produce multiple patches by replacing each individual argument of *tcpt* with the corresponding argument of *ccpt* in the same argument index. This only happens when the two components have the same number of arguments.

example, it may follow Rule 2 for loop statements to replace a loop body with another which may make changes to several statements within the body.

Algorithm 3 The Insertion Algorithm

```
Input: cstmt, cptmap[ccpt → tcpt]          ▷ cptmap maps rcchunk's component to tchunk's component
Output: plist                               ▷ a list of patches
 1: plist ←{}
 2: if cptmap.get(cstmt)!=null then
 3:    return plist
 4: cstmt_children ← get all children statements of cstmt
 5: for all cstmt_child ∈ cstmt_children do
 6:    if cptmap.get(cstmt_child)!=null then
 7:       return plist
 8: cstmt_siblings ← get the sibling statements of cstmt from its parent block
 9: cstmt_i ← get the index of cstmt in cstmt_siblings
10: cstmt_siblings_size ← get the size of cstmt_siblings
11: csl ← null, csh ← null
12: for all i from 0 to cstmt_i − 1 do
13:    cstmt_sibling ← cstmt_siblings.get(i)
14:    if cptmap.get(cstmt_sibling)!=null then
15:       csl ← cstmt_sibling
16: for all i from cstmt_i + 1 to cstmt_siblings_size − 1 do
17:    cstmt_sibling ← cstmt_siblings.get(i)
18:    if cptmap.get(cstmt_sibling)!=null then
19:       csh ← cstmt_sibling
20: tsl ← cptmap.get(csl), tsh ← cptmap.get(csh)
21: if tsl!=null && tsh!=null && tsl and tsh are from the same block then
22:    tsiblings ← get all the sibling statements of tsl (or tsh)
23:    i ← get the index of tsl in tsiblings
24:    j ← get the index of tsh in tsiblings
25:    l ← min(i, j); h ← max(i, j)
26:    for all k from l to h − 1 do
27:       p ← create a patch by inserting s after tsiblings.get(k)
28:       plist.add(p)
29: else if tsl!=null then
30:    tsiblings ← get all the sibling statements of tsl
31:    i ← get the index of tsl in tsiblings
32:    for all k from i to tsiblings.size() − 1 do
33:       p ← create a patch by inserting s after tsiblings.get(k)
34:       plist.add(p)
35: else if tsh!=null then
36:    tsiblings ← get all the sibling statements of tsh
37:    i ← get the index of tsh in tsiblings
38:    for all k from i to 0 (decreasing) do
39:       p ← create a patch by inserting s before tsiblings.get(k)
40:       plist.add(p)
41: return plist
```

Insertion & Deletion:

ssFix does insertions to insert unmatched statement components from *rcchunk* in *tchunk*. The algorithm is shown in Algorithm 3. It accepts a component *cstmt* (as a statement) from *rcchunk* and a map *cptmap* which maps a component from *rcchunk* to another component from *tchunk* (ssFix can simply produce such a map based on the component matching result it produced earlier). As output, it produces a list of patches generated by insertion. For every component *cstmt* from *rcchunk*, ssFix uses the algorithm to possibly produce a list of patches. Given a component *cstmt* from *rcchunk*, ssFix first checks whether it is qualified for insertion, i.e., (1) whether it is a statement that is not matched (i.e., has no mapped component in *cptmap*) and (2) whether it has no matched children statements (Lines 2-7). For (2), if s is a statement that has matched children statements, ssFix ignores the insertion of s because the potential occurrence of s in *tchunk* could lead to statement redundancy caused by its children statements. If s is a qualified statement that satisfies (1) and

(2), ssFix computes estimated positions where s is likely to fit in *tchunk* and later inserts s at every estimated position to yield patches. Specifically, ssFix first finds the two sibling statements of s in its parent block (as *csl* and *csh*) that are closest to s and have matches (Lines 11-19). ssFix gets their matched statements *tsl* and *tsh* in *tchunk* (Line 20). If they both exist and are from the same block, ssFix inserts s at every position between *tsl* and *tsh* (Lines 21-28). Otherwise, if at least one of *tsl* and *tsh* exists, ssFix inserts s at every position after *tsl* in its block (Lines 29-34) and/or at every position before *tsh* in its block (Lines 35-40) to yield patches. If neither *tsl* nor *tsh* exists, ssFix ignores the insertion for s since there is no matching evidence that shows s is needed for *tchunk*.

For deletion, ssFix deletes any statement component in *tchunk* that has no matched statement in *rcchunk*. Similar to insertion, if the unmatched statement has matched children statements, ssFix ignores its deletion.

3.3.4 Patch Validation

ssFix leverages a *cchunk* retrieved from the codebase to produce a set of patches for *tchunk*. In this stage, ssFix first filters aways patches that are syntactically redundant (two patches are syntactically redundant if their generated, patched programs are syntactically identical) and patches that have been tested earlier (generated by other *cchunk*s). ssFix next sorts the patches by the modification types and sizes, then validates each patched program against the test suite, and finally reports the first one (if any) that passes the test suite. Like every repair technique that uses a test suite as the correctness criterion for patch evaluation, it is possible that ssFix produces an overfitting, patched program that passes the test suite but does not actually repair the bug. Studies [183, 152] have shown that (1) a repair technique is more likely to produce an overfitting patch using deletion than using other types of modifications and (2) a simple patch is less likely to be overfitting than a complex patch. Based on these results, we developed the algorithm used by ssFix as shown in Algorithm 4 for comparing the generated patches. Using the algorithm, ssFix ranks the generated patches to make a non-overfitting patch likely be found first.

Algorithm 4 Patch Comparison Algorithm

Input: $p_1(t_1, h_1, d_1)$, $p_2(t_2, h_2, d_2)$ ▷ t: type, h: tree height, d: edit distance
Output: *order* ▷ $order \in \{-1, 0, 1\}$
1: **if** $t_1 \in \{R, I\}$ and $t_2 \in \{D\}$ **then return** -1 ▷ R: replacement, I: insertion, D: deletion
2: **if** $t_1 \in \{D\}$ and $t_2 \in \{R, I\}$ **then return** 1
3: **if** $h_1 < h_2$ **then return** -1
4: **if** $h_1 > h_2$ **then return** 1
5: **if** $d_1 < d_2$ **then return** -1
6: **if** $d_1 > d_2$ **then return** 1
7: **return** 0

The algorithm accepts as input two patches p_1 and p_2 and outputs a numeric ordering $order \in \{-1, 0, 1\}$ showing p_1 has a rank higher than, equal to, and lower than p_2 respectively. Each patch is associated with a modification type $t \in \{R, I, D\}$ where R is replacement, I is insertion, and D is deletion, and two types of modification sizes h and d: Given a component *cpt* and its modified

version *cpt′* that a patch involves (either *cpt* or *cpt′* can be *null* for a deletion and an insertion), we compute h as the maximum heights of the tree structures of *cpt* and *cpt′* and d as the edit distance of their content strings. As the algorithm shows, a patch whose modification type is non-deletion always ranks higher than a patch whose modification type is deletion (Lines 1-2). If the types cannot differentiate them, the algorithm then looks at their tree heights (for insertion and deletion, if a component is `null`, the height is 0): the patch with a smaller h ranks higher than the other (Lines 3-4). If the tree heights still cannot differentiate them, the algorithm further looks at their edit distances (we define an edit distance of a patch to be the edit distance between the content strings of *cpt* and *cpt′*, and if a component is *null*, the content string is empty): the patch with a smaller edit distance ranks higher than the other (Lines 5-6). ssFix follows the algorithm to do patch sorting and produces a list of sorted patches for *tchunk* using *cchunk*.

Next ssFix looks at the patches in the sorted order for validation. It first applies the patch on the faulty program to produce a patched program, next checks whether the program compiles. If it compiles, ssFix next tests the patched program against the test cases that the original, faulty program failed. If the patched program succeeds, ssFix finally tests it against the whole test suite. Note that ssFix may produce hundreds of patches given a *cchunk* that is dissimilar to *tchunk*. For efficiency, ssFix only selects a maximum of the top-sorted k (we set $k = 50$) patches for validation.

3.4 Empirical Evaluation

To evaluate the performance of ssFix, we used the Defects4J bug dataset (version 0.1.0) [94] which contains a set of 357 real bugs. We ask two research questions.

- **RQ1**: How does ssFix work on repairing these 357 bugs?
- **RQ2**: Compared to other APR techniques, how effective is ssFix?

We conducted two experiments to answer them. We next show each experiment in turn.

3.4.1 RQ1

We implemented ssFix and ran it to repair all the 357 real bugs in the Defects4J bug dataset. Our results show that ssFix repaired 20 bugs with valid patches generated. The median time for generating a plausible patch is about 11 minutes. The minimum time for generating a plausible patch is about 2 minutes. The maximum time for generating a plausible patch is about 101 minutes.

Experimental Setup

Bug Dataset The Defects4J dataset (version 0.1.0) [94] as shown in Table 3.4 consists of 357 real bugs from five Java projects: *JFreeChart* (C), *Closure Compiler* (Cl), *Commons Math* (M), *Joda-Time* (T), and *Commons Lang* (L). Each bug in the dataset is associated with a developer patch showing how the bug can be correctly repaired. The dataset has been commonly used for evaluating an automated repair technique for Java [145, 234, 113, 232].

Table 3.4: The Defects4J Dataset (version 0.1.0)

Bug Projects	Abbrv	#Bugs	#Tests	#Bugs Selected
JFreeChart	C	26	2,205	6
Closure Compiler	Cl	133	7,927	38
Commons Lang	L	27	2,245	19
Commons Math	M	106	3,602	24
Joda-Time	T	27	4,130	6
Total	-	357	20,109	93

The table except the column Abbrv is copied from Table 1 in [94].

Table 3.5: All Plausible Patches Generated by ssFix

Project (#Bugs)	Time (in minutes)			#P	#V*	#Correct Sem(Syn)*	CChunk Rank			CChunk Locality		#Tested Patch		
	min	max	med				min	max	med	#L	#NL	min	max	med
C (26)	3.4	77.9	8.8	7	3	2(2)	1	65	34	2	5	2	4337	132
Cl (133)	7.6	34.8	12.8	11	2	1(1)	1	51	1	9	2	2	489	84
M (106)	2.2	100.5	14.8	26	10	7(6)	1	91	7	12	14	1	5609	171.5
T (27)	1.8	8.1	4.0	4	0	0(0)	1	24	5	1	3	3	426	61.5
L (65)	3.4	56.4	6.1	12	5	5(5)	1	60	8	1	11	3	2454	34.5
Total (357)	1.8	100.5	10.7	60	20	15(14)	1	91	6.5	25	35	1	5609	99

We show the projects in their abbreviations: C is JfreeChart; Cl is Closure Compiler; M is Commons Math; T is Joda-Time; and L is Commons Lang. #P is the number of plausible patches generated. #V is the number of valid patches generated. #L is the number of local *cchunk* (retrieved from the local faulty program) used. #NL is the number of non-local *cchunk* (retrieved from the non-local programs in the code repository) used.
* We manually compared a generated patch to the developer patch to determine its validity and correctness.

ssFix's Running Setup Our implementation of ssFix used the Merobase repository [83] which contains 58,219 projects (about 2.5 million Java source files, or about 180 million LOC) as the external code repository and five versions of the projects (C8, Cl14, L6, M33, and T4) as the local faulty programs[2]. To avoid using a fixed version of a bug to produce patches, in the code search stage, ssFix ignores any candidate chunk *cchunk* retrieved from the codebase if (1) the full-class name of *cchunk*'s located class is the same as that of the target chunk's (or *tchunk*'s) located class[3] and (2) the signature of *cchunk*'s enclosing method is the same as that of *tchunk*'s method.[4] We ran ssFix to repair each bug within a time budget of 120 minutes on machines with eight AMD Phenom(tm) II processors and 8G memory.

Results

Table 3.5 is a summary[5] of the repair performance of ssFix. From left to right, the table shows the project name and the number of bugs in the project, the repairing time (min, max, and median), the number of bugs for which plausible patches were generated (a patch is *plausible* if the patched program passes the test suite), the number of valid patches generated (we consider a patch to be *valid* if the patched program passes the test suite and does not introduce any new bugs which the

[2]For each of the three bugs: M53, M59 and M70, the fault's located class contains a repair statement. The repair statement, however, is not contained in the class of M33 whose class name is identical to that of the fault's located class. So we additionally indexed the fault's located class for each bug (three classes in total).

[3]The *Commons Lang* & the *Commons Math* projects may use either *lang3* & *math3* or *lang* & *math* as parts of their package names respectively. We unified these name differences for comparing two class names.

[4]Even doing so, we still manually found, in our initial experiments, the two *cchunk*s (for *L43* and *L33*) that ssFix used to yield patches are suspicious to be from the fixed versions of the two faulty programs. We created a black-list for the enclosing methods of those *cchunk*s.

[5]The complete result can be found at https://github.com/qixin5/ssFix/blob/master/expt0/rslt.

test suite does not expose), the number of correct patches generated (we consider a patch to be *correct* if it is semantically equivalent to the developer patch associate with the bug, in a stricter case, such a patch can be syntactically equivalent to the developer patch), the ranks (in min, max, and median from 1 to 100) of the candidate chunks used for generating the patches, the numbers of chunks retrieved from the local project and from the external code repository, and the number of failed patches ssFix created and tested (against at least one test case) before finding a plausible patch.

As shown, ssFix produced plausible patches for 60 bugs in total (a patch is plausible if the patched program passes the test suite). The running time (in minutes) for repairing these 60 bugs ranges from 1.8 to 100.5 with the median being 10.7. A plausible patch is produced and identified by ssFix automatically. To determine whether a generated, plausible patch is correct and/or valid, we manually compared the patch with the corresponding developer patch contained in the Defects4J dataset. Among the 60 plausible patches generated (for the 60 bugs), we determined 20 patches to be valid. Among the 20 valid patches, we determined 15 patches to be semantically equivalent to the developer patches associated with their repaired bugs, and 14 of the 15 patches to be not only semantically but also syntactically equivalent to the corresponding developer patches. In terms of passing the test suite without introducing any new bugs, we determined 5 patches to be valid though they are not semantically equivalent to the developer patches. Below is one such patch generated for the bug *M57*:

```
+ double sum=0; (by developer)
+ float sum=0; (by ssFix)
- int sum=0;
```

ssFix patched the program by changing the declared type of *sum* from *int* to *float* to avoid precision loss. The patched program now passes the test suite. Although the patch is not semantically equivalent to the developer patch, we consider it as valid. We manually determined 7 of the 60 plausible patches to be *defective* (and thus overfitting): they introduce new bugs to their original programs and are thus invalid and incorrect. For four of them, ssFix deleted the expected program semantics. For the remaining 33 (60-20-7) patches, it is not easy for us to manually determine their validity since the patches are not syntactically equivalent or similar to the developer patches. All the 60 plausible patches and the corresponding candidate chunks can be found under *https://github.com/qixin5/ssFix/tree/master/expt0*. For each of the 20 valid patches, we provided an explanation as to why we believe it is valid/correct.

Our results show the feasibility of leveraging existing code from a codebase to repair bugs. Section 4.2 shows an example of how ssFix finds and reuses a candidate code chunk from an external code repository to produce a correct patch. It is also possible for ssFix to leverage a candidate code chunk retrieved from the local faulty program to produce a correct patch. As an example, below is a faulty code fragment (bug id: M33) as a part of the implementation for the phase-1 stage of the simplex method. The method `compareTo` at Line 4 misuses the irrelevant threshold `maxUlps` which is intended to be used as the amount of error for floating-point comparison and is much greater than

the correct threshold `epsilon` that should have been used here for checking the general optimality.

```
for (...) {
  ...
  + if (Precision.compareTo(entry,0d,epsilon)>0) {
  - if (Precision.compareTo(entry,0d,maxUlps)>0) {
    columnsToDrop.add(i);
  }
}
```

For repair, ssFix found a candidate chunk as shown below from the local program.

```
for (...) {
  ...
  if (Precision.compareTo(entry,0d,epsilon)<0) {
    return false;
  }
}
```

It matched the two method calls `compareTo(...,maxUlps)` and `compareTo(...,epsilon)` and replaced `maxUlps` with `epsilon` to produce the correct patch.

Our results also show that ssFix is efficient in producing patches by leveraging a candidate chunk *cchunk* (retrieved from our codebase) that is syntax-related to the target chunk *tchunk*: the median rank of the candidate chunks ssFix used to produce patches is 6.5 and the median time to produce a plausible patch is only 10.7 minutes. ssFix is efficient because it leverages the syntactic differences between *tchunk* and *cchunk* to produce patches. For a *cchunk* that is syntax-related (i.e., structurally and conceptually similar) to *tchunk*, such differences are small, and the search space is much reduced.

ssFix failed 297 (357-60) bugs with no patches generated. To understand the failures, we manually examined the developer patches for all the 357 bugs and found that there are 263 complex bugs for which the correct patches are not within the search space of ssFix (recall that ssFix can currently only repair relatively simple bugs by making modifications within a relatively small code chunk). Among the 297 failed bugs, there are 221 such complex bugs for which ssFix cannot produce *correct* patches. (But note that ssFix did produce *valid* patches for 2 of the 263 complex bugs: Cl115 & M30. Each such patch makes and only makes some but not all of the changes made by the developer patch, and the corresponding patched program passes the test suite.)

Among the other 94 (357-263) simple bugs, ssFix produced plausible patches for 33 bugs, and it failed 61 bugs with no patches generated. One challenge lies in the accuracy of fault localization. We found GZoltar simply failed to identify the target faulty statements for 15 bugs (among the 61 failed ones). We also found there are 19 bugs for which the suspicious ranks of the target statements are greater than 50 (with the median rank being 159), and ssFix did not actually looked at any of these target statements under the current running setup.

Another challenge lies in ssFix's code search ability in finding effective candidates. The current

way ssFix does code search is not effective for all cases. The bug Cl10 is one example. ssFix produces a target chunk as shown below.

```
if (recurse) {
  - return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
  + return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
} else { return mayBeStringHelper(n); }
```

Since all the identifier names are locally defined by the faulty program, ssFix creates a code pattern with all the names symbolized, and extracts a list of structural tokens that are a little too general (which roughly say that the code chunk contains an if-statement and two method calls to be returned). The extracted conceptual tokens together are a little too unique to be used for finding related candidate chunks in the codebase. As a result, ssFix failed to find candidate chunks that are truly syntax-related from Merobase. The candidate chunks found from the local program however do not contain the correct expression to be used for bug repair. So ssFix failed to repair the bug. ssFix's candidate translation, component matching, and modification can also be the limitations for producing a valid/correct patch. We conducted more experiments to evaluate ssFix's code search and code reuse abilities in Section 5.3 of Chapter 5.

Since ssFix uses a test suite (as opposed to a formal specification) as the correctness criterion for patch evaluation, it can generate a defective patch which introduces new bugs. An inaccurate fault localization technique and an ineffective candidate could both lead to a defective patch being generated. We actually found that it can be problematic to produce patches by deletion using a candidate chunk that is not very related to the target chunk. ssFix produced four defective patches by deleting the non-buggy statements.

3.4.2 RQ2

We compared ssFix to five other repair techniques for Java: jGenProg [145] (available at [16]) which is a Java version of GenProg [116, 74], jKali [145] (available at [16]) which is a Java version of Kali [183], Nopol (version 2015) [234] (available at [17]), HDRepair [113] (available at [9]), and ACS [232] (available at [1]) on the same dataset. ssFix produced larger numbers of patches that are valid and correct with the efficiency of producing a plausible patch being either comparable or better. We did not compare ssFix to other repair techniques that are written for C (e.g., SearchRepair [97], Code Phage [196], SPR [132], Prophet [134], and Angelix [153]) or are not publicly available as of August, 2017 (e.g., PAR [101]).

Experimental Setup

We ran jGenProg, jKali, Nopol, HDRepair, and ACS each to repair all the 357 bugs in the Defects4J dataset on machines that have the same configurations with the ones on which we ran ssFix. The time budget for repairing a bug is two hours (the same for ssFix). Since jGenProg and HDRepair

Table 3.6: All Plausible Patches Generated by ssFix and Five Other Techniques (see Figure 3.8 for the specific bugs for which valid patches were generated by the six techniques)

Tool	Time (in minutes)			#Plausible	#Valid	#Correct	
	min	max	med			sem	syn
ssFix	1.8	100.5	10.7	60	20	15	14
jGenProg	10.8	78.5	30.5	19(27)	3	3(5)	2
jKali	4.4	81.6	8.5	18(22)	1	1(1)	1
Nopol	1.6	101.3	12.6	33(35)	0	0(5)	0
HDRepair	8.2	87.7	52.3	16(23†)	5	4(10†)	3
ACS	88.8	113.1	93.9	7(23‡)	3	3(18‡)	2

The numbers in parentheses (in the 5th and 7th columns) are copied from the results reported in [61, 113, 232] (where the reported results in [61, 232] are based on four of the five projects except the Closure Compiler project, and the reported result in [113] is based on all the five projects). Our results (not in parentheses) are based on all the five projects.

† The results reported in [113] are based on a repair experiment on 90 selected bugs using a fault localization technique performed at the method level (with a faulty method known in advance). For each bug, the authors of [113] looked for a correct patch within the top 10 generated patches (if any). Our results are based on all the 357 bugs. The fault localization was performed at the project level. For a consistent comparison, we only checked the validity and correctness of the first generated patch (if any).

‡ In our experimental setup, we found that ACS (available at [1]) took longer than what is reported in the paper [232] to produce a plausible patch, and we did not reproduce many correct patches reported in [232].

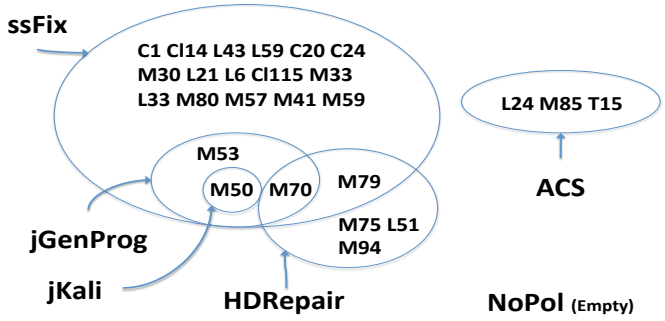

Figure 3.8: Valid Patches Generated by Different Techniques

use randomness for patch generation, we ran the tool (either jGenProg or HDRepair) in three trials[6] to repair a bug, and we considered the tool to have a valid/correct patch generated if it did so in at least one trial. For the other three techniques, we ran them each only in one trial to repair each bug.

