Improving Patch Quality by Enhancing Key Components of Automatic Program Repair

Mauricio Soto

CMU-ISR-20-113
December 2020

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues (Chair)
Christian Kästner
William Klieber
Nicholas A. Kraft

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This research was sponsored by the National Science Foundation under grants CCF-1563797 and CCF1750116; the Air Force Research Laboratory (AFRL Contract No. FA8750-15-2-0075), and the US Department of Defense through the Systems Engineering Research Center (SERC), Contract H98230-08-D-0171.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Automatic Program Repair, APR, Patch Quality, Test Suites, Mutation Operators, Diversity

To the girls that make my life whole, my beautiful wife Diana, my daughter Luna, and my dog Naila. My love for them is infinite, like the search space of high-quality plausible patches.

Abstract

The error repair process in software systems is, historically, a resource-consuming task that relies heavily on manual developer effort. Automatic program repair approaches have enabled the repair of software with minimum human interaction mitigating the burden on developers, reducing the costs of manual debugging and increasing software quality.

However, a fundamental problem current automatic program repair approaches suffer is the possibility of generating low-quality patches that overfit to one program specification as described by the guiding test suite and not generalizing to the intended specification.

This dissertation rigorously explores this phenomenon on real-world Java programs and describes a set of mechanisms to enhance key components of the automatic program repair process to generate higher quality patches. These mechanisms include an analysis of test suite behavior and their key characteristics for automatic program repair (Chapter 4). In this Chapter we analyze the effectiveness of three well-known repair techniques: GenProg, PAR, and TrpAutoRepair, on defects made by the projects' developers during their regular development process, and modify and analyze the impact modifying characteristics such as size, coverage, provenance, and number of failing test cases has on the quality of the produced patches.

A second mechanism toward increase patch quality describes a set of research questions aimed at analyzing developer code changes to inform the mutation operator selection distribution (Chapter 5). In this Chapter we create a probabilistic model that describes how often human developers choose each of the different mutation operators available to automated repair techniques, and we later use this probabilistic model to create an APR approach informed by this distribution to generate higher quality patches.

Finally, the third mechanism describes a repair technique based on patch diversity as a means to explore n-version software and consolidated higher quality fixes (Chapter 6). In this Chapter, we explore a set of ways to increase diversity in patch generation and a set of approaches to create consolidated patches.

We propose a group of research questions aimed at further understanding how to increase patch quality in automated repair and we create a collection of approaches that increments the quality of generated patches. Most of the work discussed in this dissertation has been proposed, implemented, peer-reviewed, and published by the author of this thesis, his advisor, and professional colleagues in venues such as the journal IEEE Transactions on Software Engineering [148], the IEEE International Conference on Software Analysis, Evolution and Reengineering [192], the IEEE/ACM International Conference on Automated Software Engineering [193], and the International Conference on Mining Software Repositories [191, 194].

To answer our research question we created an experimental setup composed of three sections. We obtained a corpus of bugs to fix using Defects4J, a database and extensible framework of bugs which includes 357 bugs from five popular open-source projects. These projects expand into a diverse set of domains and also each bug

includes a comprehensive test suite created by the developers working in said project, and a human written patch which fixes each of the bugs described in the database.

We developed JaRFly, an open-source framework for implementing techniques for automatic search-based improvement of Java programs. Our study uses JaRFly to faithfully study mechanisms to increase patch quality in automated repair and also to include APR approaches to compare against. JaRFly reimplements GenProg and TrpAutoRepair to work on Java code, and makes the first public release of an implementation of PAR. Unlike prior work, our study carefully controls for confounding factors and produces a methodology, as well as a dataset of automatically-generated test suites, for objectively evaluating the quality of Java repair techniques on real-world defects.

The third component of our experimental setup includes a methodology for evaluating patch quality based on high-quality held-out test suites. We propose and implement a standard for using independently generated test suites to evaluate patch quality. This methodology can be used in future work to evaluate patch quality given its scalability and lack of human bias in the quality evaluation process. We also make publicly available our held-out test suites generated for this thesis and the source code of JaRFly for future extension and scrutiny.