Results

Table 3.6 shows the repairing time (min, max, and median) and the numbers of plausible, valid, and correct patches generated by all the six techniques. Figure 3.8 shows the ids of the bugs for which the techniques produced valid patches. Our results show that ssFix significantly outperforms jGenProg, jKali, and Nopol: ssFix produced many more valid patches (using either less or comparable

[6]Note that our experiment was very expensive and we only ran jGenProg/HDRepair in three trials. We believe our current running setup is sufficient to show that ssFix outperforms the two tools: the number of valid and correct patches generated by ssFix in one trial is about four times larger than the number of those patches generated by jGenProg or HDRepair in three trials.

time) than these techniques did. All the valid patches generated by these techniques were actually generated by ssFix. M50 is a bug whose fix needs the deletion of a statement. jGenProg, jKali, and ssFix all produced the correct patch by deleting the statement. For M53 and M70, jGenProg successfully found the correct statements in the original program and did insertion and replacement to produce the respective correct patches. ssFix also found the candidate code chunks including the correct statements using the local contexts of the faulty statements and produced the correct patches using the candidate code chunks. According to our results, jGenProg, jKali, and Nopol have relatively poor performances. We found they have their own limitations in repairing most of the bugs with valid patches generated. jGenProg cannot practically produce a correct patch when the repair statement does not exist in the faulty program. It deletes a statement with high probability and this often leads to a defective patch generated. jKali can only delete a statement, so it is not expected to produce any correct patch that does not involve statement deletion. Nopol uses a conditional synthesis technique to produce patches related to an if-statement. Our results show that it is prone to producing a patch with a synthesized condition being too constrained or too loose. An example (M85) is shown below.

```
if (fa * fb >= 0.0) { ... } //The faulty statement
if (fa * fb > 0.0) { ... } //The correct patch
if (fa * fb >= 1) { ... } //Nopol's patch
```

Although Nopol created a patch by constraining the original if-condition to make the test suite pass, it is overly constrained and would not be correct in general.

HDRepair is an extension of GenProg. It uses more modification operations than GenProg does but leverages the bug-fix patterns mined from existing bug-fixing instances to guide the patch search process. However, our results show that HDRepair's bug-fix-pattern driven algorithm is not truly effective. Compared to jGenProg, it only produced two more valid patches with the median repairing time being longer. Overall, ssFix outperforms HDRepair. But HDRepair did produce three valid patches that ssFix failed to produce. For example, to produce one of such patches (for M75), HDRepair reused a statement `return getPct(Long.valueOf(v))` from the class of the faulty statement `return getCumPct((Comparable<?>) v);` and applied a modification to replace `getCumPct` with `getPct`. ssFix did not find the repair statement since its local context is not similar to that of the faulty statement.

ACS is a recently developed technique that also uses condition synthesis to repair a program. It leverages techniques of test case analysis, document analysis, dependency analysis, and predicate mining to produce an if-statement with a synthesized condition that is likely to be correct. Our results show that ACS generated valid patches for three bugs that none of the other techniques successfully repaired. Below is an example (T15):

```
case -1:
+ if (val1==Long.MIN_VALUE) { //A correct patch
+     throw new ArithmeticException();
+ }
```

```
return -val1;
```

ACS analyzes the failed test case and figures out that the faulty program does not handle a boundary case where an *ArithmeticException* needs to be thrown. It creates an if-statement to throw that exception, synthesizes the condition *val1==Long.MIN_VALUE* (the boundary case), and inserts it in the faulty program to yield the patch. M85 is another example that ACS successfully repaired by synthesizing a correct if-condition as `fa*fb>=0.0 && !(fa*fb==0.0)`: it first identified a target expression (`fa*fb`), then performed keyword search over the GitHub repositories to find relevant predicates and produced the expression `!(fa*fb==0.0)`, and it finally produced the correct condition by conjoining this expression with `fa*fb>=0.0`. Through using relevant expressions from GitHub, ACS synthesized a correct condition that is neither too constrained nor too loose. Although ACS produced three valid patches that no other techniques produced, our results show that ssFix still outperforms ACS in terms of the number of valid patches generated and the repairing time. Since ACS is designed to repair bugs related to if-conditions, it is not easy for ACS to produce a direct, valid patch for bugs like L59 for which ssFix produced a correct patch by replacing a method argument `strLen` with another `width` as shown below.

```
+ str.getChars(0, width, buffer, size);
- str.getChars(0, strLen, buffer, size);
```

3.4.3 Discussion

In our experiments, we referred to the developer patch associated with a bug to manually determine the validity and correctness of a patch generated by a repair technique. There can be in general other ways to define the validity and correctness of a patch. For a fraction of plausible patches generated by ssFix and other techniques, we cannot easily determine their validity or correctness, but it is possible that some of the generated patches are valid and correct even though they are not syntactically equivalent or similar to the developer patches. Even so, we do not believe there can be a significant fraction of valid/correct patches among such plausible ones, and we released all the plausible patches at *https://github.com/qixin5/ssFix/tree/master/expt0/patch*. Though possibly biased, a manual evaluation method like ours is commonly used to evaluate the quality of patches generated by current automated repair techniques. The problem however can be mitigated through using a held-out test suite (to quantify overfitting) and/or other approaches that can identify overfitting patches (e.g., [229, 238]).

ssFix is built upon the assumption that existing code from a code database contains the *repair* code needed for producing the correct patch. In Chapter 5, we show an experiment we conducted to test the assumption to see how often it holds in practice. ssFix is a relatively complicated repair system. Its repair performance depends on its components for doing fault localization, code search, and code reuse (for patch generation and validation). In Chapter 5, we conducted experiments to evaluate ssFix's code search and code reuse abilities. ssFix relies on an existing technique GZoltar to fault localization, and we showed in this chapter (in Section 3.4.1) that GZoltar works poorly. In

Chapter 5, we conducted more experiments to see whether ssFix can work well given the fault being accurately located (at the statement and method levels). ssFix can produce overfitting patches. To possibly avoid producing such patches, ssFix can leverage techniques like [206, 244, 229, 238, 130]).

3.5 Summary

In this chapter, we presented our automated repair technique ssFix which performs syntactic code search to find existing code from a code database that is syntax-related to the context of a bug and further leverages such code to produce patches for bug repair. Our experiments have demonstrated the effectiveness of ssFix in repairing real bugs. In Chapter 5, we show the experiments we conducted to test ssFix's built-upon assumption (i.e., the repair code needed for bug repair often exists in the code database) and to evaluate ssFix's code search and code reuse abilities.

Chapter 4

Identifying Test-Suite-Overfitted Patches through Test Case Generation

A typical automated program repair (APR) technique that uses a test suite as the correctness criterion can produce a patched program that is test-suite-overfitted, or overfitting, which passes the test suite but does not actually repair the bug. In this chapter, we propose DiffTGen which identifies a patched program to be overfitting by first generating new test inputs that uncover semantic differences between the original faulty program and the patched program, then testing the patched program based on the semantic differences, and finally generating test cases. Such a test case could be added to the original test suite to make it stronger and could prevent the repair technique from generating a similar overfitting patch again. We evaluated DiffTGen on 89 patches generated by four APR techniques for Java with 79 of them being possibly overfitting and incorrect. DiffTGen identifies in total 39 (49.4%) overfitting patches and yields the corresponding test cases. With the fixed version of a faulty program being the oracle, the average running time is about 7 minutes. We further show that an APR technique, if configured with DiffTGen, could avoid yielding overfitting patches and potentially produce correct ones.

4.1 Introduction

Given a faulty program and a fault-exposing test suite (which the faulty program failed), an APR technique can generate a patched program that passes the test suite. However the patched program may not actually repair the bug. It may introduce new bugs which the test suite does not expose. Such a patched program, or the patch, is test-suite-overfitted and is called overfitting [200]. Studies [183, 200] have shown that early repair techniques suffer from severe overfitting problems. According to [183], the majority of patches generated by GenProg [74], AE [219] and RSRepair [182] are

incorrect.More recent techniques look at many other methods (e.g., using human-written patches [101], repair templates and condition synthesis [132], bug-fixing instances [134, 113], and forbidden modifications [206]) for repair. However, their repair performances are still relatively poor. Within a 12-hour time limit, the state-of-the-art repair techniques SPR [132] and Prophet [134] generated plausible patches that pass the test suite for less than 60% bugs in a dataset containing 69 bugs, with more than 60% of the plausible patches (the first found ones) being incorrect.

The low quality of a test suite is a critical reason why an overfitting patch might be generated. Unlike a formal specification, the specification encoded in a test suite is typically weak and incomplete. For example, the fault-exposing test case in the test suite associated with the bug *Math_85* from the Defects4J dataset [94] simply checks whether a method returns a result without any exception thrown, but does not check the correctness of the result. A patch generated by jGenProg (the Java version of GenProg [74]) simply removes the erroneous statement that triggers the exception without actually repairing it. The patch avoids the unexpected exception but deletes the expected functionality of the original program and thus introduces new bugs. It is not surprising that a test suite sometimes contains such weak test cases since the test suite is designed for humans but not for machines, and a human seldomly makes an unreasonable patch by deleting the desirable functionality of a program. However, such a weak test suite harms the performance of an APR technique, e.g., jGenProg. When a patched program that passes the test suite is generated, jGenProg would simply accept it as there is no extra knowledge other than the given test suite to validate its correctness.

In this chapter, we first give a formal definition of an overfitting patch, and then propose DiffTGen, a patch testing technique to be used in the context of automated program repair. DiffTGen identifies overfitting patches generated by an APR technique through test case generation. Based on the syntactic differences between a faulty program and a patched program, DiffTGen employs an external test generator to generate test methods (test inputs) that could exercise at least one of the syntactic differences upon execution. To actually find any semantic difference, DiffTGen instruments the two programs, runs the programs against the generated test method, and compares the running outputs. If the outputs are different, DiffTGen reports the difference to the oracle for correctness judging. If the output of the patched program is incorrect, DiffTGen determines the patch to be overfitting. If a correct output could be provided by the oracle, DiffTGen would produce an overfitting-indicative test case by augmenting the test method with assertion statements. (Note that it is not interesting when the running outputs are identical, since they are not related to any changes the patch makes.)

DiffTGen can be combined with an APR technique to enhance its performance. After a patch is generated by the repair technique, DiffTGen may produce a test case showing the patch is overfitting. Such a test case could be added to the original test suite to make the test suite stronger. Using the augmented test suite, the repair technique avoids yielding a category of patches that have similar overfitting properties, and could potentially produce a correct patch (See Section 4.4.2).

The main contributions we make are as follows:

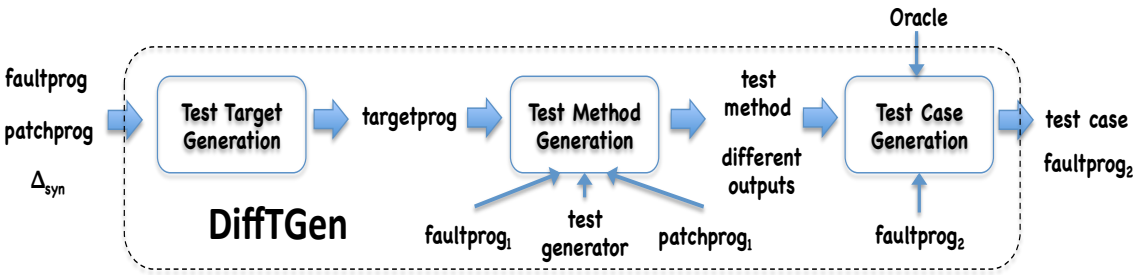

Figure 4.1: The Overview of DiffTGen. $faultprog$: the faulty program; *patchprog*: the patched program; Δ_{syn}: the syntactic differences between $faultprog$ and *patchprog*; *targetprog*: the test target program; $faultprog_1$, $patchprog_1$: the output-instrumented versions of $faultprog$ and *patchprog*; $faultprog_2$: the test-case-instrumented version of $faultprog$.

- We built a patch testing tool DiffTGen which could identify an overfitting patch generated by an APR technique through test case generation. The tool is currently available at *https://github.com/qixin5/DiffTGen*.
- We empirically evaluated DiffTGen on a set of 89 patches generated by four APR techniques for Java: jGenProg [145] (available at [16]) which is a Java version of GenProg [116, 74], jKali [145] (available at [16]) which is a Java version of Kali [183], Nopol (version 2015) [234] (available at [17]), and HDRepair [113] (available at [9]). Among the 89 patches, we identified 10 patches to be non-overfitting and 79 patches to be possibly overfitting (and thus incorrect). DiffTGen identified 39 (49.4%) patches to be overfitting with the corresponding test cases generated. With a bug-fixed program as the oracle, the average running time is only about 7 minutes.
- We empirically evaluated the effectiveness of DiffTGen in the context of automated program repair. Our results show that an APR technique, if configured with DiffTGen, could avoid generating overfitting patches and generate correct patches eventually.

4.2 Overview

In this section, we go over how DiffTGen works with an example. DiffTGen accepts as input a faulty program $faultprog$, a patched program *patchprog*, a set of syntactic differences Δ_{syn} between the two programs, and an oracle. A syntactic difference $\delta_{syn} \in \Delta_{syn}$ is a tuple $< faultstmt, patchstmt >$ where $faultstmt$ and *patchstmt* are the respective statements in $faultprog$ and *patchprog* that are related to the change. Note that a δ_{syn} could have a *null* value for either $faultstmt$ or *patchstmt* (but not both) to represent an insertion or a deletion. If neither $faultstmt$ nor *patchstmt* is *null*, δ_{syn} is a replacement. In the context of automated program repair, a repair technique often produces a patch report containing what changes it has made, and Δ_{syn} could be obtained by a simple report analysis. As output, DiffTGen either produces a test case showing *patchprog* is overfitting or produces nothing

```
1 public static boolean toBoolean(String str) {
2  if (str=="true") return true;
3  if (str==null) return false;
4  switch (str.length()) {
5    case 2: { ... }
6    case 3: {
7     char ch = str.charAt(0);
8     if (ch=='y')
9       return
10      (str.charAt(1)=='e'||str.charAt(1)=='E')
11      && (str.charAt(2)=='s'||str.charAt(2)=='S');
12
13    if (ch=='Y')//Changed to "if (str!=null)" (Overfitting Patch)
14      return
15      (str.charAt(1)=='E'||str.charAt(1)=='e')
16      && (str.charAt(2)=='S'||str.charAt(2)=='s');
17
18    //Inserted "return false;" (Correct Patch)
19   }
20   case 4: {
21     char ch = str.charAt(0);
22     if (ch=='t') {
23       return
24       (str.charAt(1)=='r'||str.charAt(1)=='R')
25       && (str.charAt(2)=='u'||str.charAt(2)=='U')
26       && (str.charAt(3)=='e'||str.charAt(3)=='E');
27     }
28     if (ch=='T') { ... }
29   }
30 }
31 return false;
32 }
```

Figure 4.2: The Lang_51 Bug & an Overfitting Patch

if no such test cases can be found. (For a generated test case, DiffTGen also produces a test-case-instrumented version of *faultprog*. For testing, one needs to run this version against the test case. In the instrumented version, the original semantics of *faultprog* is preserved, see Section 4.3.4.) Intuitively, an overfitting, patched program passes the original test suite but either does not fully repair the bug or repairs the bug but introduces new bugs which the test suite does not expose (see Section 4.3.1 for a formal definition of an overfitting patch). DiffTGen goes through three stages to produce a test case: *Test Target Generation*, *Test Method Generation*, and *Test Case Generation*. In the first stage, DiffTGen produces a target program *targetprog* on which a test generator works to generate test inputs. In the second stage, DiffTGen employs a test generator to actually generate test methods (as test inputs) that uncover semantic differences between *faultprog* and *patchprog*. In the third stage, DiffTGen produces test cases, if any, showing *patchprog* is overfitting based on the semantic differences. Figure 4.1 shows an overview of DiffTGen.

We use the example shown in Figure 4.2 to explain the three stages. The faulty program (in

Java) is a real bug (*Lang_51*) in the Defects4J bug dataset [94]. The functionality of the program is to convert a string into a boolean value. The fault-exposing test case from the test suite associated with the bug invokes the method `toBoolean` with the string "tru" as the value for `str`. Upon execution with `str="tru"`, the method `toBoolean` is expected to return *false* as the output. However, without the correct return statement inserted at Line 17, the branch of `case 4` is executed where an *IndexOutOfBoundsException* is thrown at Line 26. A patched program that modifies the if-condition at Line 13 (from `ch=='Y'` to `str!=null`) is generated by an APR technique Nopol [234] in the repair experiments conducted by Martinez et al. [143]. The patched program now works fine for the input "tru" (it returns *false* after executing the return statement starting at Line 14) and passes the original test suite, but it does not actually repair the bug. For this example, DiffTGen generates a new test case with the input string "@es" which exposes a new failure: the expected output is *false* but the patched program returns *true*. With a fixed version of the program being the oracle, it only took DiffTGen about 3.8 minutes to generate the test case.

4.2.1 Test Target Generation

In the first stage, DiffTGen generates a program which we call the test target program, or *targetprog*, based on $faultprog$, *patchprog*, and the syntactic differences Δ_{syn} between them. *targetprog* is the actual program on which a test generator later works to generate test inputs. It is an extended version of *patchprog* with dummy statements inserted as the coverage goals. A test input that is generated by a test generator with at least one of the coverage goals satisfied can lead to a differential execution between $faultprog$ and *patchprog*. Such an input is likely to uncover a semantic difference δ_{sem} between the two programs and further expose an overfitting behavior of *patchprog*.

To obtain *patchprog*, for each $\delta_{syn} \in \Delta_{syn}$, DiffTGen inserts a dummy statement in *patchprog*. For simple cases, where a patching modification does not involve changing an if-condition, DiffTGen simply inserts a dummy statement before the modified statement (for insertion or replacement) or in place of the removed statement (for deletion). For more complicated cases, where a patching modification is related to an if-statement and effectively modifies an if-condition (which is a common situation [172, 144]), DiffTGen produces a synthesized if-statement containing a dummy statement and inserts it in *patchprog*. The advantage of such a synthesized if-statement is as follows: a test input that covers the dummy statement (the coverage goal) would expose different branch-taking behaviors related to a modified if-statement between $faultprog$ and *patchprog*. Such a test input is thus likely to uncover a δ_{sem} between the two programs.

For the example in Figure 4.2, DiffTGen creates a *targetprog* by inserting a newly synthesized if-statement (which is a dummy statement) before Line 13. The synthesized if-statement is as shown below.

```
if (((ch=='Y')&&!(str!=null))||(!(ch=='Y')&&(str!=null))) {
    int delta_syn_3nz5e_0 = -1;
}
```

```
1 @Test public void test078() throws Throwable  {
2   boolean boolean0 = BooleanUtils.toBoolean("@es");
3 }
```

Figure 4.3: A Test Method Generated by EvoSuite

```
1 public static boolean toBoolean(String str) {
2   Object o_7au3e = null;
3   String c_7au3e =
4   "org.apache.commons.lang.BooleanUtils";
5   String msig_7au3e =
6   "toBoolean(String)" + eid_toBoolean_String_7au3e;
7   try {
8     o_7au3e = toBoolean_7au3e(str);
9     FieldPrinter.print(o_7au3e, eid_toBoolean_String_7au3e,
10                       c_7au3e, msig_7au3e, 0, 5);
11  } catch (Throwable t7au3e) {
12    FieldPrinter.print(t7au3e, eid_toBoolean_String_7au3e,
13                       c_7au3e, msig_7au3e, 0, 5);
14    throw t7au3e;
15  } finally {
16    eid_toBoolean_String_7au3e++;
17  }
18  return (boolean) o_7au3e;
19 }
```

Figure 4.4: The Output-Instrumented Version of *faultprog*.

In the context of the program, the if-condition is equivalent to `if (ch!='Y')`. In this example, for any input *str* that covers the dummy statement in *targetprog*, it would lead to a differential execution between *faultprog* and *patchprog*: the input would not exercise the return statement in *faultprog* (starting at Line 14) but would exercise the one in *patchprog*.

4.2.2 Test Method Generation

In the second stage, DiffTGen employs an external test generator (we use EvoSuite [65]) to generate test methods (test inputs) for *targetprog* that can cover at least one of the dummy statements upon execution. (Note that a test method contains no assertion statements, but there is at least one assertion statement in a test case.) For our example, one of the generated test methods is shown in Figure 4.3.

A generated test method can exercise a δ_{syn} between *faultprog* and *patchprog* upon execution, but may or may not be able to uncover a δ_{sem}. To tell whether a test method can uncover a δ_{sem}, DiffTGen creates instrumented versions of *faultprog* and *patchprog* (called the *output-instrumented* versions), runs them against the test method to obtain running outputs, and compares the outputs. In an output-instrumented version of a program (either *faultprog* or *patchprog*), DiffTGen creates statements that print as outputs values that can be affected by δ_{syn}.

```
1 //The output of the faulty program (instrumented)
2 Test Method: test078
3 PRIM_LOC:(E)0,(C)org.apache.commons.lang.BooleanUtils,(MSIG)toBoolean(
    ↪ String)0,(I)0
4 TYPE:Boolean
5 VALUE:false
6
7 //The output of the patched program (instrumented)
8 Test Method: test078
9 PRIM_LOC:(E)0,(C)org.apache.commons.lang.BooleanUtils,(MSIG)toBoolean(
    ↪ String)0,(I)0
10 TYPE:Boolean
11 VALUE:true
```

Figure 4.5: The outputs of running the faulty program and the patched program (both instrumented) against the test method in Figure 4.3 (the input).

For our example, DiffTGen creates an output-instrumented version of `toBoolean` shown in Figure 4.4[1] (which can be used for either *faultprog* or *patchprog*). Essentially, the code calls the original version of the method at Line 8 (the one shown in Figure 4.2, now renamed as `toBoolean_7au3e`) and prints the returned value `o_7au3e` at Lines 9-10. Along with the return value, the code also prints other values (e.g., one of them is the full class name of the method `c_7au3e`) which DiffTGen later uses to retrieve the output value for producing an assertion statement for a test case. If any exceptions are thrown, DiffTGen would also print the exceptional information (Lines 12-13). More details can be found in Section 4.3.3.