Some of the main findings in this dissertation are:

- Using our open-source framework JaRFly we were able to generate 68 patches for the 357 analyzed defects.
- Fundamental test suite characteristics such as test suite coverage, size, provenance, and number of triggering test cases determine the quality of the resulting plausible patches generated by automated program repair.
- An automatic program repair technique informed in human-based mutation operator distribution increases the quality of the patches generated when compared to other APR techniques.
- We analyze how current APR approaches typically lack the diversity necessary to generate n-version software and consolidated patches. We propose a set of techniques to increase patch diversity and study the effects this diversity has in patch quality in automated repair.

Acknowledgments

I would like to especially thank my Ph.D. advisor **Claire Le Goues** for being a crucial person in my learning process and to help create the professional I am today. For her investment in my learning process and the great help not only for me to become a successful scientist but overall a better person. For all the good times together with the research group, like the trip to Sweden and the game nights with take out and board games; especially the one where we played Telestrations and Duy took an hour to draw a heart, and then two hours to draw a house.

Nick Kraft for being a great mentor, coworker, and friend, and for being there to drive me back that time I had too much Mint Julep.

Christian Kästner and **Will Klieber** for the mentoring and advice through this dissertation process, I couldn't have picked better committee members. I wish I would have had more time to learn other concepts from you, like how do QBF solvers work or how to juggle five clubs while riding a monocycle.

My professional colleagues and **co-authors** (Manish, René, Yuriy, Nick, Dave, Karen, etc.) for the great work in different publications and contributing to my learning process. Thanks to the work we've done together I will soon be able to say "Yes!" when there's an emergency in a plane and they ask if there's a doctor onboard.

My beautiful wife and daughter **Diana** and **Luna** who are my definition of happiness and make me feel like a dream every time I wake up. For them it might be more like a nightmare with all my snoring and bear-like movement but hey, love is love.

My dog **Naila** who is my everything and has been by my side every single day of this Ph.D. process, honestly I think it would be fair if my diploma is actually addressed to "Mau and Naila". CMU should give her at least partial credit; a Masters minimum.

My parents who guided me my whole life and somehow managed to survive my teenage years. I thought rejection was over then, but then I decided to start submitting research papers...

My **research group** (Zack, Rijnard, Afsoon, Deby, Chris, Cody, and Jeremy) for all the knowledge, feedback and learning process together, and for all the pics shared in #random. If Dr. Pepper could get his Ph.D., so can we! I should warn you guys though: most Marvel supervillains are Doctors[1] so watch those ambitions.

Gabriel Ferreira who taught me swear words in Portuguese and to drink caipirinha (a.k.a. lemon juice). I wouldn't have made it without his bad jokes and his feijoada.

I must say that finishing a Ph.D. while working full time, while raising a new born, while dealing with a world-wide pandemic has been... challenging, to say the least. My advise for future students: don't start a Ph.D. 6 years before a pandemic hits.

Also, maybe now it's a good time to come clean about that time I hacked the ICSE website and made a video about it[2]

Finally, if you think you were an important part of this process and I didn't mention you, please add yourself here: _____

[1] https://en.wikipedia.org/wiki/Doctor_(comics)

[2] https://www.youtube.com/watch?v=dQw4w9WgXcQ

Contents

1 Introduction **1**
- 1.1 Examples of Automatic Program Repair . . . 2
- 1.2 Generation of Low Quality Plausible Patches . . . 3
- 1.3 Thesis . . . 5
  - 1.3.1 Contributions . . . 7
  - 1.3.2 Potential Applications . . . 8

2 Background and Related Work **11**
- 2.1 Heuristic-Based Automatic Program Repair . . . 11
- 2.2 State of the Art Heuristic-Based Automatic Program Repair . . . 13
- 2.3 Constraint-Based Techniques . . . 15
- 2.4 Empirical Studies On Automatic Program Repair . . . 16
- 2.5 Defect Benchmarks . . . 18
- 2.6 Software Diversity . . . 19

3 Experimental Approach Overview **21**
- 3.1 Real-World Defects and Test Suites . . . 22
- 3.2 JaRFly: The Java Repair Framework . . . 23
  - 3.2.1 Problem Representation . . . 23
  - 3.2.2 Fitness Function . . . 24
  - 3.2.3 Mutation Operators . . . 25
  - 3.2.4 Search Strategy . . . 26
  - 3.2.5 Population Manipulation . . . 26
  - 3.2.6 Localization and Code Bank Management . . . 27
- 3.3 Quality Evaluation . . . 27
  - 3.3.1 Evaluating Patch Quality Through Held-out Test Suites . . . 28
  - 3.3.2 Analyzing Test Generation Tool Behavior . . . 29
  - 3.3.3 Creating High-Quality Held-Out Test Suites . . . 30

4 Analyzing the Role of Test Suites in the APR Process **33**
- 4.1 Ability to Produce Plausible Patches . . . 34
- 4.2 Analyzing Plausible Patch Quality . . . 37
  - 4.2.1 Patch Overfitting . . . 37
  - 4.2.2 Test Suite Coverage and Size . . . 40

4.2.3 Defect Severity . . . 45
4.2.4 Test Suite Provenance . . . 46

5 Analyzing Developer Software Changes to Inform APR Selection Mechanisms 53
5.1 Analysis of Developer Changes in Java Projects . . . 55
5.2 Corpus Mining from Popular GitHub Projects . . . 58
5.2.1 Categorizing Mutation Operators . . . 58
5.2.2 Building Probabilistic Model . . . 60
5.3 APR Tool Informed by Developer Edit Distribution . . . 62
5.4 Multi-Edit Rules to Inform Automatic Program Repair . . . 66

6 Incentivizing Patch Diversity in Automatic Program Repair 71
6.1 Quality of Hypothetical N-Version Patch vs. Individual Patches . . . 76
6.2 Improve Diversity of Generated Patches . . . 78
6.2.1 Modify Fitness Function and Implement Multi-Objective Search to Incentivize Patch Diversity . . . 80
6.2.2 Slicing Mutation Operators, Fault Locations, or Test Cases . . . 82
6.3 N-Version Patch Quality . . . 91

7 Conclusions and Future Work 95
7.1 Summary . . . 95
7.2 Limitations . . . 96
7.3 Threats to Validity . . . 97
7.4 Discussion and Future Work . . . 99

List of Figures

1.1 Illustrative example of buggy program 2
1.2 Example of a program patched by an automatic program repair technique 3
1.3 Example of a program patched by an automatic program repair technique 4
1.4 Example of low quality patch 5

2.1 Generate-and-validate process 12

3.1 Experimental approach 21
3.2 The 357 defect dataset 22
3.3 JaRFly 23
3.4 Evaluating the quality of generated plausible patches 28
3.5 Statement coverage of EvoSuite-generated test suites 31

4.1 Results of repair attempts 35
4.2 Distribution of patches generated 36
4.3 Quality of patches 37
4.4 Distribution of patch quality 38
4.5 Change in patch quality GenProg 39
4.6 Change in patch quality PAR 39
4.7 Change in patch quality TrpAutoRepair 40
4.8 Patch overfitting 41
4.9 Change in quality 41
4.10 Test suite coverage and size graphs. 43
4.11 Test suite coverage and size values. 43
4.12 Test suite coverage and size statistics. 44
4.13 Defect severity distribution 46
4.14 Defect severity linear regression 47
4.15 Patch results test suite provenance 49
4.16 Test suite provenance patche distribution 49
4.17 Test suite provenance patch statistics 50

5.1 Percentage of statements added, deleted and unmodified 57
5.2 APR Templates 59
5.3 Distribution of mutation operators 61
5.4 Two level probabilistic model 63

5.5 Patch example from probabilistic model 65
5.6 Association rule evaluation 69

6.1 Consolidating patches diagram 72
6.2 Patch diversity code snipet 73
6.3 Example of three overfitting plausible patches 74
6.4 Example of a high-quality consolidated patch 75
6.5 Patch quality individual vs. n-version patch 78
6.6 Graph individual vs. n-version patch 79
6.7 Semantically identical patches example 80
6.8 Semantic difference measurement 81
6.9 Patch quality combination vs individual patches 93