DiffTGen runs the output-instrumented versions of *faultprog* and *patchprog* against the test method shown in Figure 4.3 to obtain two outputs in Figure 4.5. (To do so, DiffTGen first removes the test method's annotation *@Test* and runs a class containing a main method where the test method is called.) The outputs basically show that for the first execution (indicated by *(E)0* at Lines 3 & 9) of the `toBoolean` method in the `BooleanUtils` class, the return values (indicated by *(I)0* at Lines 3 & 9) are different: one being `false` and the other being `true`. In the next stage, DiffTGen produces a test case based on the two different outputs.

4.2.3 Test Case Generation

In the third stage, DiffTGen compares the two outputs generated in the previous stage to identify specific values that are different, and then asks the oracle to tell which is correct. If the value generated by *patchprog* is incorrect, DiffTGen determines *patchprog* to be overfitting with a test case generated.

Given the generated output strings shown in Figure 4.5, DiffTGen found that output values (at Lines 5 & 11) are different and are comparable since their location properties (the *PRIM_LOC* values at Lines 3 & 9) are the same. DiffTGen then asks an oracle to determine which output value

[1]The code needs a JDK version higher than 1.5 to compile.

```
1 public static boolean toBoolean(String str) {
2   Object o_7au3e = null;
3   String c_7au3e =
4   "org.apache.commons.lang.BooleanUtils";
5   String msig_7au3e =
6   "toBoolean(String)" + eid_toBoolean_String_7au3e;
7   try {
8     o_7au3e = toBoolean_7au3e(str);
9     addToORefMap(msig_7au3e, o_7au3e);
10    addToORefMap(msig_7au3e, null);
11    addToORefMap(msig_7au3e, null);
12  } catch (Throwable t7au3e) {
13    addToORefMap(msig_7au3e, t7au3e);
14    throw t7au3e;
15  } finally {
16    eid_toBoolean_String_7au3e++;
17  }
18  return (boolean) o_7au3e;
19 }
```

Figure 4.6: The Test-Case-Instrumented Version of *faultprog*.

```
1 @Test public void test078() throws Throwable  {
2   BooleanUtils.clearORefMap();
3   boolean boolean0 = BooleanUtils.toBoolean("@es");
4   List obj_list_7au3e = (List) BooleanUtils.oref_map
5   .get("toBoolean(String)0");
6   Object target_obj_7au3e = obj_list_7au3e.get(0);
7   assertFalse(
8   "(E)0,(C)org.apache.commons.lang.BooleanUtils," +
9   "(MSIG)toBoolean(String)0,(I)0",
10  ((Boolean) target_obj_7au3e).booleanValue());
11 }
```

Figure 4.7: Test Case Generated by DiffTGen

is correct. For this example, we used the fixed version of the faulty program (the manually fixed version available in the Defects4J dataset) and found that the output value of *faultprog* (which is *false*) is correct but the output value of *patchprog* (which is *true*) is incorrect. (To do so, we created an output-instrumented version for the fixed version and ran it against the test method to obtain the expected output. In general, a human oracle would be needed and DiffTGen needs to be amenable to a human. The research of involving a human oracle for test case generation is left as future work.)

With the expected output provided by an oracle, DiffTGen creates a test-case-instrumented version for *faultprog* (Figure 4.6) and produces a test case (Figure 4.7) by augmenting the test method with an assertion statement and other statements for creating the assertion. In the test-case-intrumented version of *faultprog*, DiffTGen saves the reference to the object `o_7au3e`, the target object whose value to be asserted, in a static map field *oref_map* in the class of `toBoolean`.

In the test case (Figure 4.7), DiffTGen creates two statements (Lines 4-6) obtaining the target object and one assertion statement (Lines 7-10) asserting the value to be *false* as expected. More details can be found in Section 4.3.4. DiffTGen finally reports the patch to be overfitting with the generated test case as an evidence.

4.3 Methodology

In this section, we first give the definition of an overfitting patch, and then elaborate on the three stages that DiffTGen takes to identify an overfitting patch with a test case generated.

4.3.1 The Definition of an Overfitting Patch

Let $faultprog$ be the faulty program which we assume contains only one bug and I be the input domain of $faultprog$. I can be divided into two sub-domains I_0 and I_1 on which $faultprog$ has the *correct* and *incorrect* behaviors respectively. Let $fixprog$ be a correct version of $faultprog$ that only repairs the bug and does not contain any new features. Assuming both programs are deterministic, then we have $\forall i_0 \in I_0.faultprog(i_0) = fixprog(i_0) \wedge \forall i_1 \in I_1.faultprog(i_1) \neq fixprog(i_1)$ where we use $p(i)$ to denote the program behavior of p on a specific input i. Let *patchprog* be a patched program that was generated by a repair technique and can pass a test suite that $faultprog$ failed. Assuming *patchprog* is also deterministic, then we have $\exists i_1 \in I_1.patchprog(i_1) = fixprog(i_1)$. A repair technique can produce an overfitting patch which does not actually repair the bug. An overfitting patch (or a patched program) can be categorized into two types:

- **Overfitting-1**: The patch repairs *some (or even all)* of the incorrect behaviors of the original program but *breaks* some of its correct behaviors.
- **Overfitting-2**: The patch repairs *some (but not all)* of the incorrect behaviors of the original program and *does not break* any of its correct behaviors.

For a *patchprog* that is overfitting-1, we have

$$\exists i_0 \in I_0.\exists i_1 \in I_1.patchprog(i_0) \neq fixprog(i_0) \wedge patchprog(i_1) = fixprog(i_1)$$

For a *patchprog* that is overfitting-2, we have

$$\begin{aligned}
&\forall i_0 \in I_0.patchprog(i_0) = fixprog(i_0) \wedge \\
&\exists i_{10} \in I_1.\exists i_{11} \in I_1.i_{10} \neq i_{11} \wedge \\
&patchprog(i_{10}) \neq fixprog(i_{10}) \wedge \\
&patchprog(i_{11}) = fixprog(i_{11})
\end{aligned}$$

Our definition[2] is consistent with the definition of a bad fix given by Gu et al. [75]: a bad fix either introduces disruptions (regressions) or does not cover all the bug-triggering inputs or both. A patched

[2]We note a very similar definition given by Yu et al. in their work [244].

program that is overfitting-1 introduces regressions and is not acceptable, but a patched program that is overfitting-2 does not introduce regressions (though it only makes a partial repair) and may thus be considered as still valid. DiffTGen can identify a patched program[3] to be overfitting-1 by finding an input that exposes a semantic difference between $faultprog$ and *patchprog* and further showing the semantics of *patchprog* is incorrect while the semantics of $faultprog$ is correct with the assistance of an oracle. However, it cannot directly identify a patched program to be overfitting-2. Identifying such a patched program involves two steps: (1) showing $\exists i_1 \in I_1.patchprog(i_1) \neq fixprog(i_1)$ (**Overfitting-2a**) and (2) showing $\forall i_0 \in I_0.patchprog(i_0) = fixprog(i_0)$ (**Overfitting-2b**). DiffTGen can achieve (1) by finding a test input exposing a semantic difference between $faultprog$ and *patchprog* and further showing the semantics are both incorrect[4]. However, it cannot achieve (2) by proving the patched program contains no regressions.

4.3.2 Test Target Generation

In the first stage, DiffTGen creates a test target program, or *targetprog*, based on the syntactic differences Δ_{syn} between $faultprog$ and *patchprog*. *targetprog* is the program on which a test generator later works to generate test inputs that uncover semantic differences between $faultprog$ and *patchprog*.

DiffTGen creates *targetprog* by extending *patchprog* with dummy statements inserted (one for each δ_{syn}). The inserted dummy statements do nothing but can be detected by a test generator as the coverage goals. DiffTGen inserts dummy statements into *targetprog* in such a way that at least a dummy statement would be executed if and only if the execution of $faultprog$ and *patchprog* would differ.

For simple cases, where a patching modification δ_{syn} does not involve modifying an if-condition (e.g., it modifies an assignment), DiffTGen simply creates a dummy statement and inserts it in front of the modified statement (for insertion or replacement), or in place of the deleted statement (for deletion) in *patchprog*. If a generated test input can cover the dummy statement upon execution, the input would cover the modified statement in *patchprog* but the unmodified statement in $faultprog$, and would thus lead to a differential execution between $faultprog$ and *patchprog*.

The more complicated cases arise when δ_{syn} is related to an if-statement and effectively modifies an if-condition. (This is a common situation [172, 144]. In fact, there exist repair techniques that only look at condition-related bugs [234, 232].) In such cases, it might be ineffective just to insert a dummy statement in front of an if-statement whose condition is modified. Figure 4.8 is an example where the faulty program `faultprog` and the patched program `patchprog` are shown at the top and in the middle. The faulty if-condition `x<999` at Line 2 was changed to `x<1000` at Line 5. If DiffTGen simply creates a dummy statement and inserts it before the if-statement at Line 5 as the

[3] A patched program is known to have something repaired, since it passed the test suite that the original program failed.

[4] Note that DiffTGen does not find an input showing the semantics of the two programs are identical but incorrect. Such an input is not directly related to what changes a patch makes.

```
1 int faultprog(int x) {
2   if (x < 999) { x++; } //the faulty statement
3   return x; }
4 int patchprog(int x) {
5   if (x < 1000) { x++; } //the patched statement
6   return x; }
7 int targetprog(int x) {
8   if ((!(x<999)&&(x<1000))||((x<999)&&!(x<1000))){
9     int delta_syn_3nz5e_0 = -1; } //a dummy statement
10  if (x < 1000) { x++; }
11  return x; }
```

Figure 4.8: A Test Target Example

coverage goal, then a test generator could quite possibly end up with finding an input `x` taking a random value, say 33, to make the dummy statement covered. However, such an input can expose no semantic difference between the two programs.

To address the problem, DiffTGen creates a synthesized if-statement and inserts it before the modified statement or at the modification place in *targetprog*. The new if-statement contains a new if-condition. It also contains a dummy statement as its then-branch. The advantage of such a synthesized if-statement is as follows: a generated test input that can cover the dummy statement would expose different branch-taking behaviors between the unmodified statement in $faultprog$ and the modified statement in *patchprog*. For example, in Figure 4.8, DiffTGen creates a synthesized if-statement starting at Line 8. For the dummy statement at Line 9 to be covered, a test input x has to satisfy the condition at Line 8 which is essentially `x==999`. Such an input can expose different branch-taking behaviors between $faultprog$ and *patchprog*: Given `x==999`, $faultprog$ does not execute its then-branch `x++`, but *patchprog* does. This input further exposes a semantic difference between the two programs, the return value of $faultprog$ is 999, but the return value of *patchprog* is 1000.

DiffTGen considers in total 10 different types of modifications to produce dummy statements to be inserted in *targetprog*. Table 4.1 shows the 10 cases with code examples. The three cases *Non-partial-if Insertion*, *Non-partial-if Deletion*, and *Other Change* cover the simple cases where DiffTGen simply inserts dummy statements into *patchprog* to produce *targetprog*. For each of the other cases where the modification effectively changes an if-condition, DiffTGen creates a synthesized if-statement to be inserted in *patchprog*. (Note that some of the cases can be considered as changing if-conditions. For example, inserting a partial if-statement `if(c){s}` can be considered as changing the condition of an if-statement `if(false){s}` from `false` to `c`.) To create a target program, for each $\delta_{syn} \in \Delta_{syn}$, DiffTGen looks at the 10 change cases in the same ordering as listed in Table 4.1 (from top to bottom), finds the first change case that is matched, produces the new statement, and inserts it in *targetprog*.

Table 4.1: Test Target Generation of 10 Change Cases

change case	faultstmt	patchstmt	targetprog
Partial-if Insertion	`null`	`if(c){s}`	patchprog with `if(c){dummystmt}` inserted before patchstmt
Non-parial-if Insertion	`null`	`s`	patchprog with dummystmt inserted before patchstmt
Partial-if Deletion	`if(c){s}`	`null`	patchprog with `if(c){dummystmt}` inserted where faultstmt is deleted
Non-partial-if Deletion	`s`	`null`	patchprog with dummystmt inserted where faultstmt is deleted
If-Guard Insertion	`s`	`if(c){s}`	patchprog with `if(!c){dummystmt}` inserted before patchstmt
If-Guard Deletion	`if(c){s}`	`s`	patchprog with `if(!c){dummystmt}` inserted before patchstmt
If-Cond Change	`if(c1){s}`	`if(c2){s}`	patchprog with `if(!c1&&c2||c1&&!c2){dummystmt}` inserted before patchstmt
If-Cond-Else Change	`if(c1){s}`	`if(c2){s}else{e2}`	patchprog with `if(!(c1&&c2)){dummystmt}` inserted before patchstmt
If-Cond-Then Change	`if(c1){s1}else{e}`	`if(c2){s2}else{e}`	patchprog with `if(c1||c2){dummystmt}` inserted before patchstmt
Other Change	`s1`	`s2`	patchprog with dummystmt inserted before patchstmt

A partial-if-statement does not have an else branch.

4.3.3 Test Method Generation

In this stage, DiffTGen employs a test generator EvoSuite to generate test methods (test inputs) for *targetprog* with at least one of the coverage goals satisfied (i.e., with at least one of the dummy statements covered). Such a test method can exercise at least a δ_{syn} and can cause the executions of $faultprog$ and *pathprog* to differ. However, the test method may not be able to expose any semantic difference δ_{sem} between the two programs. To determine whether a test method can expose a δ_{sem}, DiffTGen creates instrumented versions of $faultprog$ and *patchprog*, runs the two instrumeted versions against the test method to obtain running outputs, and compares the outputs. We call such an instrumented program on which DiffTGen executes to obtain outputs an *output-instrumented* program. For the rest of the section, we focus on explaining how to create an output-instrumented version of a program.

Creating an Output-Instrumented Version

DiffTGen needs to be able to detect whether a given test run exposes a semantic change between $faultprog$ and *patchprog*. In the simplest case, a test method (as a test driver) runs a patched method directly and any difference is seen in the return value of the method. However, real-world patches are seldom that simple: a test method might call other methods which in turn call the patched method; the difference between two executions might not be reflected in the return value, but might be reflected in a changed field accessible from an argument passed to the method.

To accommodate these various possibilities, DiffTGen creates an output-instrumented version of a program by augmenting the program with printing statements. We assume a patching modification is made within a method and a semantic change can propagate to the "input" and "output" elements of the method. We define the input elements of a method to be the arguments (including the *this* argument) that are passed to the method on entry, and we define the output elements to be the return value and any exceptions thrown on exit[5]. For each $\delta_{syn} \in \Delta_{syn}$, DiffTGen looks at the input and output elements of the method where the change is made (also called the *delta-related* method), and prints the values of the elements and the types. (Note that DiffTGen does not print any input argument that is of a primitive type, a *String* type, or is passed as a *final* type, since a change cannot propagate to such an argument after the method execution.)

[5]Note that a change can also propagate to a static class field which currently we not handle. Handling this type of changes is left as future work.

DiffTGen actually calls a printer (*FieldPrinter*) that we created to print values and types. For an element that is of a primitive type or a *String* type, the printer simply prints its value and type; For an element that is an array, a list, a set, or a map, the printer creates a list for the element, and prints the list elements in turn; For an element that is of a Java *Throwable* type, the printer calls the *toString* method and prints the returned string as the value, and it prints the keyword "Throwable" as the type; For an element that is of other types, the printer uses Java reflection[6] to explore the structure of the element[7] (as an object) and prints the fields in a depth-first approach for which we use 5 as the maximum depth for exploration.

At the implementation level, for each delta-related method m_δ, DiffTGen creates a stub method m'_δ whose method signature, method name, and parameter names are equal to those of m_δ. DiffTGen then renames m_δ. In m'_δ, it creates a statement calling the renamed m_δ in a try statement. After calling m_δ, DiffTGen creates statements calling *FieldPrinter.print* to print the input and output elements of m_δ. In the catch clause, it creates a statement printing the thrown exception. The printer accepts six arguments. The first argument is the element to be printed. The printer either simply prints the value of the element and its type or explores the element's internal structure to print a sequence of values and the corresponding types. For each value, the printer also prints the retrieval information showing how the value can be retrieved from an execution (e.g., indicating the printed value is the return value of the method in its first execution). For printing the retrieval information, the printer also accepts as arguments the call count (which is associated with m'_δ), the class name, the extended method signature (which is a string consisting of the method signature of m_δ and the call count), and the property of the element to be printed (indicating, e.g., it is a return value). The final argument the printer accepts is the maximum printing depth (we use 5). In the finally clause, DiffTGen creates a statement increasing the call count. In m'_δ, DiffTGen also creates other statements that define variables and return the final result (if needed).

To obtain outputs, DiffTGen creates a test class, copies each test method (with the annotation *@Test* removed) to the class, creates a main method in the class, and calls the main method to run each test method over the output-instrumented versions of $faultprog$ and *patchprog*. An output is printed in a stylized form so that the corresponding lines can be easily compared.

4.3.4 Test Case Generation

In the previous stage, DiffTGen runs the output-instrumented versions of $faultprog$ and *patchprog* against a test method to obtain running outputs. In this stage, DiffTGen compares the outputs to identify specific values that are different, and then asks the oracle to tell which is correct. When the value generated by *patchprog* is incorrect, DiffTGen determines *patchprog* to be overfitting. If a correct value could be provided by the oracle, DiffTGen performs two steps to produce a test case: (1) creating a test-case-instrumented version (for the original $faultprog$ for which a test case

[6] We use FieldUtils from the apache package commons-lang3-3.5.

[7] DiffTGen ignores an element that is declared to be a *final* or a *static* type which usually does not contain a semantic change.

is created) and (2) augmenting the test method. DiffTGen uses the two steps mainly to create an assertion in the test case that asserts the value (that was checked and compared) to be equal to the expected one provided by the oracle.

Comparing the Running Outputs

DiffTGen compares the running outputs of $faultprog$ and *patchprog* to identify *comparable* values that are different[8]. Two values are comparable if the two pieces of retrieval information associated with the values (indicating how the values can be generated) are identical. More specifically, DiffTGen goes through the two outputs (as two strings) line by line in parallel. When the two lines examined both start with $VALUE$ (e.g., Lines 5 & 11 in Figure 4.5), DiffTGen obtains the corresponding value items which we call the *check value*s. DiffTGen also obtains the retrieval information by looking at two lines before the current lines that start with $PRIM\_LOC$. We call the corresponding value items the *loc value*s. When the two loc values are identical but the two check values are different, DiffTGen successfully identifies comparable values that are different, and it provides to the oracle (1) the test method, (2) the loc value, (3) the two check values, and (4) the types of the check values (obtained from one line before the check value lines that start with $TYPE$). DiffTGen asks the oracle to determine which value is correct (and if the value types are different, what is the correct type). If neither is correct, DiffTGen further asks the oracle to provide a correct value (possibly with a value type). An oracle may not provide a correct value or a type (correctness judging between two values might not be easy for a human oracle). In that case, DiffTGen discards the current check values and keeps looking for other check values in the outputs. (For our experiments in Section 4.4, DiffTGen uses a fixed version of $faultprog$ as the oracle.)

Generating a Test Case

Given an expected value (possibly with an expected type) and a loc value used to generate the value to be asserted, DiffTGen produces a test case mainly by augmenting the test method with an assertion statement. To create the assertion statement, DiffTGen needs to do three things: (1) obtain the input/output element to be asserted; (2) obtain the value to be asserted from the input/output element; and (3) produce an assertion statement asserting the value to be equal to the expected value.

(2) and (3) are easy to do. Once an input/output element is available, DiffTGen parses the loc value to obtain the access path which it needs to follow to obtain the value to be asserted (or the *target* value). With the access path being ready, DiffTGen uses Java reflection to explore field structure of the element, creates statements that follow the path to obtain the target value syntactically, and inserts the statements in the test method. Then DiffTGen simply creates an assertion statement asserting the target value to equal to the expected value.

The difficulty lies in (1): how to obtain the input/output element to be asserted (or the *target*

[8]DiffTGen currently does not produce any test case based on output values that are not comparable.

element). For the simple test method as shown in Figure 4.3, the target element (i.e., the return value `boolean0`) is syntactically available. In general, however, the target element might not be syntactically available in the test method: consider the case where the delta-related method (where a patching modification is made) is a private method called by a public method called in the test method. To still be able to syntactically obtain the target element in the test method, DiffTGen creates an instrumented version of $faultprog$, which we call the test-case-instrumented version, that keeps track of the input/output elements of a delta-related method by storing the elements in a map (as a static field of the method's located class). Later, to syntactically obtain an input/output element in the test method, DiffTGen simply creates a statement that refers to the field map to get the element.

For the rest of the section, we first explain how to create a test-case-instrumented version of a program, then explain how to augment a test method to produce a test case.

Creating a Test-Case-Instrumented Version In a test-case-instrumented version, the parent class of each delta-related method contains a static field map named *oref_map* that stores the input and output elements of the method. The key of the map is a string consisting of the signature of the delta-related method and a call count associated with the method (i.e., the *extended method signature*). The value of the map is a list of the input/output elements.

Creating a test-case-instrumented version is similar to creating an output-instrumented version: DiffTGen looks at each delta-related method m_δ (where a patching modification is made), creates a stub method m'_δ, renames m_δ, and creates a try-statement within m'_δ where m_δ is called. Here, after this method call, instead of creating statements printing the input/output elements, DiffTGen creates statements calling a static method *addToORefMap* it creates to store the elements in the map *oref_map*. *addToORefMap* accepts two arguments: (1) the extended method signature of m_δ (before it is renamed, as a key stored in *oref_map*) and (2) the input/output element (stored in a list as the value of the key). DiffTGen calls *addToORefMap* to store in a list the return value, the *this* argument, and the method arguments in turn. (If an element is not available, it stores *null*.) Similarly, in the catch clause, DiffTGen calls *addToORefMap* to store the thrown exception.

Augmenting the Test Method Given the expected value provided by the oracle, the loc value obtained from the running output, and a test-case-instrumented version created, DiffTGen finally produces a test case by augmenting the test method. In the test case, DiffTGen mainly creates statements that (1) syntactically obtain the target element (i.e., the input/output element to be asserted) by referring to the static field map (*oref_map*) created in the test-case-instrumented version, (2) syntactically obtain the target value (i.e., the value to be asserted) by following the access path contained in the loc value to explore the target element, and (3) assert the target value to be equal to the expected value.

More specifically, DiffTGen creates a test case whose method signature is identical to that of the test method and contains an extra *@Test*. In the test case, DiffTGen first creates a statement clearing the map *oref_map* contained in the test-case-instrumented version. Next it copies all the

```
1 Object target_obj_7au3e = null;
2 boolean not_thrown = false;
3 try {
4   <CLASS NAME>.clearORefMap();
5   <TESTING CODE GENERATED BY EVOSUITE>
6   not_thrown = true;
7   fail();
8 } catch (Throwable t) {
9   if (not_thrown) { fail("Throwable␣Expected!"); }
10  else {
11    target_obj_7au3e=...;//get the input/output element
12    assertEquals(<MESSAGE>, <EXPECTED VALUE>,
13      ((Throwable) target_obj_7au3e).toString());
14  }
15 }
```

Figure 4.9: Augmenting a Test Method with an Expected Throwable

statements from the test method. Next it creates statements to syntactically obtain the target element by referring to *oref_map* using the extended method signature and the property value it obtained from the loc value (the property value is actually the index of the target element in the element list stored in *oref_map*). Next it creates statements to syntactically obtain the target value from the target element. Again, when the target element is not of a primitive, a *String*, or a *Throwable* type, DiffTGen uses Java reflection to explore the structure of the target element and follows the access path (contained in the loc value) to get the target value. Finally, DiffTGen creates a *JUnit* statement asserting the target value to be equal to the expected value.

Note that when the element to be asserted is an exception, DiffTGen uses the template shown in Figure 4.9 to produce a test case. Essentially, DiffTGen creates a try statement and copies the testing statements from the test method to the try-body. DiffTGen creates the augmented statements in the catch clause.

4.4 Empirical Evaluation

To empirically evaluate the effectiveness of DiffTGen, we ask two questions:

- **RQ1**: Could DiffTGen identify overfitting patches generated by APR tools? What is its performance?
- **RQ2**: Could DiffTGen enhance the reliability of an APR technique and guide the technique to produce correct patches?

We conducted two experiments to answer the two questions. We next show each experiment in turn.

4.4.1 RQ1

To evaluate the performance of DiffTGen in identifying overfitting patches, we created a patch dataset containing 89 patches (the patched programs) generated by four APR techniques for Java: jGenProg [145] (available at [16]) which is a Java version of GenProg [116, 74], jKali [145] (available at [16]) which is a Java version of Kali [183], Nopol (version 2015) [234] (available at [17]), and HDRepair [113] (available at [9]). Among the 89 patches, we identified 10 patches to be correct and non-overfitting and the other 79 patches to be possibly overfitting. We ran DiffTGen on each patched program and its original faulty program. Our results show that DiffTGen found 39 out of the 79 patches (49.4%) to be overfitting with the corresponding test cases generated.

Experimental Setup

Patch Dataset. The current implementation of DiffTGen is in Java. To evaluate its performance, we collected 89 patches generated by four APR techniques: jGenProg, jKali, Nopol, and HDRepair for bugs in the Defects4J dataset [94]. Martinez et al. did an experiment [143] running three repair tools: jGenProg, jKali, and Nopol on the Defects4J bugs and generated in total 84 patches. We included all these patches in our dataset. For patches generated by HDRepair, we contacted Le et al. (the authors of [113]) and obtained a set of 14 patches (for each of the 14 repaired bugs, we used the first found patch reported by HDRepair). We also included these patches in our dataset. Among the 84+14=98 patches, we found 9 patches are syntactically repetitive. We removed them and obtained a final dataset containing 89 individual patches[9]. It turns out each patch makes only a small change on only one statement.

Among the 89 patches, we determined 10 patches to be correct by syntactically comparing each of the 89 patches against the correct human patch (the fixed version) associated with the bug in the dataset (the syntactic comparisons are easy and the correctness of these 10 patches are obvious, see our provided links below for what they are). For the remaining 79 patches, we consider them as possibly incorrect[10].

DiffTGen. To test if a patch is overfitting or not, we ran DiffTGen with the faulty program $faultprog$, the patched program *patchprog*, and the syntactic changes between the two as input. For a syntactic change, we manually identified the two change-related statements from $faultprog$ and *patchprog* respectively. As the oracle, we used the human-patched program (the fixed version) in the Defects4J bug dataset associated with the bug[11]. For correctness judging, DiffTGen created an output-instrumented version for a bug's fixed version and ran it against any test method generated by EvoSuite twice to mark any printed fields whose values are inconsistent during the two executions.

[9] See *github.com/qixin5/DiffTGen/tree/master/expt0/dataset* for all the 89 patches we used (including the ones we identified to be correct) and the patches we removed.

[10] It is not easy to determine the correctness of the 79 patches by hand since they are not syntactically identical to the corresponding human patches (this is a reason why a tool like DiffTGen is needed). The rate of overfitting patches identified by DiffTGen (49.4%) is actually a lower bound.

[11] Note that the human patches only make changes about bug repairs and do not add any new features for the original bugs. This makes sure a test case generated by DiffTGen specifies the correct behavior of a bug but not any new features expected.

Table 4.2: The Running Result of DiffTGen (#Bugs: 89, #Bugs that are likely to be incorrect: 79)

Running Setup	Time	#SynDiff	#SemDiff	#Overfitting	#Regression (Overfitting-1)	#Defective (Overfitting-2a)
trial30_time60	6.9m	72	61	39	34	18
trial10_time180	8.0m	73	59	36	32	13
trial3_time600	11.4m	73	56	32	28	12
trial1_time1800	30.6m	69	48	27	23	8

For a running setup, DiffTGen ran the trials in parallel.
Note that DiffTGen cannot identify an Overfitting-2 patch, but can identify a patch to have an Overfitting-2a behavior.

DiffTGen considers the marked fields as non-deterministic and does not use them for test case generation. By using a fixed version of the bug as the oracle, DiffTGen runs automatically to produce a test case.

DiffTGen employs *EvoSuite-1.0.2* to generate test methods. (We did not use EvoSuite's functionality to generate assertions in a test method because we found the generated assertions often do not expose any semantic differences between $faultprog$ and *patchprog*.) EvoSuite uses an evolutionary search algorithm and allows the user to specify a searching timeout. For our experiments, as the default setup, DiffTGen generates test methods by calling EvoSuite in 30 trials with the searching timeout being 60s for each trial (or the setup *trial30_time60*). We implemented DiffTGen to run the trials in parallel. In Section 4.4.1, we compared the performances of DiffTGen running in different setups. We ran all the experiments on a machine with 8 AMD Opteron 6282 SE processors and 8G memory.

Results

The Performance of DiffTGen. DiffTGen's running result can be found in Table 4.2[12] (the first row corresponds to the default running setup). From left to right, the table shows the running setup (*Running Setup*); the average running time in minutes (*Time*); the numbers of bugs for which the syntactic difference between the two programs (the patch and the bug) has been exercised (*#SynDiff*); a semantic difference between the two programs has been found (*#SemDiff*); overfitting-indicative test cases have been generated (*#Overfitting*); regression-indicative test cases have been generated (*#Regression*); and defective-indicative test cases (the semantics of the two programs are both incorrect) have been generated (*#Defective*). Note that we consider the time duration of a run to be from the start of the run to the time when an overfitting-indicative test case is generated or when DiffTGen terminates with no such test case is generated (but we did not actually stop running DiffTGen until it terminated).

As shown, DiffTGen identified 39 patches to be overfitting (see Table 4.3 for what they are) with the corresponding test cases generated. For 34 patches, DiffTGen generated test cases showing they contain regressions (i.e., showing the semantics of the patch is incorrect but that of the bug is correct). For 18 patches, DiffTGen generated test cases showing they are defective (i.e., the semantics of $faultprog$ and *patchprog* are both incorrect). Note that DiffTGen could generate two

[12]Table 4.2 only shows a summary of the results. The complete result tables can be found at *https://github.com/qixin5/DiffTGen/tree/master/expt0/result*.

different test cases for a patch showing it not only contains regressions but also is defective. This explains why the sum of the last two columns in Table 4.2 can be greater than the fifth column. Our results show that DiffTGen is efficient: it takes about 7 minutes on average to test a patch (with or without test cases generated).

For 72 (80.9%) patches, DiffTGen found at least a test method for each patch that exercises the syntactic change between the patched and the original programs. For the other 17 patches, we found EvoSuite generated no test method at all for 4 of the patches. This could happen either because EvoSuite failed to generate anything within the time limit or because an error occurred during its run. For the other 13 of the 17 patches, although EvoSuite generated test methods, they do not exercise the syntactic changes and would not be useful to reveal any semantic differences. We think the reason could be that the overall goal of EvoSuite is to generate test methods to achieve a high coverage of the class under test, and it is not designed to generate test methods to cover a certain statement in particular.

A test method that exercises the syntactic change may or may not reveal a semantic difference. Using the underlying search algorithm of EvoSuite plus the synthesized if-statements created in the test target, for 84.7% (61/72) of the patches, DiffTGen obtained test methods that uncover some semantic differences. For 11 of the 72 patches, however, the test methods do not reveal any semantic difference. In general, finding a test that uncovers a semantic difference between two programs is undecidable: there could be a large number of paths exercising a syntactic change but only a small fraction of them may reveal a real semantic difference. Below is an example. For the bug *Chart_1*, jGenProg creates a patch by deleting the first if-statement.

```
1  if (dataset != null) { return result; } //Patch by deletion.
2  ... //The code here may change the value of "result".
3  //But no change would be made if "dataset" is empty & non-null.
4  return result;
```

To test the patch, DiffTGen creates a test target program by inserting a newly synthesized if-statement `if (dataset!=null){dummystmt}` at Line 1 (i.e., at the place where the original if-statement is deleted). Using EvoSuite, DiffTGen found a test method which initializes `dataset` to be a new empty object (non-null), and the dummy statement is exercised. However, since the object is empty, no changes are made to `result`, and no semantic difference is actually made (the code for this is not shown). The problem here is that the search algorithm of EvoSuite tends to build "simple" objects to satisfy its coverage goals. A simple object here (the empty `dataset`) would not reveal any semantic difference although the syntactic difference is exercised.

When a semantic difference is found, DiffTGen asks the oracle for semantic checking. DiffTGen found 34 patches that contain regressions with the corresponding test cases generated. For 18 patches, DiffTGen generated test cases showing they are defective (the outputs of $faultprog$ and *patchprog* are both incorrect), though they may or may not contain regressions. We use a simple example shown below to explain this.

```
1 int foo(int x) {
```

```
2   x = x + 1; //Bug. Should be x = x * 2;
3   //x = x + 2; (Patch)
4   return x; }
```

A patch changes the buggy statement at Line 2 to a new statement at Line 3. The patched program works fine for an input $x = 2$. We know it contains regressions because the program fails for an input $x = 1$ for which the original program works fine. But we also know the patched program is generally defective, since for many other inputs (e.g., when $x = 3$), both the original and the patched programs fail.

There are 22 cases where the found semantic differences do not reveal any overfitting properties of a patch. For 5 cases, DiffTGen only produced repair-indicative test cases (we found they correspond to the correct patches, for the other 5 correct patches, DiffTGen does not produce any test cases). For 17 cases, the semantic differences are not interesting or cannot be leveraged by DiffTGen for semantic checking. For example, the semantic difference between the faulty program and a patched program generated by jGenProg (for *Chart_7*) is related to a class field named *time* whose type is *long*. Such a field is time-related, and is not reliable for semantic checking. DiffTGen runs the oracle program twice to identify such fields and refuses to use them for semantic checking. For this example, DiffTGen generated no test cases. There are also forms of semantic changes that DiffTGen currently does not support for correctness judging. For example, a list has one more element added in the patched program. Since the values of the new element added have no comparable values in the faulty program, DiffTGen would not produce any test case based on the new element which causes the semantics to be different.

Setup Comparison. DiffTGen employs EvoSuite to generate test methods. To do so, EvoSuite uses evolutionary algorithms. To investigate how EvoSuite affects DiffTGen's results, we compared the default setup of DiffTGen *trial30_time60* (i.e., running EvoSuite in 30 trials with the search time of each trial limited to 60s) to three other setups: *trial1_time1800*, *trial3_time600*, and *trial10_time180* (we limit EvoSuite's overall search time to be 30 minutes to have these setups created). As the results in Table 4.2 show, DiffTGen needs to run EvoSuite in more than one trial to obtain better results. Sacrificing the search time (e.g., from 600s to 60s) for more trials (e.g., from 3 to 30) would cause the number of change-exercised test methods to slightly decrease (from 73 to 72) but would enhance the overall testing performance: the running time reduces (by about 40%) and the number of generated overfitting-indicative test cases increases (from 32 to 39).

4.4.2 RQ2

DiffTGen identified 39 patches to be overfitting with test cases generated. In the context of automated program repair, we want to know whether DiffTGen could work together with an APR technique to make the repair technique avoid generating overfitting patches and produce correct patches eventually. So in this experiment, we ran the four repair tools (jGenProg, jKali, Nopol, and HDRepair) on the 39 bugs for which DiffTGen generated new test cases showing the original patches are overfitting (we augmented the corresponding test suites associated with the bugs with

Table 4.3: 39 Overfitting Patches Identified by DiffTGen

Repair Tool	Bug ID
jGenProg	M2, Ch3∗, M40, Ch5, M80∗, Ch15, M78, T4, M8, M95, M81
jKali	M32, M2, Ch13, Ch26, M40, Ch5, Ch15, T4, M95, M81, M80
Nopol	Ch21, L51, L53, M33†, Ch13, M40, M87, M97, M57, M104, Ch5, M80∗, M105, M81∗
HDRepair	C10†, L6†, M50∗†

Ch: Chart, C: Closure, L: Lang, M: Math, T: Time.
ID with †: The correct patch exists in the tool's search space.
ID with ∗: Only defective-indicative test cases were generated.

Table 4.4: Repair Experiment 0

ID	Time	#Patch	#SynDiff Patch	#Correct Patch
Math_95_jGenProg	1.8m	10	2∗	0
Chart_15_jKali	28.7m	5	1	0
Chart_26_jKali	81.2m	2	1	0
Chart_13_Nopol	3.3m	10	1	0
Math_50_HDRepair	88m	7	6	4

∗: The two generated patches are invalid since they do not pass the test cases generated by DiffTGen. We believe it is a failure of jGenProg.

the new test cases). If new patches were generated, we ran DiffTGen again, and if new test cases were generated, we augmented the test suites and ran the repair techniques again, so on and so forth.

Figure 4.10 is a summary of the results. It shows that the repair techniques with DiffTGen configured avoid yielding any incorrect patches for 36 bugs eventually. For 33 of the 36 bugs, we find that there do not exist correct patches in the repair tools' search spaces. So the best the tools can do is to yield no patches, and DiffTGen makes them achieve that. For 3 of the bugs (*Math_33_Nopol*, *Closure_10_HDRepair*, and *Lang_6_HDRepair*), the corresponding repair tools could potentially produce a correct patch, but they did not since their search spaces of patches are too large and the correct patches were not actually found. For *Math_50*, HDRepair eventually produced four correct patches with the assistance of DiffTGen. For 3 of the 39 bugs (*Math_95_jGenProg*, *Chart_13_Nopol*, and *Math_50_HDRepair*), there were incorrect patches generated eventually. jGenProg produced two invalid patches for *Math_95* which did not pass the test cases generated by DiffTGen. DiffTGen failed to generate overfitting-indicative test cases for three patches: *Chart_13_Nopol*, *Math_50_HDRepair_0*, and *Math_50_HDRepair_1* which are overfitting and incorrect.

Experimental Setup

For each patch in Table 4.3, DiffTGen generated an overfitting-indicative test case. We added the test case to the test suite associated with the bug and obtained an augmented test suite (if multiple test cases have been generated for a patch, we added the one showing the patch contains regressions). For each bug, we next ran the repair technique (the one produced its initial patch) with the augmented test suite to try to find a new patch. For each of the four repair techniques, we ran it in 10 trials with the time limit being two hours for each trial. The original repair experiments reported in [113] ran HDRepair to repair a bug with a buggy method provided manually. To be consistent, we provided HDRepair with same buggy methods provided in [9] for repairing three of

Table 4.5: DiffTGen Experiment 0

ID	Time	SynDiff	SemDiff	Overfitting	Regression (Overfitting-1)	Defective (Overfitting-2a)
Chart_26_jKali	23.4m	true	true	true*	true	false
Chart_15_jKali	22.8m	true	true	true*	true	false
Chart_13_Nopol	11.0m	true	true	false	false	false
Math_50_HDRepair_0	11.2m	true	false	false	false	false
Math_50_HDRepair_1	16.7m	true	true	false	false	false

*: A following repair experiment shows that jKali failed to produce any patches using the test suite augmented with the newly generated overfitting-indicative test case.

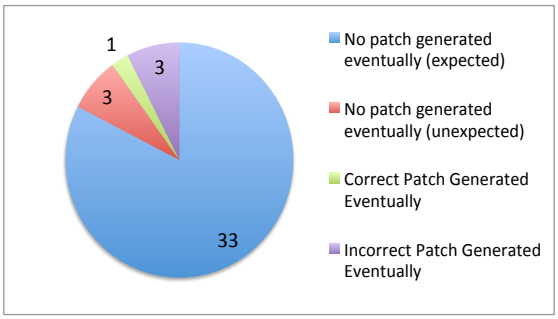

Figure 4.10: The numbers of bugs for which no patches (expected or unexpected), correct patches and incorrect patches were eventually generated. (For *Math_50_HDRepair*, both correct and incorrect patches were generated.)

the bugs *Closure_10*, *Lang_6*, and *Math_50*. For any new patches generated, we ran DiffTGen again to generate new test cases. In this experiment, we used the default setup of DiffTGen for test case generation. Currently, we do not have an integrated version of a repair technique and DiffTGen. So each time we ran a repair technique, we manually added the newly generated test case to the test suite, and each time we ran DiffTGen, we manually provided it with the syntactic changes that the patch makes.

The Potential Of Producing a Correct Patch

We analyzed the fixed version (the human patch) for each of the bugs listed in Table 4.3 and found that for only 4 bugs (marked with †), the corresponding repair techniques could potentially produce correct patches. For the other bugs, the correct patches do not exist in the tools' search spaces. We find that jGenProg, jKali, and Nopol have their own limitations. jGenProg fails to produce a correct patch if the fix statements do not exist in the original faulty program. jKali can only produce patches that remove statements. Nopol can only repair an if-condition-related bug whose fix needs a simple change (on only one condition). Compared to the other techniques, HDRepair could potentially generate correct patches for its three bugs. Its search space is much larger, but it leverages historical repair data to make the search guided.

Results

As Figure 4.10 shows, there are in total 36 bugs for which no patches were generated by the repair techniques. For 33 of the bugs, the corresponding repair techniques do not have the abilities in

producing correct patches, and the fact that no patches were eventually generated is expected. For three of the bugs (*Math_33_Nopol*, *Closure_10_HDRepair*, and *Lang_6_HDRepair*), although the corresponding repair techniques can potentially produce correct patches, they failed to do so since the search spaces are large and the correct patches were not actually found.

For 5 of the bugs, there were patches generated by the repair techniques. In Table 4.4, the first column shows the bugs and the repair techniques. The fourth column shows that there were in total 11 different patches generated. Among the 11 patches, we found 4 patches generated by HDRepair for *Math_50* are correct: they essentially remove the faulty statement *x0 = 0.5 * (x0 + x1 - delta)* (see *https://github.com/qixin5/DiffTGen/tree/master/expt1* for these 4 patches, all the other generated patches and all the generated test cases). We also found two patches generated by jGenProg for *Math_95* are invalid: they did not pass the test cases previously generated by DiffTGen. We next ran DiffTGen again for the other five (11-4-2) patches and the corresponding bugs. As the result shown in Table 4.5, DiffTGen identified two overfitting patches, *Chart_26_jKali* and *Chart_15_jKali*, with the corresponding test cases generated. We added each test case to the bug's test suite, and then ran jKali to repair the two bugs again. This time, no patches were generated by jKali. For the other three patches (*Chart_13_Nopol*, *Math_50_HDRepair_0*, and *Math_50_HDRepair_1*), we believe they are overfitting and incorrect, but DiffTGen did not produce any overfitting-indicative test cases[13].

4.4.3 Discussion

We conducted two experiments showing the *feasibilities* of (1) using DiffTGen to identify overfitting patches within a short amount of time (a few minutes) and (2) combining DiffTGen with a repair technique to enhance the technique's reliability.

In the experiments, DiffTGen used a bug-fixed version as the oracle. In general, however, we need a human oracle, and DiffTGen should provide testing information that is human-amenable. DiffTGen employs EvoSuite to generate test methods. There are cases where EvoSuite failed to generate any test methods exercising any changes that the patch makes. We think using more sophisticated techniques (e.g., [179]) may improve this but may also take more time to run and make DiffTGen less scalable. Given the fact that the current version of DiffTGen runs fast, we believe it could always be used for a first trial.

4.5 Summary

Automated program repair techniques often produce overfitting patches which do not actually repair the bugs. In this chapter, we presented a patch testing technique DiffTGen which could identify overfitting patches through test case generation. We demonstrated through experiments the feasibility of using DiffTGen in the context of automated program repair: DiffTGen can identify about a half of the overfitting patches with test cases generated in only a few minutes. An APR technique,

[13]Two of the three patches (*Chart_13_Nopol* & *Math_50_HDRepair_0*) make changes on statements created for instrumentation. It could be avoided but involves modifying a repair technique.

if configured with DiffTGen, could produce less overfitting patches and more correct patches. Future work may look at (1) using DiffTGen to identify overfitting patches generated by other APR techniques (including our techniques ssFix and sharpFix), (2) checking whether DiffTGen can help many other APR techniques in producing less overfitting patches and more correct patches, (3) optimizing DiffTGen with more sophisticated test generation techniques, and (4) making DiffTGen more practical by using a human oracle.