List of Tables

5.1 Likelihood of statement replacement 56
5.2 Patches from probabilistic model 64
5.3 Probabilistic model patches vs. other APR techniques 65
5.4 Association rule example 67
5.5 Association rules 100% confidence 68

6.1 Patches Standard APR Approaches 85
6.2 Slicing Mutation Operators Patches 85
6.3 Slicing Test Cases Patches 87
6.4 Slicing Test Cases Patches Per Target Analysis 87
6.5 Slicing Fault Location Patches 88
6.6 Slicing Fault Location Patches Detailed 89
6.7 Diversity-Driven APR Approach Patches 90

Chapter 1

Introduction

The cost of locating and repairing bugs in software systems has risen to $312 billion dollars annually and it is responsible for half of developer's time at work on average [73]. Given the pervasiveness of software, errors within these systems can compromise their security and privacy such as the Heartbleed bug. This bug allowed users to steal information protected by the SSL/TLS encryption used to secure the Internet [34].

Bugs may also be deadly such as the software powering the Therac-25 medical radiation therapy device where massive overdoses of radiation were administered to patients receiving up to 100 times the intended dose between 1985-87 [118]. These bugs can also cause millions of dollars to the economy, such as the Millenium bug (a.k.a. Y2K bug) which manifested when systems stored the last two digits per calendar year, therefore causing major repercussions in government, financial, and scientific software systems [105].

Repairing bugs like these is one of the most resource-intensive tasks in software development, requiring substantial manual effort [73, 199, 212]. Therefore, significant attempts have been dedicated in the last several years to create automatic program repair (APR) approaches, which are able to repair errors with minimum human interaction [99, 113, 139, 139, 153, 187, 192, 197, 210].

One well-known family of approaches known as *generate-and-validate* focuses on modifying the source code to *generate* patch candidates, which are later *validated* by the initial program specification (e.g. test suite) with the intent to produce a modified version of the source code that behaves accordingly to the behavior described in the provided program specification. We call this variant of source code a *plausible patch*.

A fundamental problem with automatic program repair approaches is their proneness towards generating low-quality patches that behave correctly in the specification provided, but not on a broader set of cases [122, 148, 190]. This is called "overfitting" to the initial specification.

The work described in this dissertation is of exceptional importance in this context given its portfolio of methods to increase patch quality of plausible patches generated by automatic program repair approaches. By applying the techniques proposed and analyzed in this thesis, plausible patches increase quality, therefore more often behaving correctly in a wider range of scenarios than the ones described in their limited program specification.

1.1 Examples of Automatic Program Repair

To further understand the APR generate-and-validate process we show an example of usage. In Figure 1.1 we show an example of a program with a bug in its source code. This function is meant to compute the maximum between two integers. In line 2 it assigns a default return value, which then gets modified in lines 4 and 6 to the expected value given the conditions in lines 3 and 5. Finally in line 7 the value is returned.

```
1 public int max(int x, int y){
2   int ret = 0;
3   if(x>y)
4       ret = x;
5   else if(x < y)
6       ret = y;
7   return ret;
8 }
```

Test	Return
assertEquals(**greater(1,2)**,2)	2
assertEquals(**greater(2,1)**,2)	2
assertEquals(**greater(1,1)**,1)	0

Figure 1.1: Illustrative example of a program with a bug. Below we show a test suite with three tests to validate correct functionality. The bug is manifested when running the third test case which calls the function for values ’1’ and ’1’. The expected value to be returned is ’1’, but the program returns ’0’ exposing buggy behavior.

At the bottom of Figure 1.1 we have three test cases to specify samples of correct expected behavior of the program. The first test case executes the program using inputs ‘1’ and ‘2’ to test cases where the first input is lesser than the second input, and it expects the return value to be the greater of the two (‘2’ in this example). The program behaves as expected returning the value ‘2’.