Chapter 5

Revisiting ssFix for Better Program Repair

In Chapter 3, we showed our APR technique ssFix which finds and reuses existing code fragments from the local faulty program and an external code repository to do bug repair. We demonstrated through experiments that ssFix was reasonably effective and outperformed five other APR techniques in producing more valid patches for the Defects4J bugs. Nevertheless, ssFix was only able to repair a small fraction of the Defects4J bugs (it repaired 20 of the 357 bugs with valid patches generated), and we do not know much about its potential since (1) it is not clear whether its built-upon assumption, i.e., reusing existing code for bug repair generally works and (2) whether ssFix's code search and code reuse methods are truly effective. In the first part of the chapter, we show the experiments we conducted to answer (1) and (2). For (1), we found the idea of reusing existing code for bug repair is promising. For (2), we evaluated ssFix's code search and code reuse, and found that there is still room for improvement. We developed sharpFix as an improved version of ssFix. In the second part of the chapter, we elaborate on how sharpFix works and show the experiments we conducted for its evaluation. We found that compared to ssFix, sharpFix has better code search, code reuse, and repair abilities. For the 357 Defects4J bugs, sharpFix repaired in total 36 bugs with correct patches generated, and outperformed existing techniques (that were evaluated on the same dataset) in producing more correct patches.

5.1 Introduction

One major problem faced by current search-based APR techniques is the search space problem [133]. A search-based APR technique often defines a huge search space of patches to support repairing different types of bugs. However, searching for a correct patch in such a huge search space is often difficult [133]. To address the problem, we proposed our APR technique ssFix in Chapter 3 which performs syntactic code search to find existing code fragments (from the local faulty program and

an external code repository) that are similar to the bug context and reuses those code fragments to produce patches for bug repair. In Chapter 3, we showed that ssFix worked reasonably well: it generated valid patches for 20 bugs in the Defects4J bug dataset with the median running time for generating a patch being only about 11 minutes and it outperformed five other APR techniques for Java [116, 183, 234, 113, 232].

Though ssFix seems like a promising repair technique, we do not know much about its potential. For bug repair, ssFix implicitly makes the assumption that existing programs (the local faulty program and the non-local programs in the code repository) contain the fix ingredients as the statements/expressions that are needed for producing a correct patch. However, we do not really know how often the assumption holds in practice. Furthermore, assuming the fix ingredients do exist in existing programs, we do not know whether ssFix is truly effective at finding the fix ingredients and reusing them to produce a correct patch.

In the first part of the chapter, we show the experiments we conducted to (1) test the fix-ingredient-assumption (i.e., to investigate whether the fix ingredients for bug repair often exist) and to (2) evaluate ssFix's abilities in finding and reusing the fix ingredients for bug repair. We looked at real bugs in the Defects4J bug dataset [94] for our experiments. For (1), we defined the fix ingredient for a bug in the context of automated program repair (we looked at six types of modifications commonly used by existing APR techniques and defined the fix ingredient for each modification, see Section 5.2.1) and we performed syntactic code search to check whether the fix ingredient exists (with and without any parameterization) in the local faulty program and in the non-local programs in a code repository. There are existing studies that also test the fix-ingredient-assumption. Some of the studies [31, 203] only looked at whether the fix ingredient exists at the level of code lines. A fix ingredient does not have to be an entire code line to be reused for bug repair (we will use an example to explain this in Section 5.2.1). The others [146, 160] do not actually fit our context (for example, Martinez et al. [146] studied whether the fix ingredient exists in any previous versions of the current program. The previous versions of a faulty program may not be available at the repair time, and ssFix does not look at using any previous versions of the faulty program for bug repair). For (2), we conducted multiple experiments looking at (a) whether ssFix can effectively find the fix ingredients that do exist in existing programs, (b) whether ssFix can effectively reuse the fix ingredients it found to do bug repair, and (c) whether ssFix can do effective repair with the fault being accurately located.

For (1), our results show that the idea of reusing existing code for bug repair is promising (Section 5.2): For the 103 simple bugs we experimented with, we retrieved the exact fix ingredients (needed for producing the correct patches) for 50 bugs and the parameterized fix ingredients for 80 bugs. This implies that a possibly effective approach for producing patches is to reuse existing code (at least as an initial effort) rather than to create artificial code from scratch. For (2), our results show that ssFix retrieved code fragments that contain the parameterized fix ingredients for 61 simple bugs. After translation, for 38 bugs, the retrieved code fragments contain the exact fix ingredients (Section 5.3.1). Our results show that ssFix reused the retrieved code fragments to produce correct

patches for 23 bugs (Section 5.3.2).

Based on our experimental observations, we developed a new repair technique sharpFix. sharpFix uses different code search methods to retrieve code fragments from the local faulty program and from the non-local programs in the code repository. For patch generation, sharpFix goes through the same steps used by ssFix: code translation, code matching, and modification. Each step however is different and improved. We evaluated sharpFix and found that compared to ssFix, sharpFix has better code search and code reuse abilities: sharpFix targets on repairing relatively simple bugs and it retrieved code fragments that contain the parameterized fix ingredients for 59 bugs. After translation, for 42 bugs, the code fragments contain the exact fix ingredients (Section 5.4.2). sharpFix reused the retrieved code fragments to produce correct patches for 30 bugs (Section 5.4.3). An APR technique may consider using sharpFix's code search to obtain a search space of fix code (or the fix space) and then leveraging code in the fix space to produce patches (for example, GenProg [116] may obtain the fix space using sharpFix's code search, and leverages its genetic algorithms to produce patches using code in the fix space). An APR technique that can identify its fix space may also use sharpFix's code reuse method to actually produce patches. Our results also show that sharpFix is better than ssFix in doing bug repair (Section 5.4.4). For a full repair experiment, sharpFix produced correct patches for 36 of the 357 Defects4J bugs. To our knowledge, compared to existing APR techniques that were evaluated on this dataset, it produced the largest number of correct patches.

5.2 Testing the Fix-Ingredient-Assumption

For bug repair, ssFix was built upon the assumption that existing code from the code database (which consists of the local faulty program and the non-local programs in a large code repository) contains the fix ingredient needed for producing a correct patch. After identifying a suspicious statement in the faulty program using fault localization, ssFix performs syntactic code search to find code that contains the fix ingredient from the database to be reused for repair. We conducted experiments to test the fix-ingredient-assumption, i.e., to check how often the fix ingredient for bug repair exists in the local and non-local programs. In this section, we first give the definition of a fix ingredient, then show the experiments we conducted and the results we got. For our experiments, we looked at relatively simple patches. We consider a patch to be simple if the fixing changes are made within an expression or a primitive statement which contains no children statements. A more complex patch can be considered as being made of more than one simple patch.

5.2.1 Defining the Fix Ingredient

Some Possible Definitions

For bug repair, there can be different ways of defining a fix ingredient. In the simplest case, we can define the fix ingredient as the exact change(s) between the faulty program and the correctly

patched program. But this definition is often not ideal. Consider the correct patch for the bug M85 as shown below (this is the developer patch provided by Defects4J).

```
- if (fa*fb>=0.0) {
+ if (fa*fb>0.0) {
    throw new ConvergenceException(...);
  }
```

For this bug, we may consider the exact change, i.e., the operator `>`, as the fix ingredient. Then we can find lots of code fragments from the code database that contain `>`. But very few of those code fragments can actually be useful for repair since very few of those code fragments that contain `>` are indeed relevant to the bug context.

Another way of defining a fix ingredient is to consider the fix ingredient as the patched statement. However, this definition is not ideal either. The patched statement might be too project-specific, and it is not likely to find the exact statement from the code database (either from the local program or from any non-local programs in the code repository). For the example above, it is not likely to find the exact patched statement that (1) compares the multiplication of two variables (i.e., `fa*fb`) and `0.0` and (2) throws the correct type of exception (possibly with the exact exceptional message) from existing programs.

Another way of defining a fix ingredient (as used in [146, 31, 203]) is to consider the fix ingredient as the patched line(s). This was used for analyzing how many of the patched lines for a bug-fix can be found from existing code. A fixing change as multiple patched lines or even as a single patched line can be too unique and not ideal to be used as the fix ingredient. For the above example, using the patched code line as the fix ingredient, we would miss any opportunities of finding the correct expression as a non-if-condition.

For the example shown above, we define the fix ingredient to be the conditional expression `fa*fb>0.0`, and we successfully found the exact fix ingredient from the local faulty program in the following loop statement. (For this bug, this is the only statement in the local project where `fa*fb>0.0` is contained.)

```
do {
  ...
} while ((fa*fb>0.0) && ...);
```

Comparing the loop statement with the faulty if-statement, it is relatively easy for a repair technique (e.g., ssFix) to identify `fa*fb>=0.0` and `fa*fb>0.0` to be related. Then a simple modification by changing `>=` to `>` would produce the correct patch.

Our Definition

We now give our definition of a fix ingredient. For bug repair, we looked at six types of repairing modifications. For each type of modification, we defined the corresponding fix ingredient. The six types of modifications (M0-M5) are as shown below.

- **M0**: Combining a Boolean condition with another Boolean condition using `&&` or `||` (e.g., `if (c0) {...}` → `if (c0 || c1) {...}`)
- **M1**: Changing an expression (as a non-if-condition) to another expression (also as a non-if-condition)[1] (e.g., `e0` → `e1`)
- **M2**: Changing an if-condition to another if-condition (e.g., `if (c0) {...}` → `if (c1) {...}`)
- **M3**: Adding an if-condition for one or more statements (e.g., `s` → `if (c) {s}`)
- **M4**: Replacing a statement with another statement (e.g., `s0` → `s1`)
- **M5**: Inserting a statement (e.g., `s` → `s0; s`)

We used the above six types of modifications to model a general repair modification. The six types of modifications are commonly used by current APR techniques [116, 132, 134, 232, 230]. In fact, for all the bugs we used in our experiments, the repairing modifications can be successfully modeled by at least one of the six types. (Note that the deletion of a statement can be modeled by M4 through replacing the statement with an empty statement.)

After defining the six types of repairing modifications, we defined the corresponding fix ingredients (FIs) as shown below.

- **FI0**: The Boolean condition that is combined with the original Boolean condition (`c1` in **M0**'s example)
- **FI1**: The parent statement/expression[2] of the changed expression (the parent statement/expression of `e1` in **M1**'s example)
- **FI2**: The changed if-condition (`c1` in **M2**'s example)
- **FI3**: The added if-condition (`c` in **M3**'s example)
- **FI4**: The replaced statement[3] (`s1` in **M4**'s example)
- **FI5**: The inserted statement (`s0` in **M5**'s example)

For M0, M2, and M3, we consider the combined, changed, and added if-conditions as the fix ingredients respectively. For M4 and M5, we consider the replaced and inserted statements as the fix ingredients respectively. For M1, it is often not ideal to simply consider the changed expression (i.e, `e1` in **M1**'s example) as the fix ingredient since the changed expression can be too small and thus be lack of context (consider `e1` to be a variable argument of a method call). So instead, we consider the parent statement/expression of the changed expression as the fix ingredient (in the abstract syntax tree, this is the parent node of the changed expression node).

[1] If the changed expressions are if-conditions, we use M2 to model the repair.

[2] This is the statement/expression that represents the parent node in the abstract syntax tree of the node that is represented by the changed expression.

[3] Note that such a statement can be empty. Replacing a statement with an empty statement is equivalent to deleting the statement.

5.2.2 Experiment for Assumption Testing

Setup

We used the Defects4J bug dataset (version 0.1.0) [94] as our experimental dataset. The dataset contains 357 real bugs in total. For each bug, the dataset provides a developer patch which we considered as the correct patch. We manually examined all the 357 developer patches and identified 103 simple patches. Each of the 103 patches does only simple modifications and creates only one patched statement. In other words, for each such simple patch, we can identify one (and only one) statement as the first parent statement for all the modifications that the patch makes. Our experiment is to check whether the fix ingredient for each such patch exists. (Note that a simple patch may delete a statement but we ignored such patches. We consider the fix ingredients for such patches to always exist.) For each of the 103 simple patches, we manually identified the fix ingredient by first classifying the patch as one of the six types of modifications that we introduced earlier and then extracting the fix ingredient associated with the classified modification type. In fact, there can be more than one modification type as which a patch can be classified. To do the classification, we looked at the modification types in an sequential order from M0 to M5, and identified the first type as which the patch can be classified. For example, a patch that changes the condition of an if statement can be classified as either M2 or M4, and we actually classified such a patch as M2. For each of the 103 simple patches, we performed code search to see whether the fix ingredient is contained in either the local faulty program or any non-local program in a code repository for which we used the DARPA MUSE code repository [55] consisting of 66,341 Java projects (about 81G).

Search Methodology

For each simple patch, using the identified fix ingredient as query, we performed syntactic code search to check whether the fix ingredient can be found in the code database (the local faulty program plus all the programs in the code repository). In our definition, a fix ingredient can only be an expression or a statement. So we extracted every statement within every method in the code database. We tokenized the fix ingredient and every extracted statement. The goal is to check whether the fix ingredient's tokens are a subsequence of any extracted statement's tokens. Ideally, the exact fix ingredient can be found. But this may be too strong since different programs may use different names for variables, types, and methods. So in addition to searching for the exact fix ingredient, we also searched for the parameterized fix ingredient. The parameterization only replaces program-specific (i.e., non-JDK) variables, types, and methods with special symbols (we used `$v$` for variables, `$t$` for types, and `$m$` for methods).

To find the fix ingredient within the local faulty program, we checked whether the fix ingredient's tokens are exactly a subsequence of any extracted statement's tokens with and without parameterization. To find the fix ingredient within the code repository, we could do the same, but this can be very expensive (for each of the 103 bugs, we would have to iterate every statement in the large code repository). So instead, we indexed every statement in the code repository using ssFix, and we

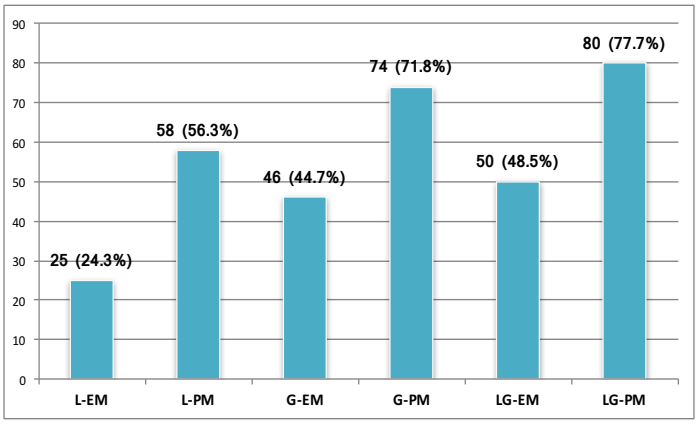

Figure 5.1: Fix Ingredient Retrieval Result (The x-axis shows the search types. L/G/LG-EM represents the Exact Match within the local program/the code repository/both; L/G/LG-PM represents the Parameterized Match within the local program/the code repository/both. The y-axis shows the numbers of bugs for which the fix ingredients were found. The label for each search type shows the number of bugs for which we found the fix ingredients using that search type and the percentage of the number of bugs (out of the 103 bugs) in parenthesis.

performed two steps to find the fix ingredient: (1) we did ssFix's code search using the fix ingredient's enclosing statement as the query to have the top-500 statements retrieved[4] and (2) we checked, for each of the retrieved statements, whether the fix ingredient's tokens are a subsequence of the statement's tokens. Although in theory, this type of code search may fail to find some statements in the code repository that contain the fix ingredient, we think it can be a reasonable approximation. For code search within the repository, we filtered away any statement whose enclosing method's signature is identical to the faulty method's signature and whose enclosing package's name is identical to the faulty method's enclosing package's name. We did so to ignore any fix ingredients that simply come from any bug-fixed versions of the faulty program. (In Chapter 3, ssFix filters away bug-fixed programs in the same way.)

Results

Our results are shown in Figures 5.1 and 5.2. We found that reusing existing code for bug repair can be a promising approach. As shown in Figure 5.1, for 50 (48.5%) of the 103 bugs, we retrieved the exact fix ingredients from existing programs (either the local faulty program or any non-local program in the code repository). If the fix ingredient can be found within a code fragment that is reasonably small, a repair technique can possibly leverage the fix ingredient to produce a correct patch. Given the faulty statement of M85 (shown in Section 5.2.1), our developed technique sharpFix successfully found a small code fragment (i.e., the loop statement shown in Section 5.2.1) that contains the fix ingredient `fa*fb>0.0`. sharpFix identified the buggy if-condition `fa*fb>=0.0` to be related to

[4]Since a fix ingredient can be an expression, it would be better if we can index every expression in the code repository and then search for the fix ingredient at the expression level. We however chose not to do that simply because there are too many expressions contained in our repository.

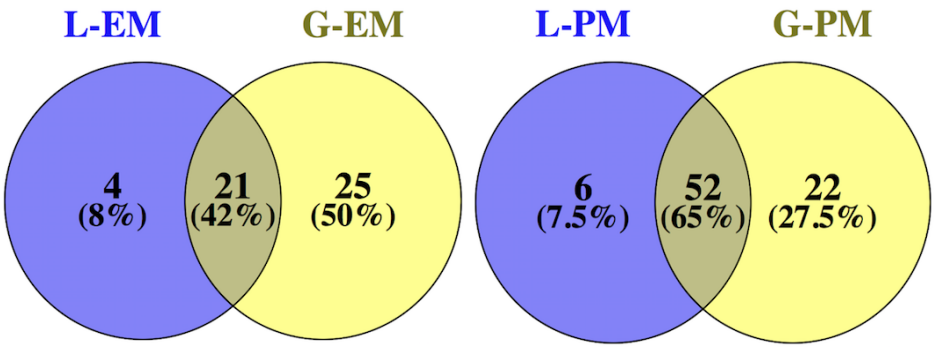

Figure 5.2: The Fix Ingredient Overlap Between L-EM & G-EM (left) and Between L-PM & G-PM (right)

`fa*fb>0.0` and produced the correct patch by condition replacement. For a total of 80 (77.7%) of the 103 bugs, we retrieved the fix ingredients in the parameterized forms. For 50 of the 80 bugs, we retrieved the fix ingredients not only in the parameterized forms but also in the exact forms. For the other 30 bugs, we retrieved the fix ingredients that are not in the exact forms but only in the parameterized forms. A repair technique may not be able to directly leverage a parameterized fix ingredient to produce a patch. Such a fix ingredient, however, can be leveraged with some translation (e.g., through identifier renaming). Below is the developer patch for M27. For this bug, we found a fix ingredient `100 * getItemCount()` from a return statement in the Java method named *getSizeBytes*.

```
- return multiply(100).doubleValue();
+ return 100 * doubleValue();
```

A repair technique may leverage the fix ingredient to repair the bug by first renaming `getItemCount` as `doubleValue` and then replacing the faulty expression with the renamed fix ingredient.

According to our results, looking for a fix ingredient within the local faulty program (or the local search) only is insufficient. For only 25 and 58 bugs, the exact and parameterized fix ingredients exist in the local program respectively. In comparison, looking for a fix ingredient within the non-local programs in the code repository (or the global search) is more likely to succeed. However, as Figure 5.2 shows, there are fix ingredients that only exist in the local faulty program. So it might be better to perform both local and global searches to possibly find a fix ingredient.

Note that it is possible for a repair technique to produce a correct patch for a bug although it failed in finding the fix ingredient we defined. So our results actually provide a lower-bound. To understand this, for repairing the bug Cl10, the technique we developed sharpFix found a method call `anyResultsMatch(n.getFirstChild().getNext(), p)`. Though we did not recognize this method call as the fix ingredient `anyResultsMatch(n, MAY_BE_STRING_PREDICATE)` because of the arguments, sharpFix did successfully reuse this method call to produce the correct patch (by replacing the buggy method call's name `allResultsMatch` with `anyResultsMatch`).

For our experiments, we only looked at simple patches. One may apply our method to deal with

a more complex patch by first dividing the complex patch into simple patches and then searching for the fix ingredient for each simple patch. This corresponds to a natural way of repairing a complex bug which requires multiple simple patches: It is not likely for a repair technique to produce the simple patches all at once, but to produce them one by one. Hopefully the repair technique can identify some progress it has made after producing each simple patch.

5.3 Analyzing ssFix

ssFix is a recent APR technique that finds and reuses existing code fragments from a code database (that consists of the local buggy program and non-local programs in a large code repository) for bug repair. In Chapter 3, the results of our repair experiments showed that ssFix was relatively effective and outperformed existing techniques in repairing more bugs in the Defects4J dataset. However, the overall repair ability of ssFix was still limited: it failed to repair more than 90% of the bugs. For a better understanding of ssFix, we conducted more experiments to evaluate ssFix's code search and code reuse abilities. ssFix relies on an existing technique [40] to do fault localization which was shown (in Section 3.4 of Chapter 3) to work poorly. To understand its repair potential, we conducted repair experiments with accurate fault localization results provided at the statement level and at the method level, and compared the results to those obtained from a fully automatic repair experiment.

5.3.1 Evaluating ssFix's Code Search

Given a faulty program and a fault-exposing test suite, ssFix does fault localization to identify a list of suspicious statements in the program that are likely to be faulty. For each suspicious statement, it performs code search to find candidate code chunks in the code database to be reused for bug repair. As one way to evaluate ssFix's code search, we looked at the 103 bugs used in Section 5.2.2 whose developer patches are simple. For each bug, we provided ssFix with the real faulty statement and checked whether ssFix can effectively retrieve any candidates that contain the fix ingredient (that we identified earlier in Section 5.2.2). We call a candidate (possibly after translation) that contains the exact fix ingredient *promising*. Our results show that ssFix retrieved promising candidates within the top-500 results for 38 bugs.

Experiment

For each of the 103 bugs whose fix ingredients we identified in Section 5.2.2, we identified the faulty statement[5] in the faulty program, provided ssFix with the statement, ran its code search to retrieve a list of ranked candidates (as code chunks) from the code database, translated the candidates using ssFix's code translation (otherwise it may not be able to reuse the fix ingredient for repair), and checked whether any of the candidates is promising, i.e., contains the exact fix ingredient. The

[5]For a bug whose patch inserts a statement in between two contiguous statements, we used each of the two statements as the faulty statement, ran ssFix's code search twice (once for each statement), and used the better result.

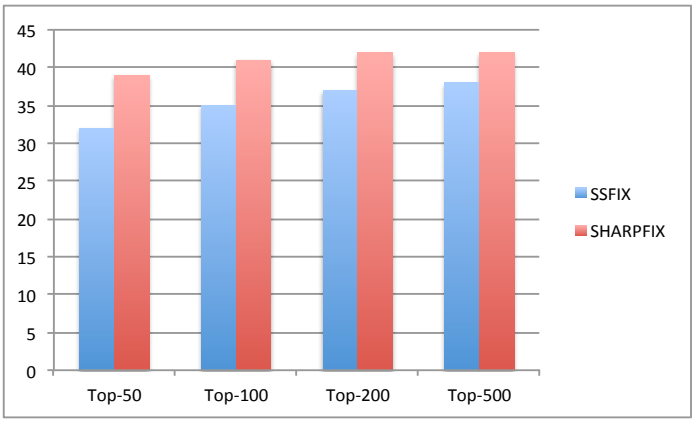

Figure 5.3: The Retrieval of Candidates that Contain the Fix Ingredients (We looked at the top-50, 100, 200, and 500 candidates. The column shows the number of bugs for which we retrieved promising candidates)

code database we used consists of the DARPA MUSE code repository (as used in Section 5.2.2) as the external code repository and the five projects used in Section 3.4 of Chapter 3 (i.e., the projects of C8, Cl14, L6, M33, and T4) as the local programs. We filtered away candidates that are syntactically redundant and those that are simply from the bug-fixed versions. We looked at the top-500 candidates as the retrieval results.

Result

Figure 5.3 shows the numbers of promising candidates retrieved by ssFix within the top-k results (with k being 50, 100, 200, and 500 respectively). Within the top-500 results, ssFix retrieved promising candidates for 38 bugs: it retrieved in total 61 candidates that contain the fix ingredients in the parameterized forms, among the 61 candidates, 38 are promising, i.e., contain the exact fix ingredients after translation. In Section 5.2.2, we found the fix ingredients that exist in the parameterized forms in our code database for as many as 80 bugs. ssFix retrieved promising fix ingredients for 38/80=47.5%[6] of such bugs.

5.3.2 Evaluating ssFix's Code Reuse

ssFix looks at a candidate it retrieved and tries to reuse the candidate to produce a plausible patch for the target code chunk (which contains the located suspicious statement). To reuse the candidate for bug repair, ssFix first translates the candidate (by renaming the identifiers used in the candidate as those that are related in the target), matches the statements and expressions between the target and the translated candidate, and performs three types of modifications: *replacement*, *insertion*, and *deletion* to produce a set of patches. Next, ssFix sorts the patches by their types and sizes, validates

[6]Note that this is only a lower bound because not all the parameterized fix ingredients for the 80 bugs can be reasonably translated and reused for bug repair.

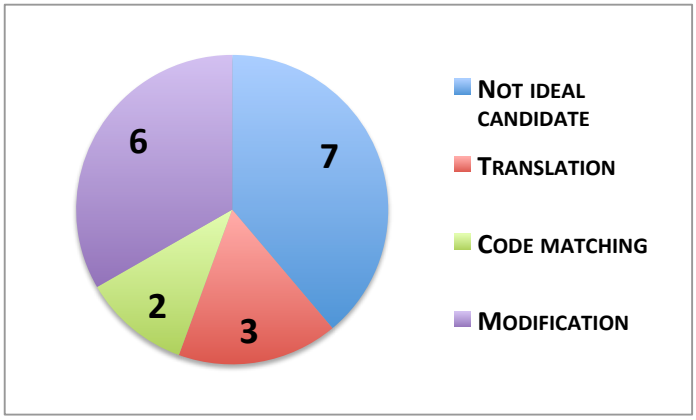

Figure 5.4: Understanding ssFix's Code Reuse Failures

each patch in the sorted order, and reports the first plausible patch (if found) whose patched program can pass the test suite.

To evaluate ssFix's code reuse ability, we looked at the 61 bugs for which ssFix retrieved candidates that contain the fix ingredients in the parameterized forms. For each bug, we provided ssFix with the target (that it produced for code search & repair) and the retrieved candidate, and checked whether ssFix can produce a plausible patch that is correct (i.e., semantically equivalent to the developer patch). Our results show that ssFix reused the retrieved candidates to produce correct patches for 23 bugs.

Experiment

For each of the 61 bugs, we provided ssFix with the target and the retrieved candidate, and ran ssFix's patch generation and patch validation automatically to possibly produce a plausible patch. If ssFix produced such a patch, we manually checked whether the patch is correct (by comparing the patch to the developer patch provided by Defects4J).

Result

Our results show that ssFix produced 25 plausible patches among which 23 are correct. It successfully reused 23/61=37.7% candidates for bug repair. Note that this is a lower bound because not all the candidates can be reasonably reused for bug repair. We found that the exact fix ingredients (without any translation) are contained in 38 candidates, and we expect ssFix to be able to reuse those fix ingredients in producing the correct patches. For other 23 (61-38) candidates which only contain the fix ingredients in the parameterized forms, we manually determined whether they can be reasonably reused for bug repair. We identified only 3 candidates to be reasonable for reuse (it may not be reasonable for a repair technique to translate an arbitrary, parameterized fix ingredient into the exact one to be reused for repair). We analyzed the failure of ssFix in reusing the 18 (38+3-23) reasonable candidates for producing the correct patches. We looked into its reusing processes to

understand the failures. Figure 5.4 shows our analyzing results for the 18 failures. We found that 7 candidates are not ideal to be reused for bug repair. As an example, for the bug Cl119, ssFix retrieved a candidate containing the exact fix ingredient `case Token.CATCH:` as a statement to be inserted in the target for producing the correct patch. However, the fix ingredient is embedded in a big switch statement, and it is therefore difficult for ssFix to leverage the fix ingredient to do the correct repair. For the other failures, we found that ssFix yielded bad candidate translations for 3 cases, it created bad code matching results for 2 cases, and its modifications are not sophisticated enough to produce the correct patches for 6 cases. We identified the key shortcomings of ssFix in translation, code matching, and modification, and developed sharpFix as an improvement. More details can be found in Section 5.4.

5.3.3 Evaluating ssFix's Repair

Based on the fix ingredients we retrieved in Section 5.2.2, we conducted experiments in Section 5.3.1 and Section 5.3.2 to evaluate ssFix's code search and code reuse abilities respectively. We also conducted experiments to evaluate ssFix's repair abilities. We conducted one experiment to evaluate ssFix's full repair ability. In this experiment, we ran ssFix to repair all the 357 Defects4J bugs automatically. We conducted two experiments to evaluate ssFix's partial repair abilities. For the experiments, we identified two sets of bugs (that contain 112 and 201 bugs respectively) for which we can identify single faulty statements and single faulty methods respectively. We provided ssFix with the fault-located statement and method and ran ssFix to do bug repair. We compared the results to those we obtained in the full repair experiment for the two sets of bugs, and found that with the faults known in advance as the faulty statement and the faulty method, ssFix can produce correct patches for 24% and 23% more bugs respectively.

Experiment

For the full repair experiment (E0), we ran ssFix to repair all the 357 Defects4J bugs. For the first partial repair experiment (E1), we manually identified 112 Defects4J bugs for repair. For each of the 112 bugs, the developer patch provided by Defects4J makes changes for only one statement[7], and we manually identified a single faulty statement and provided ssFix with the statement for bug repair[8]. In the case where the developer patch inserts a statement for bug repair, we identified the inserted statement's two adjacent statements (at most) in the inserted statement's block, considered each adjacent statement as the faulty statement, ran ssFix to repair the statements each, and used the better result. For the 112 bugs, we identified in total 128 faulty statements, and ran ssFix to repair each. For E2, we manually identified 201 bugs. For each of the 201 bugs, the developer patch

[7]Note that we considered only the first parent statement (in the faulty program) where a fixing change was made. A developer patch can make multiple changes for a single if-statement on its multiple children statements. In such case, we ignored the bug.

[8]Note that the bugs we used for E1 are not the same bugs we used for testing the fix-ingredient-assumption in Section 5.2.2. In Section 5.2.2, we looked at bugs whose patches are simple. But here for E1, we looked for every bug for which we can identify a single faulty statement.

Table 5.1: The Results of E0 (the Full Repair Experiment)

Project (#Bugs)	sharpFix						ssFix					
	Time (min.)				#P	#C	Time (min.)				#P	#C
	Min	Max	Med	Avg			Min	Max	Med	Avg		
C (26)	0.8	115.7	7.2	19.2	9	4	1	80.7	12.4	20.7	7	2
Cl (133)	1.8	96.1	21.3	26	17	4	2.5	54.9	10.1	16.3	14	2
M (106)	0.7	118.5	11.3	33.2	33	13	1	119.3	14.7	30.2	26	8
T (27)	1.6	30	12.2	15.1	5	0	1.4	37.3	7.5	13.5	4	0
L (65)	0.8	116.1	4.8	18	25	15	0.8	117.8	4.3	13.1	18	10
Sum (357)	0.7	118.5	11.3	25.1	89	36	0.8	119.3	10.1	21	69	22

We show the projects in their abbreviations: C is JfreeChart; Cl is Closure Compiler; M is Commons Math; T is Joda-Time; and L is Commons Lang. #P is the number of plausible patches generated. #C is the number of correct patches generated.

Table 5.2: Comparing the Results of E0 & E1 on the 112 bugs

EID	sharpFix						ssFix					
	Time (min.)				#P	#C	Time (min.)				#P	#C
	Min	Max	Med	Avg			Min	Max	Med	Avg		
E0	0.8	118.5	6.8	18.3	48	29	0.8	91	9.8	16.5	38	21
E1	0.8	42.7	2.8	4.2	50	39	0.8	15.5	1.9	3	42	26

In E1, for each bug, the repair technique (either sharpFix or ssFix) only targeted on repairing the faulty statement we provided without doing any fault localization. This explains why the running times of E1 (for either technique) are much shorter than the running times of E0.

For some bugs in E1, we provided two faulty statements for repair. For each of these bugs, either sharpFix or ssFix produced two repairing results (one on each statement). We took the better one as the repairing result for this bug. One result is better than the other if, for example, it can show that the repair technique produced a correct patch but the other result cannot.

makes changes within only one method. We manually identified such a method, ran ssFix's fault localization to obtain a list of suspicious statements within the method, and provided the list of statements as the fault localization result to ssFix.

As in Section 5.3.1 and Section 5.3.2, ssFix used a code database that consists of the DARPA MUSE code repository plus the five local Java projects used in Section 3.4 of Chapter 3. ssFix did not use any candidates that are from the bug-fixed versions of the faulty programs. We set the maximum number of candidates used by ssFix for repairing each suspicious statement to be 200. We set the time budget and memory budget for repairing each bug as two hours and 8G for all the three experiments. We ran all the experiments on a machine with 32 Intel-Xeon-2.6GHz CPUs and 128G memory.

Results

The results of the three repair experiments can be found in Tables 5.1 to 5.3. For E0, ssFix produced in total 69 plausible patches with the median and average time of producing a plausible patch being about 10 and 21 minutes respectively. Among the 69 plausible patches, 22 are correct. We define a patch to be correct if it is semantically equivalent to the developer patch provided by Defects4J. We manually determined the correctness of a plausible patch by comparing it to the developer patch.

Table 5.3: Comparing the Results of E0 & E2 on the 201 bugs

EID	sharpFix						ssFix					
	Time (min.)				#P	#C	Time (min.)				#P	#C
	Min	Max	Med	Avg			Min	Max	Med	Avg		
E0	0.8	118.5	9.1	25	69	36	0.8	119.3	10.6	22.8	54	22
E2	0.6	114.8	3.8	11.8	77	43	0.6	117.1	3.5	13	61	27

In E2, for each bug, the repair technique (either sharpFix or ssFix) only targeted on repairing the faulty statements we provided within the real faulty method. This explains why the running times of E2 (for either technique) are often shorter than the running times of E0.

Table 5.2 shows ssFix's running result for the 112 bugs used for E1. In the table, we provided the repairing results of these bugs from E0 for comparison. For these 112 bugs, in E0, ssFix produced correct patches for 21 bugs, but in E1 with the located faulty statement provided, ssFix produced correct patches for 5 (5/21=23.8%) more bugs. In E1, the running time for producing a plausible patch is much smaller (compared to E0) because ssFix only worked on one statement.

Table 5.3 shows ssFix running result for the 201 bugs used for E2. In the table, we provided the repairing results of these bugs from E0 for comparison. Compared to E0, ssFix produced correct patches for 5 (5/22=22.7%) more bugs. Compared to E0, the running time for producing a plausible patch is smaller in E2 because ssFix only worked on the suspicious statements within only one method.

5.4 sharpFix

In earlier sections of the chapter, we conducted an experiment to test the fix-ingredient-assumption and found that it is promising to reuse existing code for bug repair. We also conducted experiments to evaluate ssFix's code search, code reuse, and repair abilities, and found that ssFix can still be improved. We developed the new APR technique sharpFix as an improved version of ssFix and demonstrated through experiments it has better code search, code reuse, and repair abilities. In this section, we show how sharpFix works, the experiments we conducted, and the results we got. For a full repair experiment, sharpFix produced correct patches for 36 of the 357 Defects4J bugs. To the best of our knowledge, it outperformed all existing APR techniques that were evaluated on the Defects4J bug dataset in terms of the number of bugs that are successfully repaired (with correct patches generated).

5.4.1 Overview

sharpFix is an APR technique that reuses existing code fragments from a code database (which consists of the local faulty program and the non-local programs in a code repository) to do bug repair. Similar to ssFix, it goes through four stages: *fault localization*, *code search*, *patch generation*, and *patch validation* to possibly produce a plausible patch. sharpFix uses ssFix's method to do fault localization. Its methods for code search and code reuse (i.e., patch generation and patch

```
1 + if (r!=null) {
2     Collection c = r.getAnnotations();
3     Iterator i = c.iterator();
4     while (i.hasNext()) { ... }
5 + }
```

Figure 5.5: The Developer Patch for the C4 Bug

validation) however are different from ssFix's methods. For code search, sharpFix uses different methods for searching candidates from the local faulty program and from the code repository and combines the results (i.e., the retrieved candidates) for reuse. For patch generation, sharpFix goes through the same steps used by ssFix: *code translation*, *code matching*, and *modification*, but each step is different. For patch validation, sharpFix's method is identical to ssFix's method except that sharpFix performs the static resolving technique used by S^6 [185] as an extra step to check the validity of a patch (e.g., whether it used an undeclared variable) before dynamically compiling the patched program and testing it against any test cases.

We next elaborate on sharpFix's code search and code reuse methods and show the experiments we conducted for evaluation. We will use the Defects4J bug C4 (shown in Figure 5.5) as an example to explain how sharpFix works. For this bug, the developer patch (provided by Defects4J) produced an if-condition (at Line 1) to guard the three statements (at Lines 2-4) to avoid a null-pointer exception. Though the repair is relatively easy, ssFix and other existing techniques [116, 183, 234, 113, 232, 47] that were evaluated on the Defects4J bugs all failed to produce the correct patch (we looked at their generated patches from [143, 60, 9, 1, 47, 230]). sharpFix successfully produced the correct patch (i.e., the developer patch) in about 9.7 minutes[9]. For this bug, sharpFix's (also ssFix's) fault localization identified the statement at Line 2 as the top suspicious statement for repair.

5.4.2 Code Search

Motivation for Improving ssFix's Code Search

For code search, ssFix uses relatively small code chunks. For producing the target, it looks at the located suspicious statement, and produces a code chunk that contains the statement possibly with its two neighbouring statements (if the suspicious statement is not too large, i.e., within six lines of code). For candidates, it produces code chunks that contain (1) every sequence of three contiguous statements within every block and (2) every compound statement (which contain children statements) from every Java method in the code database. ssFix uses the same search method for both local (within the local faulty program) and global (within the code repository) code search.

It is possible to obtain better code search results using code chunks that are smaller or larger (than ssFix's code chunks) and using different search methods for local and global code search (a possibly

[9] Although sharpFix successfully repaired the bug, it is not specifically designed to repair null-pointer errors. For this example, we believe a repair technique that targets on such errors (e.g., [51]) can be more effective at repairing this bug (e.g., it may work faster).

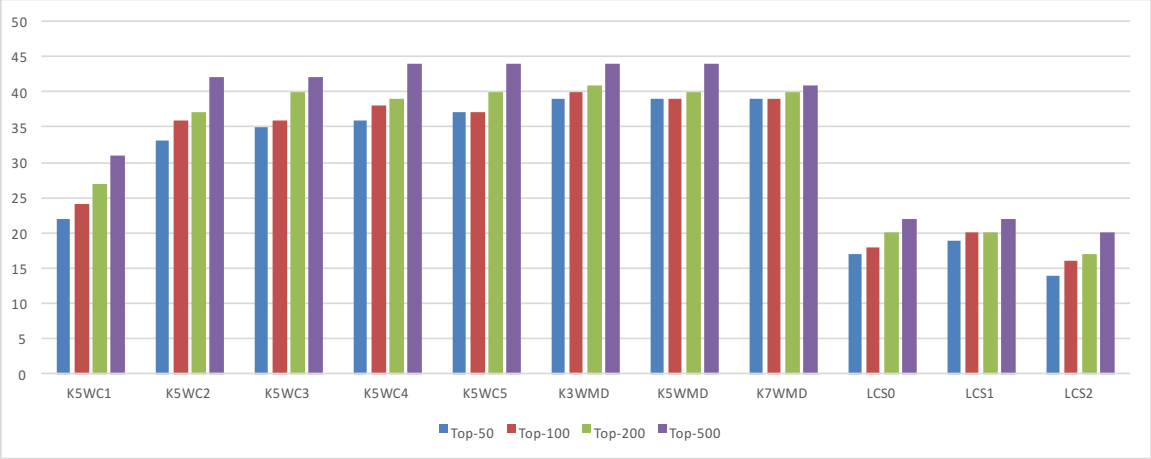

Figure 5.6: Comparing Different Search Methods (the x-axis shows the search methods and the y-axis shows the numbers of bugs whose fix ingredients were retrieved)

better way to do local code search, compared to global code search, is to use less parameterization for names). To investigate these possibilities, we conducted experiments comparing a set of different code search methods.

More specifically, we re-ran ssFix's code search experiments (Section 5.3.1) using different sizes of code chunks: we created six code search methods K5WC1, K5WC2, K5WC3, K5WC4, K5WC5, and K5WMD that look at using code chunks containing one to five statements (K5WC1 to K5WC5) and the entire method (K5WMD). Note that these search methods are equivalent to ssFix's search method except that the used chunks are in different sizes. For each search method, we counted the number of bugs whose exact fix ingredients can be found from any retrieved candidate (with and without ssFix's translation[10]). Our results can be found in Figure 5.6: We retrieved the most candidates (44 within the top-500 results) that contain the fix ingredients using chunks at the method level (i.e., using K5WMD). ssFix extracts structural tokens as k-grams (where k=5) from the target and the candidate for calculating their similarity score. We investigated using different values of k: we created two more search methods K3WMD and K7WMD and ran ssFix's code search experiment with each. We found that K3WMD yielded the best results. It retrieved the most candidates containing fix ingredients. However, K3WMD is not significantly better than the other two methods: K5WMD and K7WMD. Compared to the second-best method K5WMD, K3WMD retrieved the same amount of candidates that contain the fix ingredients within the top-50 and top-500 results, and it retrieved only one more candidate within the top-100 and top-200 results each.

We investigated using three search methods: LCS0, LCS1, and LCS2 for local code search (within the local faulty program only). Given a code chunk (either the target or the candidate), the methods extract different tokens as shown below.

[10]We also checked whether the exact fix ingredient can be found in the original candidate (without doing any translation) because it is possible for ssFix to yield a bad translation.

- **LCS0**: The tokens are the textual contents (in the compacted forms with no whitespaces) of all the children expressions of the chunk statement(s).
- **LCS1**: The tokens are all the conceptual tokens that ssFix extracts[11] from the chunk statement(s) (with the Java keywords, stop words, and short and long words filtered by ssFix actually kept).
- **LCS2**: The tokens are all the conceptual words that ssFix extracts from the chunk statement(s) (with the Java keywords, stop words, and short and long words filtered by ssFix actually kept) plus any symbols that are non-Java-identifiers (e.g., $+=$).

Given the tokens (as two lists) extracted from the target and the candidate by LCS0, LCS1, or LCS2, sharpFix does token matching and uses the Dice Similarity[12] to calculate a similarity score. For each search method, we show below as an example, the extracted tokens (in angle brackets) of the statement at Line 2 in Figure 5.5.

```
LCS0 (7 in total):
<Collectionc=r.getAnnotations();>, <Collection>, <c=r.getAnnotations()>,
<c>, <r.getAnnotations()>, <r>, <getAnnotations()>

LCS1 (7 in total):
<collection>, <collect>, <c>, <r>,
<getannotations>, <get>, <annotations>

LCS2 (12 in total):
<collection>, <collect>, <c>, <=>, <r>, <.>,
<getannotations>, <get>, <annotations>, <(>, <)>, <;>
```

Note that the code chunk (either the target or the candidate) used for local code search contains only one statement. This is because within the local program, the uniqueness of code chunks containing multiple, sequential statements is high (much higher than the uniqueness of such code chunks within a large code repository). To evaluate the three search methods, we ran each search method to retrieve candidates within the local faulty program of each of the 103 bugs and checked whether the fix ingredient (with and without any translation) can be found in any candidate from the top results. Our results can be found in Figure 5.6. We learned that LCS1 yielded the best result: it retrieved 22 candidates containing the fix ingredients from the top-500 results, though it is not significantly better than LCS0. Figure 5.7 shows that the best local search method LCS1 can complement the best global search method K3WMD in finding more candidates that contain the fix ingredients.

[11] We used ssFix's method for extracting conceptual tokens as described in Section 3.3.2.

[12] The original measure is used for sets. We changed it to be used for lists. Given two lists of tokens l_1 and l_2, we create a one-to-one mapping of tokens from l_1 to l_2. A token is mapped to another token that is identical to it. We use n to denote the number of mapped tokens in the two lists, n_1 to denote the number of tokens in l_1, and n_2 to denote the number of tokens in l_2. The Dice Similarity is computed as $(2 \times n)/(n_1 + n_2)$.

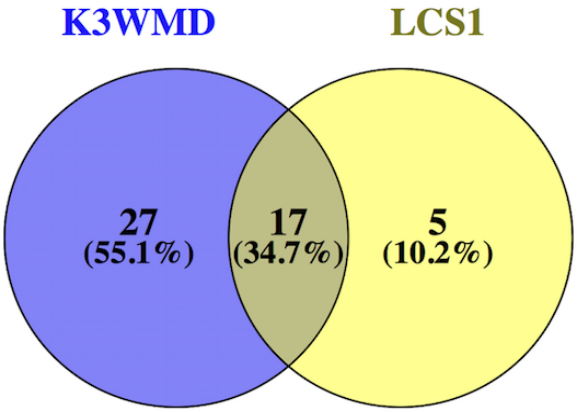

Figure 5.7: The Overlap of the Retrieved Candidates (by K3WMD & LCS1 within the top-500 results) that Contain the Fix Ingredients

Methodology

In Section 5.4.2, we showed that K3WMD and LCS1 yielded the best searching results for global and local code search respectively and that it is possible to combine the two to yield better results. Based on our findings, we developed sharpFix's code search method: sharpFix performs K3WMD and LCS1 to do code search and merge the searching results to obtain a list of candidates to be reused for bug repair.

More specifically, given a suspicious statement s, sharpFix produces two target code chunks: $tchunk_0$ which contains the enclosing method of s and $tchunk_1$ which contains s itself. Next it performs K3WMD using $tchunk_0$ as the query to obtain a list of candidate code chunks $cchunks_0$ (ranked by scores from high to low) from the code repository and it performs LCS1 using $tchunk_1$ as the query to obtain a list of candidate code chunks $cchunks_1$ (also ranked) from the local faulty program. Each $cchunk_0 \in cchunks_0$ is actually a Java method. sharpFix does not reuse all the code from the whole method to produce patches because it can generate too many patches for validation if the method is not too small. So instead, it identifies statements within the method that are relevant to s and might thus be useful for repair (we will show the evaluation of sharpFix's code search in the next sub-section) and reuses such statements (and possibly the statements in their local context) to produce patches. To obtain such code fragments, sharpFix first translates $cchunk_0$ into $rcchunk_0$ (using the translation method explained in Section 5.4.3), and uses LCS1 to identify two statements cs_0 and cs_1 in $rcchunk_0$ that are most similar to s. So for each $cchunk_0 \in cchunks_0$, sharpFix obtain two statements cs_0 and cs_1. Each statement is associated with the searching score of $cchunk_0$. Next sharpFix normalizes separately the searching scores of (1) the statement identified from $cchunks_0$ and (2) the candidates from $cchunks_1$, merges the candidates together, and ranks them by the normalized scores. This way, sharpFix obtains a list of candidates (as statements) retrieved from both the local faulty program and the code repository. For code reuse, sharpFix looks at each candidate in the list to possibly produce a plausible patch for the target which contains the

```
1 if (r!=null) {
2   result = r.getUpperBound();
3 }
```

Figure 5.8: A Candidate for the C4 Bug

suspicious statement s only.

For our example shown in Figure 5.5, given the suspicious statement (at Line 2) identified by fault localization, sharpFix performed code search and found a candidate (ranked as 105) that contains the single statement shown at Line 2 in Figure 5.8. sharpFix reused the if-condition from the statement's enclosing if-statement to produce the correct patch. We will explain the code reuse process in Section 5.4.3.

Evaluation

To evaluate sharpFix's code search, for the 103 bugs whose fix ingredients we identified in Section 5.2.2, we provided sharpFix with the faulty statements that we used for ssFix's code search experiments, and ran sharpFix's code search on each faulty statement to obtain a list of candidate statements. As we did in Section 5.3.1, we filtered away candidates that are syntactically redundant[13] and those that are simply from the bug-fixed versions, and we looked at the top-500 retrieved candidates. For each candidate (that contains a single statement), we performed sharpFix's translation to translate its enclosing method, and checked whether the exact fix ingredient is contained in the translated statement, its two neighbouring statements (because sharpFix may use the neighbouring statements to do insertion), and the enclosing if-condition if the enclosing statement is an if-statement (because sharpFix may use the enclosing if-condition to produce a new if-statement to replace the current statement).

Our results can be found in Figure 5.3. We can see that sharpFix's code search method is better than ssFix's. It retrieved promising candidates (which contain the exact fix ingredients) for 39 bugs within the top-50 results and for 42 bugs within the top-500 results. Compared to ssFix's code search, though sharpFix's code search retrieved only four more promising candidates within the top-500 results, it retrieved 39 candidates within the top-50 results which are more than all the candidates ssFix retrieved within the top-500 results. In Section 5.2.2, we found fix ingredients that exist in the code database in the parameterized forms for 80 bugs. sharpFix retrieved promising fix ingredients for 42/80=52.5%[14] bugs.

[13]Note that we looked at the candidate statement plus its two neighbouring statements to identify redundancy. sharpFix may a candidate statement's two neighbouring statements to do insertion for patch generation.

[14]This is a lower bound because not all the parameterized fix ingredients for the 80 bugs can be reasonably translated and reused for bug repair.

5.4.3 Code Reuse

We analyzed the failures in ssFix's code reuse steps: code translation, code matching, and modification (as shown in Figure 5.4), identified possible ways to improve each step, and developed sharpFix's method for each step. We next elaborate on sharpFix's code translation, code matching, and modification and show at last the experiments we conducted to evaluate sharpFix's code reuse and the results we got.

Code Translation

To do code translation, ssFix maps identifiers in the candidate to the related identifiers in the target, and then renames those candidate identifiers as the mapped target identifiers. We identified two weaknesses of the current method ssFix uses to identify related identifiers and create an identifier mapping: (1) ssFix only maps candidate identifiers to identifiers that are used in the target. A candidate identifier can be related to an identifier that is not used in the target but is accessible there; (2) ssFix identifies two identifiers to be related only based on matching the code patterns of their usage contexts. This can be not enough: Two identifiers can be highly related but their usage contexts are not identical (though very similar). Identical usage contexts may also not imply that the two identifiers being compared are the most related.

Algorithm 5 Creating an Identifier Mapping

Input: *tchunk*, *cchunk*
Output: *imap*[*id* → *id*] ▷ *id* is an identifier binding
1: *imap*[*id* → *id*]← empty
2: *cids* ← collect all the non-JDK candidate identifiers (those appear in *cchunk*'s method)
3: *tids* ← collect all the non-JDK target identifiers (those appear in *tchunk*'s method and appear as the declared fields and methods of *tchunk*'s class)
4: **for all** *cid* ∈ *cids* **do**
5: **if** !shortName(*cid*) **then** ▷ the string length is greater than 2
6: *tid* ← find the first *tid* ∈ *tids* whose name is equal to *cid*'s name
7: **if** *tid* exists and is compatible with *cid* **then**
8: *imap*.add(*cid*,*tid*)
9: *cmid* ← get *cchunk*'s enclosing method identifier
10: *tmid* ← get *tchunk*'s enclosing method identifier
11: *ccid* ← get *cchunk*'s enclosing class identifier
12: *tcid* ← get *tchunk*'s enclosing class identifier
13: **if** !*imap*.containsKey(*ccid*) && !*imap*.containsValue(*tcid*) **then**
14: *imap*.add(*ccid*,*tcid*)
15: **if** !isConstructorId(*cmid*) && !*imap*.containsKey(*cmid*) && !*imap*.containsValue(*tmid*) **then**
16: *imap*.add(*cmid*,*tmid*)
17: *imap* ← mapIdsByContexts(*cids*,*tids*,*imap*) ▷ using ssFix's method
18: *umcids* ← get all *cid*s that are currently unmapped
19: *umtids* ← get all *tid*s that are currently unmapped
20: **for all** *umcid* ∈ *umcids* **do**
21: *bestmatch* ← find the *umtid* in *umtids* that is compatible with *umcid* and share the most conceptual tokens with *umcid* (measured by Dice Similarity)
22: **if** *bestmatch* exists **then**
23: *imap*.add(*umcid*,*bestmatch*)
24: **return** *imap*

To address the two problems, we developed a new heuristic algorithm for mapping identifiers (shown in Algorithm 5). sharpFix uses the algorithm to create an identifier mapping and then renames each candidate identifier as its mapped target identifier to produce a translated candidate. To address (1), sharpFix looks for candidate identifiers that appear in the candidate's enclosing

method and it looks for target identifiers that not only appear in the target but (a) appear in the target's enclosing method, (b) appear as the declared fields of the target's class, and (c) appear as the declared methods of the target's class. sharpFix collects candidate identifiers that not only appear in the candidate itself but its enclosing method because it may use those identifiers to produce a patch. sharpFix collects target identifiers from (a), (b), and (c) to represent a set of accessible identifiers in the target to which a candidate identifier can be mapped. To address (2), sharpFix identifies related identifiers based on not just their usage contexts but their string lengths, string equality, locations, usage contexts, and shared concepts (measured by the overlap of the extracted conceptual tokens).

To create a name mapping, sharpFix accepts the target chunk *tchunk* and the candidate chunk *cchunk* as input. As output, sharpFix creates an identifier mapping *imap*. sharpFix first collects two lists of non-JDK identifiers *cids* and *tids* (Lines 2 & 3) which are actually identifier bindings (e.g., representing a variable declaration and its use). (Note that sharpFix does not collect JDK identifiers for translation.) *cids* are all the identifiers that appear in *cchunk*'s enclosing method and *tids* are all the identifiers that appear in *tchunk*'s enclosing method, appear as the declared fields of *tchunk*'s class, and appear as the declared methods of *tchunk*'s class. sharpFix visits the ASTs in pre-order to collect the identifiers (as did by ssFix). To collect *tids*, sharpFix first visits *tchunk*'s enclosing method, then the class fields, and then the class methods. Next sharpFix finds each *cid* in *cids* whose string is not short (i.e., the string length is greater than 2) and is equal to the string of some compatible *tid* from *tids* (two identifiers are compatible if they are both variables, types, or methods), and maps such *cid* to the found *tid* (Lines 4-8). Intuitively, if two identifiers share a long name (e.g., `getSize`), sharpFix treats them as related and maps one to the other. Next, sharpFix gets the identifiers of *cchunk*'s enclosing class and method. If they are not mapped, sharpFix maps them to the identifiers of *tchunk*'s enclosing class and method (Lines 9-16). (If *cchunk*'s enclosing method is a constructor, sharpFix does not map it to any method identifier.) Next, sharpFix maps the unmapped identifiers from *cchunk* to the unmapped ones from *tchunk* by usage contexts (Line 17). Here sharpFix uses ssFix's method and compares the code patterns of two identifiers' usage contexts to map identifiers. Finally, for each unmapped name *umcid* from *cids*, sharpFix finds the unmapped compatible *umtid* from *tids* that shares the most conceptual words (to extract such words, sharpFix uses ssFix's method described in Section 3.3.2), and then maps *umcid* to *umtid* (Lines 20-23). As an example, for the bug M69, sharpFix extracted two conceptual words *t* and *distribution* from the candidate identifier *tDistribution* and successfully mapped it to another identifier *distribution*. To measure the overlap of the extracted conceptual words, sharpFix uses the Dice Similarity.

The translated version yielded by sharpFix for the candidate shown in Figure 5.8 is just as itself (sharpFix did change the candidate's enclosing method's name and the name of a method call that do not appear in the candidate). For the variable identifier `result` in the candidate, sharpFix found an identifier in the target's method that has the same name, mapped `result` to this identifier, and renamed `result` as itself. For the variable `r` in the candidate, sharpFix mapped it to the identifer `r` in the target based on their matched usage contexts (e.g., as both `r!=null`). sharpFix did not map

`getUpperBound` to any target identifier and thus did not change it (note that sharpFix did not map `getUpperBound` to `getAnnotations` since they only share a stop word `get` which was not counted as a conceptual word).

Code Matching

ssFix's code matching method is based on matching rules and thresholds that are human-created. We found this makes ssFix's code matching not very flexible sometimes. As an example, for the bug M75, ssFix failed to match the two method calls: `getCumPct((Comparable<?>)v)` and `getPct(Long.valueOf(v))` simply because this is not supported by its matching rule: two method calls can only match if the method names being called are identical. Due to this code matching failure, ssFix failed to produce the correct patch by replacing `getCumPct` with `getPct`.

Algorithm 6 Code Matching Algorithm

```
Input: tchunk, rcchunk
Output: cmap[se → se]   ▷ se is a statement or a non-trivial expression (not an identifier, a number constant, or a literal)
 1: cmap[se → se]← empty
 2: tses ← collect all statements & non-trivial expressions from tchunk
 3: cses ← collect all statements & non-trivial expressions from rcchunk
 4: for all tse ∈ tses do
 5:    bestscore ← 0, bestcse ← null
 6:    tse_tks ← extract the LCS2 tokens from tse
 7:    for all cse ∈ cses do
 8:       if canMatch(tse,cse) then                          ▷ Check if tse and cse are compatible
 9:          cse_tks ← extract the LCS2 tokens from cse
10:          score ← DiceSimilarity(tse_tks,cse_tks)
11:          if score > bestscore then
12:             bestscore ← score
13:             bestcse ← cse
14:    if bestcse is not null then
15:       cmap.add(tse,bestcse)
16: return cmap
```

sharpFix uses a heuristic method (Algorithm 6) that we developed to do code matching. Compared to ssFix's code matching method, the new method uses simplified matching rules and uses no thresholds. It matches statements/expressions based on the extracted conceptual tokens and symbols, i.e., the LCS2 tokens shown in Section 5.4.2. To do code matching, sharpFix accepts the target *tchunk* and the translated candidate *rcchunk* as input. As output, it produces a code mapping *cmap* that maps each statement/expression in *tchunk* to its matched statement/expression in *rcchunk*. To create such a mapping, sharpFix starts by collecting two lists of statements and expressions *tses* and *cses* from *tchunk* and *rcchunk* respectively (Lines 2-3). It visits the ASTs in pre-order to collect the statements and expressions. The collected expressions are non-trivial and do not include identifiers, number constants, or literals (as *boolean*, *null*, *character*, or *string* literals). For each statement/expression *tse* in *tses*, sharpFix finds a *cse* in *cses* that is compatible with *tse* (determined by the method *canMatch* at Line 8) and shares the most LCS2 tokens with *tse* (measured by the Dice Similarity) and maps *tse* to *cse* (Lines 4-15). We defined the method *canMatch* to check whether two *se*s (statements/expressions) are compatible. Given two *se*s that are both statements, *canMatch* checks whether they are both loops. If so, it returns true. Otherwise, it only

returns true if their statement types[15] are equal (e.g., both as if-statements). Given two *se*s that are both expressions, *canMatch* returns true if and only if their expression types are equal. Given one *se* as a statement and the other as an expression, *canMatch* only returns true if the statement's type is *VariableDeclarationStatement* and the expression's type is one of *Assignment* and *VariableDeclarationExpression*. sharpFix extracts the LCS2 tokens from *tse* and *cse* and calculates a score based on the extracted tokens using the Dice Similarity.

For the bug example, sharpFix maps the target statement at Line 2 in Figure 5.5 to the matched (also translated) candidate statement at Line 2 in Figure 5.8. The extracted tokens and the similarity calculation are shown below.

```
LCS2 Tokens from the Target Statement (12 in total):
<collection>, <collect>, <c>, <=>, <r>, <.>,
<getannotations>, <get>, <annotations>, <(>, <)>, <;>

LCS2 Tokens from the Candidate Statement (11 in total):
<result>, <=>, <r>, <.>,
<getupperbound>, <get>, <upper>, <bound>, <(>, <)>, <;>

Overlapped Tokens (7 in total):
<=>, <r>, <.>, <get>, <(>, <)>, <;>

Dice Similarity: (2*7)/(12+11)=0.609
```

Modification

ssFix uses three types of modifications: replacement, insertion, and deletion to produce patches based on the matched and unmatched statements/expressions between the target and the translated candidate. sharpFix extends ssFix by using two more modifications: *adding if-guard* and *method replacement*. To produce patches using the modification *adding if-guard*, sharpFix looks at a target statement s (which appears in the target) and its mapped candidate statement s' (which appears in the translated candidate). If the parent of s' is an if-statement with a condition e', sharpFix creates new if-statements with the condition e' to guard existing statements in the target and in its enclosing method. To do so, sharpFix needs to decide what statements should be guarded. Currently, it selects two sets of statements: (1) the target statement s itself and (2) s plus the following statements its block, and it produces two patches: (1) it creates a new if-statement `if(e'){s}` and replaces s with the new statement and (2) it creates a new if-statement `if(e'){s,s0,...,sk}` where the statements from $s0$ to sk are all the statements that follow s in its block, it next replaces the statements from s to sk with the new if-statement. To produce patches using the modification *method replacement*, sharpFix replaces the enclosing method of the target with the enclosing method of the translated

[15]The type of a statement/expression is the node type of the statement/expression in the abstract syntax tree that sharpFix builds using the Eclipse JDT library [5].

candidate. It uses this modification to possibly support making multiple changes within a method for bug repair.

sharpFix uses the same method used by ssFix to do replacement (as described in Section 3.3.3). For insertion, sharpFix looks at the candidate statement s' to which the target statement s is mapped, identifies the adjacent statements of s' in its block: $s0'$ and $s1'$ that come before and after s', and inserts $s0'$ before s and $s1'$ after s to yield two patches. (Note that sharpFix looks at target and candidate code chunks each containing only one statement, so it does not use the same insertion method used by ssFix to do insertion.) sharpFix does not use ssFix's deletion method which was shown in Section 3.4.1 to be likely to produce defective patches.

For the bug example, sharpFix produces two patches using the modification *adding if-guard*. As one patch, sharpFix uses the if-condition `r!=null` to guard the target statement (Line 2 in Figure 5.5) only. The patched program fails to compile because variable `c` at Line 3 becomes undeclared (its declaration is now in the then-branch of the newly created if-statement). As another patch, sharpFix uses the if-condition to guard the target statement plus all its following statement in the block. This is actually the patch shown in Figure 5.5 and is correct (identical to the developer patch).

Evaluation

To evaluate sharpFix's code reuse ability, we looked at the 59 bugs for which sharpFix retrieved candidates that contain the fix ingredients in the parameterized forms. For each bug, we provided sharpFix with the target and the retrieved candidate and checked whether it can produce a plausible patch that is correct.

Experiment For each of the 59 bugs, we provided ssFix with the target and the retrieved candidate, and ran sharpFix's patch generation and patch validation automatically to possibly produce a plausible patch. If sharpFix produced such a patch, we manually checked whether the patch is correct (by comparing the patch to the developer patch provided by Defects4J).

Results Our results show that sharpFix produced 30 plausible patches which are all correct. sharpFix successfully reused 30/59=50.8% candidates for bug repair. This is actually a lower bound because not all the candidates can be reasonably reused for bug repair. We found that the exact fix ingredients (without any translation) are contained in 39 candidates, and we expect sharpFix to be able to reuse those fix ingredients in producing the correct patches. For the other 20 (59-39) candidates which only contain the fix ingredients in the parameterized forms, we manually determined whether they can be reasonably reused for bug repair. We identified only 3 candidates to be reasonable for reuse. We analyzed the failures of sharpFix in reusing the candidates for repairing the 12 (39+3-30) bugs. We found that the candidates are not ideal to be reused for repairing 9 bugs (we gave an example of such a candidate in Section 5.3.2). To successfully reuse the candidates to repair the other 3 bugs, we found that sharpFix's modification is still not sophisticated enough. For example, to repair the bug M101, we need to change the faulty if-condition `d` to `a||b||d`. For this

bug, we found a candidate that contains the if-condition `a||b||c` (after translation). Currently, sharpFix cannot apply its modification method to produce the correct patch in one step. It is however possible to use two steps by replacing replacing `d` with `a||b||c` and then replacing `c` with `d` (to successfully perform the second step, sharpFix needs to identify `c` and `d` are related). Though it is possible to make sharpFix's current modification method more sophisticated, doing so may not actually improve sharpFix's overall repair performance (as suggested by [133]).

5.4.4 Repair

We conducted the three experiments (E0, E1, and E2) used in Section 5.3.2 to evaluate sharpFix's repair abilities. The result of the full repair experiment E0 can be found in Table 5.1, sharpFix produced plausible patches for 89 bugs among which it produced correct patches for 36 bugs. (Recall that a patch is plausible if the patched program can pass the test suite and a patch is correct if it is semantically equivalent to the developer patch.) The median and average times for producing a plausible patch are about 11 and 25 minutes respectively. Compared to ssFix, sharpFix produced correct patches for 14 more bugs with the times (median and average) of producing a patch being comparable. To our knowledge, among all the existing APR techniques that were evaluated on the Defects4J dataset (by the time we ran the experiments), sharpFix produced correct patches (as the first-found plausible patches) for the largest number of bugs. The results of the experiments E1 and E2 can be found in Table 5.2 and in Table 5.3. For E1 and E2, with the faulty statement and the faulty method manually provided, sharpFix produced correct patches for (39-29)/29=34.5% and (43-36)/36=19.4% more bugs.

5.5 Summary

In this chapter, we revisited ssFix: the recent APR technique which finds and reuses existing code fragments to do bug repair. We conducted the experiment to test the fix-ingredient-assumption upon which ssFix was built. We found that the assumption is reasonable and that doing bug repair through reusing existing code is promising. We evaluated ssFix's code search, code reuse, and repair abilities, identified possible ways for improvement, and developed sharpFix as a new APR technique. We showed that compared to ssFix, sharpFix has better code search, code reuse, and repair abilities. For the full repair experiment, our results showed that sharpFix produced correct patches for 36 Defects4J bugs, and outperformed existing APR techniques in producing correct patches for the largest number of bugs.

Chapter 6

Conclusions & Directions for Future Research

As mentioned in Chapter 1, current search-based automated program repair (APR) techniques face two problems: (1) the search space problem and (2) the patch overfitting problem. For bug repair, a search-based APR technique uses pre-defined modification rules to create a search space of patches and works by searching within the space for finding a correct patch. The search space however is often very huge which makes finding a correct patch hard. Using a test suite, an APR technique can somehow produce a patched program that passes the test suite with the bug however not repaired in a valid way: the patch may not fully repair the bug or may repair the bug but introduce new bugs which the test suite does not expose.

In this thesis, we discussed three pieces of work (from Chapters 3 to 5) that we did to address the two problems for improving the effectiveness of automated program repair. The three pieces of work are the main contributions of the thesis.

In Chapter 3, we presented the APR technique ssFix which finds and reuses existing code fragments (from the local faulty program and the non-local programs in a large code repository) to do bug repair. By finding code fragments that are syntactically similar/related to the bug context and leveraging the small syntactic differences between the bug context and each similar/related code fragment to produce patches, ssFix essentially creates a search space that is reduced in size but can still contain the correct patches. Our experiments show that ssFix produced valid patches (as the first found plausible patches) for 20 of the 357 bugs with median time of producing a plausible patch being only about 11 minutes. ssFix was shown to outperform five other APR techniques in producing more correct patches.

In Chapter 4, we presented the patch testing technique DiffTGen which generates new test cases to expose any overfitting behaviors of a patched program. Given a faulty program and a patched program, DiffTGen calls a test generator to generate new test inputs to expose the differential behaviors between the two programs. Such test inputs are interesting because the exposed differential

behaviors are related to the patch (given the two programs are deterministic). DiffTGen asks an oracle to judge the correctness of the differential behaviors. If the behavior of the patched program is incorrect, the patch is overfitting. With an expected behavior possibly provided by the oracle, DiffTGen can produce an overfitting-indicative test case. Such a test case can be added to the original test suite for augmentation. Using the augmented test suite, an APR technique can avoid producing a similar overfitting patch again. We evaluated DiffTGen on 89 patches generated by four APR techniques for Java. We manually identified 10 of the 89 patches to be non-overfitting (and the other 79 patches are possibly overfitting). DiffTGen identified about 39 (49.4%) overfitting patches and generated the corresponding test cases exposing their overfitting behaviors. We further showed that the four APR techniques configured with DiffTGen produced less overfitting patches and more correct ones.

In Chapter 5, we conducted experiments to test ssFix's built-upon assumption and found that it is often possible to find the fix ingredients (the statements and expressions that can be used for producing the correct patches for bug repair) from the local faulty program and the non-local programs in a large code repository. We also conducted experiments to evaluate ssFix's code search, code reuse, and repair abilities. We identified possible ways to improve ssFix and developed sharpFix as an improved version of ssFix. sharpFix improves ssFix's code search by using different search methods for retrieving code fragments from the local faulty program and from the non-local programs in the code repository. For patch generation, sharpFix improves the ssFix's three steps: code translation, code matching, and modification. We evaluated sharpFix and found that compared to ssFix, sharpFix has better code search, code reuse, and repair abilities. sharpFix produced correct patches for 36 bugs in the Defects4J bug dataset. Compared to existing APR techniques that were evaluated on this dataset, sharpFix repaired more bugs with correct patches generated.

6.1 Directions for Future Research

Automated program repair is challenging. Current APR techniques are still far from maturity. They are only able to produce relatively simple patches. Their repair processes are often very expensive. And they are prone to producing overfitting patches. Due to these reasons, no current APR technique has been extensively applied in practice. We propose five research directions for making automated program repair more practical by improving its repairability (the ability of producing a patch for a bug in general), accuracy (the proneness of producing an non-overfitting patch rather than an overfitting patch), and efficiency.

- **Using more sophisticated code search for bug repair**: Our APR techniques ssFix and sharpFix reuse existing code fragments (from a large code database) that are syntactically similar to the context of a bug to do repair and have been shown to be promising. In fact, code fragments that are syntactically similar to the bug context are not necessarily semantics-related and may not be useful for bug repair. Using such code fragments, either ssFix or sharpFix may spend much of its repairing time generating patches that are not likely to be correct. It might be

worth investigating using semantics-based code search to improve ssFix's and sharpFix's code search performances and further to improve their repair performances. Standard semantics-based code matching and code search techniques like [107] are too expensive for retrieving code at the repository level. However, it is possible to combine using such standard techniques with ssFix's or sharpFix's syntactic code search to achieve better scalability. For example, it is possible to run ssFix's or sharpFix's code search to obtain an initial (large) set of code fragments that are syntactically similar to the bug context and then use standard semantic techniques to re-rank those code fragments to identify those that are the most semantics-related to the bug context for repair. It is also possible to apply approximated but scalable semantics-based code search techniques (e.g., based on PDG-based vectors [66], based on identifying similar API usage [30], based on finding similar code from Q&A sites for query expansion and further using expanded query code for finding more similar code [103]) to do a better job. A possibly better approach is to leverage machine learning techniques to train a model based on a set of pairs of buggy code and candidate code that is similar to it but contains the fix ingredients. Then we can further use the model to re-rank the candidates retrieved by code search.

- **Combining using multiple repair techniques**: Many automated repair techniques have been developed over the past decade. There has been increasing evidence [47, 190, 223] showing that different repair techniques are good at repairing different types of bugs. It is thus possible to combine using these techniques in a dynamic way to achieve better repair than using any particular technique alone. (For example, Genesis [131] uses specialized repair transformations mined from existing bug-fixes to produce patches and was shown to work reasonably well to repair three types of bugs caused by null pointer, out of bound, and class cast exceptions. If we can somehow tell that the program failure is caused by one of the three types of bugs, we may try using Genesis to produce patches first. If Genesis fails, then we may consider using another repair technique, e.g., ssFix, to do the repair.) It is also possible to learn (e.g., through machine learning) the "best" repair strategy (i.e., the best repair technique to be used) for a bug based on the bug context, the failure information, etc. and then apply the learned strategy to do effective repair.

- **Human intervention**: Achieving effective program repair in a fully automated way can be too hard. Doing semi-automated program repair by involving a human in the repair process of an automated repair technique can be a better option. A recent study [121] demonstrated the feasibility of involving a human in the fault localization process to improve the fault localization accuracy. It is also possible to involve a human in the patch testing process to identify an overfitting patch generated by the repair technique. Our patch testing technique DiffTGen [229] demonstrated the feasibility of doing so. DiffTGen works by generating new test inputs uncovering the semantic difference between a patched program and the original unpatched program, presents the differential semantics as different outputs to an oracle (a human), and asks for correctness judging. To be practical, DiffTGen needs to be improved

to be more human amenable: It needs to try finding and presenting outputs that are high-level and can be easily understood by a human (e.g., the return value of a public method as opposed to the value of a local variable from a private method), it needs to provide more testing information (related to the outputs) to a human to make correctness judging easier, etc.

- **Speeding up patch validation**: An automated repair technique that uses a test suite to validate patches spends a significant fraction of its repairing time on patch validation, i.e., on compiling and testing patches. Speeding up the patch validation process would make a repair technique work more efficiently. Current APR techniques do patch validation by first compiling the patched programs, next testing it against the test case(s) that the original program failed, and finally testing it against the whole test suite. It is possible to do some static analysis prior to patch compilation to avoid spending time on compiling an invalid patched program (which for example uses an undeclared variable). Testing patches is expensive, and it is possible to speed up this process. For example, after validating a patched program using the test case(s) the original program failed, instead of directly testing it against the whole test suite to determine whether the program introduced regressions, a repair technique can first identify relevant test cases that were affected by the patch (e.g., using the coverage information) and then test the program against these test cases. By doing so, a repair technique can save time testing a patch on irrelevant test cases. It is also possible to reason about the behavioral equivalence between two patches. If we can somehow identify two patches to be semantically equivalent, then we only need to validate one of the two patches (there are some initial research efforts [219, 151] that look at this direction).

- **Handling complex bugs**: Current techniques can only do simple bug repair: they often use a single repair action to produce a patch. The study by Zhong and Su [254] showed that the fix for more than 70% of the real bugs requires more than one repair action. Therefore, to be practical, an automated repair technique needs to support using multiple repair actions to repair a complex bug. GenProg [116, 74] implements using multiple repair actions through iterations. At each iteration, it applies single repair actions to produce a set of patches and copies the "promising" patches to the next iteration to produce more complex patches. A promising patch does not fully repair the bug but makes some progress and can be further modified to produce a correct patch. Currently, GenProg uses a simple method to identify a promising patch by computing a fitness score based on the number of test cases the patched program passed and failed. This has been shown to be not very effective. However, it is possible to consider using more information (e.g., the failure information, the context of the located bug, etc.) in addition to what GenProg does to make the identification of a promising patch more effective to be able to repair a complex bug in practice.

We believe it is possible to further improve the effectiveness of automated program repair by making progress in the research directions we proposed (and in other directions). In the future, we

hope to see an APR technique that can work fast and can generate high-quality patches for a large fraction of real bugs.

Bibliography

[1] ACS. https://github.com/Adobee/ACS.

[2] Apache Lucene. https://lucene.apache.org.

[3] BitBucket. https://bitbucket.org.

[4] Bugfest! Win2000 has 63,000 'defects'. http://www.zdnet.com/article/bugfest-win2000-has-63000-defects/.

[5] Eclipse JDT. https://www.eclipse.org/jdt.

[6] GitHub. http://github.com.

[7] GitHub Octoverse 2017. https://octoverse.github.com/#build.

[8] The GNU project debugger. https://www.gnu.org/software/gdb/.

[9] HDRepair. https://github.com/xuanbachle/bugfixes.

[10] Krugle. https://opensearch.krugle.org.

[11] Lucene practical scoring function. https://lucene.apache.org/core/4_6_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

[12] Open Hub. https://www.openhub.net.

[13] Program repair website. https://http://program-repair.org.

[14] searchcode. https://searchcode.com.

[15] SourceForge. https://sourceforge.net.

[16] SpoonLabs ASTOR. https://github.com/SpoonLabs/astor.

[17] SpoonLabs Nopol. https://github.com/SpoonLabs/nopol.

[18] Stack Overflow. https://stackoverflow.com.

[19] Valgrind. https://valgrind.org/.

[20] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan JC van Gemund. Exploiting count spectra for bayesian fault localization. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 12. ACM, 2010.

[21] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, pages 1780–1792, 2009.

[22] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[23] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[24] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE Computer Society, 2009.

[25] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering*, pages 289–299. IEEE Press, 2017.

[26] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM, 1990.

[27] Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th international conference on Software engineering*, pages 1003–1006. ACM, 2008.

[28] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168. IEEE, 2008.

[29] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, 2006.

[30] Sushil K Bajracharya, Joel Ossher, and Cristina V Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166. ACM, 2010.

[31] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[32] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pages 368–377. IEEE, 1998.

[33] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133. ACM, 2002.

[34] Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability*, pages 137–146. IEEE, 2007.

[35] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[36] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[37] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.

[38] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 209–224, 2008.

[39] Cristian Cadar and Hristina Palikareva. Shadow symbolic execution for better testing of evolving software. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 432–435. ACM, 2014.

[40] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381. ACM, 2012.

[41] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[42] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-Oriented Programming*, pages 609–633. Springer, 2011.

[43] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *International Conference on Formal Concept Analysis*, pages 273–288. Springer, 2008.

[44] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *International Conference on Software Engineering and Knowledge Engineering*, pages 238–243, 2011.

[45] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *33rd International Conference on Software Engineering (ICSE)*, pages 121–130. IEEE, 2011.

[46] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[47] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 637–647. IEEE Press, 2017.

[48] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, 2009.

[49] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220. ACM, 2002.

[50] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of 27th International Conference on Software Engineering*, pages 342–351. IEEE, 2005.

[51] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. NPEfix: Automatic runtime repair of null pointer exceptions in Java. *arXiv preprint arXiv:1512.07423*, 2015.

[52] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 99–104. ACM, 2005.

[53] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*, pages 383–401. Springer, 2016.

[54] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. NoFAQ: Synthesizing command repairs from examples. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2017.

[55] DARPA MUSE. The DARPA MUSE repository, 2016. `https://opencatalog.darpa.mil/MUSE.html`.

[56] Andrea De Lucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE, 2001.

[57] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 65–74. IEEE, 2010.

[58] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 78–95, 2003.

[59] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[60] Thomas Durieux, Benjamin Danglot, Zhongxing Zu, Matias Martinez, and Martin Monperrus. The Patches of the Nopol Automatic Repair System on the Bugs of Defects4J version 1.1.0. Technical Report hal-01480084, Université de Lille, 2017.

[61] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the Defects4J dataset. Technical Report 1505.07002, Arxiv, 2015.

[62] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *Proceedings of ACM/IEEE 30th International Conference on Software Engineering*, pages 855–858. IEEE, 2008.

[63] Robert B Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *Proceedings of the 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM, 2007.

[64] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

[65] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[66] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ACM/IEEE 30th International Conference on Software Engineering*, pages 321–330, 2008.

[67] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[68] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 786–791. IEEE, 2016.

[69] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 459–470. IEEE, 2015.

[70] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing q&a sites (t). In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318. IEEE, 2015.

[71] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005.

[72] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. Interactive fault localization leveraging simple user feedback. In *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 67–76. IEEE, 2012.

[73] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, pages 173–188, 2011.

[74] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *Proceedings of 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[75] Zhongxian Gu, Earl T Barr, David J Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 55–64, 2010.

[76] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *International Conference on Compiler Construction*, pages 80–95. Springer, 2006.

[77] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of Thirty-First AAAI Conference on Artificial Intelligence*, pages 1345–1351, 2017.

[78] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[79] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 267–280. Springer, 2004.

[80] Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125. IEEE, 2005.

[81] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245. ACM, 1990.

[82] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):345–387, 1989.

[83] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. An unabridged source code dataset for research in software reuse. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 339–342. IEEE, 2013.

[84] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 167–178. ACM, 2008.

[85] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[86] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE, 2007.

[87] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009.

[88] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400. ACM, 2011.

[89] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 137–146, 2010.

[90] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[91] Manu Jose and Rupak Majumdar. Bug-Assist: assisting fault localization in ANSI-C programs. In *International conference on computer aided verification*, pages 504–509. Springer, 2011.

[92] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–446. ACM, 2011.

[93] Xiaolin Ju, Shujuan Jiang, Xiang Chen, Xingya Wang, Yanmei Zhang, and Heling Cao. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, 90:3–17, 2014.

[94] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[95] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. MintHint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276. ACM, 2014.

[96] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[97] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[98] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Internet-scale real-time code clone search via multi-level indexing. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 23–27. IEEE, 2011.

[99] Iman Keivanloo, Juergen Rilling, and Philippe Charland. SeClone - a hybrid approach to internet-scale real-time code clone search. In *Proceedings of IEEE 19th International Conference on Program Comprehension (ICPC)*, pages 223–224. IEEE, 2011.

[100] Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 165–176. ACM, 2015.

[101] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE, 2013.

[102] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 301–310. IEEE, 2011.

[103] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY: A code-to-code search engine. In *Proceedings of International Conference on Software Engineering (to appear)*. IEEE, 2018.

[104] Sunghun Kim and E James Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[105] Andrew J Ko and Brad A Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[106] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, pages 301–310. ACM, 2008.

[107] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.

[108] Bogdan Korel and Ali M Al-Yami. Automated regression test generation. *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–152, 1998.

[109] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[110] Tien-Duy B Le, David Lo, and Ferdian Thung. Should I follow this fault localization tools output? *Empirical Software Engineering*, 20(5):1237–1274, 2015.

[111] Tien-Duy B Le, Ferdian Thung, and David Lo. Theory and practice, do they match? A case with spectrum-based fault localization. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 380–383. IEEE, 2013.

[112] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[113] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, 2016.

[114] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21(3):421–443, 2013.

[115] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.

[116] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE transactions on software engineering*, 38(1):54–72, 2012.

[117] Hua Jie Lee, Lee Naish, and Kotagiri Ramamohanarao. Effective software bug localization using spectral frequency weighting function. In *Proceedings of 2010 IEEE 34th Annual Computer Software and Applications Conference (COMPSAC)*, pages 218–227. IEEE, 2010.

[118] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Instant code clone search. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 167–176. ACM, 2010.

[119] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 212–221. ACM, 2014.

[120] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, 2007.

[121] Xiangyu Li, Marcelo dAmorim, and Alessandro Orso. Iterative user-driven fault localization. In *Haifa Verification Conference*, pages 82–98. Springer, 2016.

[122] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[123] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26. ACM, 2005.

[124] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[125] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.

[126] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[127] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 286–297. SIAM, 2005.

[128] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 282–291. IEEE, 2013.

[129] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 715–726. ACM, 2016.

[130] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. Technical Report 1706.09120, Arxiv, 2017.

[131] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739. ACM, 2017.

[132] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[133] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. ACM, 2016.

[134] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312. ACM, 2016.

[135] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Proceedings of 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 545–549. IEEE, 2015.

[136] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. Data-driven program completion. *arXiv preprint arXiv:1705.09042*, 2017.

[137] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective code search based on API understanding and extended boolean model (e). In *Proceedings of 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[138] James R Lyle. Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, 1987.

[139] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 249–260. ACM, 2017.

[140] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61. ACM, 2005.

[141] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[142] Lee Martie, Thomas D LaToza, and Andre van der Hoek. CodeExchange: Supporting reformulation of internet-scale code queries in context (t). In *Proceedings of 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35. IEEE, 2015.

[143] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[144] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[145] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[146] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM, 2014.

[147] Wolfgang Mayer and Markus Stumptner. Model-based debugging–state of the art and future challenges. *Electronic Notes in Theoretical Computer Science*, 174(4):61–82, 2007.

[148] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137. IEEE Computer Society, 2008.

[149] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2011.

[150] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM, 2011.

[151] Sergey Mechtaev, Gao Xiang, Shin Hwei Tan, and Abhik Roychoudhury. Partitioning patches into test-equivalence classes for scaling program repair. *arXiv preprint arXiv:1707.03139*, 2017.

[152] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE, 2015.

[153] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[154] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[155] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[156] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of 2014 IEEE Seventh International*

Conference on Software Testing, Verification and Validation (ICST), pages 153–162. IEEE, 2014.

[157] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[158] Syeda Nessa, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 548–559. Springer, 2008.

[159] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. GraPacc: a graph-based pattern-oriented, context-sensitive code completion tool. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1407–1410. IEEE Press, 2012.

[160] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[161] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 301–318. Springer, 2017.

[162] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

[163] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 902–912. IEEE, 2015.

[164] Gene Novark, Emery D Berger, and Benjamin G Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Acm sigplan notices*, volume 42, pages 1–11. ACM, 2007.

[165] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. Vejovis: suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847. ACM, 2014.

[166] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with JINSI. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 3–10. ACM, 2006.

[167] Alessandro Orso and Tao Xie. BERT: Behavioral regression testing. In *Proceedings of the 2008 international workshop on dynamic analysis*, pages 36–42. ACM, 2008.

[168] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proceedings of 2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 283–292. IEEE, 2011.

[169] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM, 1984.

[170] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, pages 75–84. IEEE, 2007.

[171] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1181–1192. ACM, 2016.

[172] Kai Pan, Sunghun Kim, and E James Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[173] Mike Papadakis and Yves Le Traon. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[174] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[175] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[176] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[177] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[178] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[179] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515. ACM, 2011.

[180] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[181] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P Reiss. Automated fault localization using potential invariants. *arXiv preprint cs/0310040*, 2003.

[182] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[183] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[184] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[185] Steven P. Reiss. Semantics-based code search. In *IEEE 31st International Conference on Software Engineering*, pages 243–253, 2009.

[186] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[187] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE, 2017.

[188] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[189] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Proceedings of 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[190] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.

[191] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 139–152. ACM, 2013.

[192] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287. IEEE Press, 2012.

[193] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[194] Ehud Y Shapiro. *Algorithmic program debugging*. MIT press, 1983.

[195] Stelios Sidiroglou and Angelos D Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 3(6):41–49, 2005.

[196] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 43–54. ACM, 2015.

[197] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26. ACM, 2013.

[198] Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, pages 1–33, 2017.

[199] Alexey Smirnov and Tzi-cker Chiueh. DIRA: Automatic detection, identification and repair of control-hijacking attacks. In *Proceedings of The Network and Distributed System Security Symposium*. Citeseer, 2005.

[200] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[201] Armando Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[202] Higor A Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[203] Soichi Sumi, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Toward improving graftability on automated program repair. In *Proceedings of 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–515. IEEE, 2015.

[204] Rishi Surendran, Raghavan Raman, Swarat Chaudhuri, John Mellor-Crummey, and Vivek Sarkar. Test-driven repair of data races in structured parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–25. ACM, 2014.

[205] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 471–482. IEEE, 2015.

[206] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[207] Kunal Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410. IEEE, 2008.

[208] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. eXpress: guided path exploration for efficient regression test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2011.

[209] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[210] Suresh Thummalapenta and Tao Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[211] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762. ACM, 2017.

[212] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .NET. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

[213] Frank Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica, 1994.

[214] Guda A Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119. ACM, 1991.

[215] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. Test input generation with Java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM, 2004.

[216] Weihang Wang, Yunhui Zheng, Peng Liu, Lei Xu, Xiangyu Zhang, and Patrick Eugster. Arrow: Automated repair of races on client-side web pages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 201–212. ACM, 2016.

[217] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[218] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[219] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: models and first results. In *Proceedings of 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[220] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

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

[222] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636. IEEE Press, 2017.

[223] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of Interational Conference on Software Engineering (to appear)*. IEEE, 2018.

[224] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[225] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012.

[226] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[227] W Eric Wong and Yu Qi. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[228] Xiaoyuan Xie, W Eric Wong, Tsong Yueh Chen, and Baowen Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[229] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[230] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE, 2017.

[231] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of International Conference on Software Engineering (to appear)*, 2018.

[232] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *ICSE*, pages 416–426, 2017.

[233] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[234] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[235] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 191–200. IEEE, 2014.

[236] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63. ACM, 2014.

[237] Jinqiu Yang and Lin Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.

[238] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[239] Wuu Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21:739–755, 1991.

[240] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell: fault localization using time spectra. In *Proceedings of the 30th international conference on Software engineering*, pages 81–90. ACM, 2008.

[241] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[242] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN*, 14(14):14, 2014.

[243] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 189–200. ACM, 2016.

[244] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. Technical Report 1703.00198, Arxiv, 2017.

[245] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267. Springer-Verlag, 1999.

[246] Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[247] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[248] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[249] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272. ACM, 2017.

[250] Sai Zhang and Congle Zhang. Software bug localization with markov logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 424–427. ACM, 2014.

[251] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pages 319–329. IEEE, 2003.

[252] Lei Zhao, Lina Wang, Zuoting Xiong, and Dongming Gao. Execution-aware fault localization based on the control flow analysis. In *International Conference on Information Computing and Applications*, pages 158–165. Springer, 2010.

[253] Lei Zhao, Lina Wang, and Xiaodan Yin. Context-aware fault localization via control flow analysis. *JSW*, 6(10):1977–1984, 2011.

[254] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 913–923. IEEE Press, 2015.

[255] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

[256] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24. IEEE Press, 2012.