Similarly with the following test case where it executes the program with values ‘2’ and ‘1’ to test cases where the first input is greater than the second one. The program behaves as expected returning the value ‘2’ again. Finally, the third test case executes the program with inputs ‘1’ and ‘1’ to test cases where both inputs are the same. In this case the program fails to return the expected value of ‘1’, and erroneously returns the value ‘0’ manifesting its buggy behavior.

Generate-and-validate approaches have the ability to generate variations of source code following their mutation operators, which are different kinds of edits that can be applied to a program depending on the target portion of code desired to modify and the type of behavior needed to be modified in the source code.

In Figure 1.2 we can see a possible patch generated by automatic program repair. In this example a common mutation operator named “Append” is used, where its functionality is to add a line of code to a particular section of the program. In this example line number 4 was appended after line 2. This change in the source code modifies the behavior of the program by modifying the default return value. In the patched code, the default value is now the input “x’, and therefore

the program now behaves correctly in cases where the first input is greater, lesser, or equal to the second input. The mutation operator "Append" is a course-grain edit used for adding functionality to a section of code. This mutation operator is used by several APR approaches [96, 113, 210].

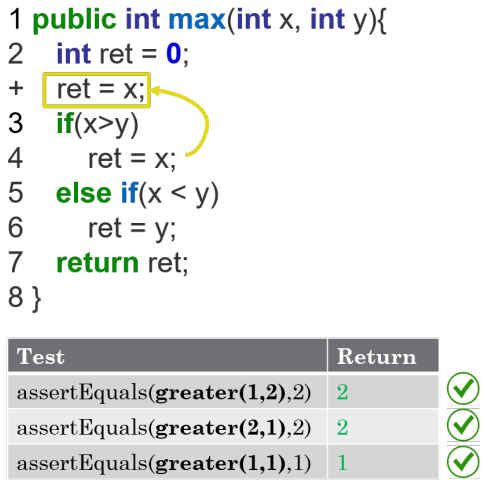


Figure 1.2: Example of a program patched by an automatic program repair technique. In this example the "Append" mutation operator is used to patch the source code by adding line 4 after line 2 generating a correct default value. All test cases pass after this fix.

In Figure 1.3 another possible patch is generated for the bug in Figure 1.1. In this example, a patch is generated by using the "Change Condition" mutation operator. As a result, the if condition in line 5 gets modified from using a lesser sign ("<") to a lesser equals sign ("<=") which then modifies the behavior of the program to include cases where both inputs contain the same value and therefore patching the buggy behavior of the program. Similarly fine-grain mutation operators as such are used by several APR approaches [99, 126, 129].

1.2 Generation of Low Quality Plausible Patches

There are several key challenges that generate-and-validate heuristic-based techniques must overcome to find patches [210]. First, reasoning about what is the correct potion of code that contains the error and thus must be modified is a whole field of study by itself. The set of potentially buggy program locations and the probability that any one of them is changed at a given time describes the *fault space* of a particular program repair problem.

Second, there are many ways to change potentially faulty code in an attempt to fix it. The source of the "fix code" that, when introduced into the program, produces a correct fix describes the *fix space* of a particular program repair problem. Given the premise that programs are often repetitive [20, 68] it is common for automated repair programs to build patches from portions of code within the same program.

Third, there are many ways to edit the code snippets identified by the *fix space* to patch the bug. We refer to these edits in this dissertation as *mutation operators* and they define the *repair strategy* of the APR technique. In this thesis we divide our mutation operator set in two groups:

```
1 public int max(int x, int y){
2   int ret = 0;
3   if(x>y)
4       ret = x;
5   else if(x <= y)
6       ret = y;
7   return ret;
8 }
```

Test	Return	
assertEquals(**greater(1,2)**,2)	2	✓
assertEquals(**greater(2,1)**,2)	2	✓
assertEquals(**greater(1,1)**,1)	1	✓

Figure 1.3: Example of a program patched by an automatic program repair technique. In this example the "Change Condition" mutation operator is used to patch the source code. The "<" symbol is modified to a "<=" symbol creating a correct path for the case where both parameters contain the same value. All test cases pass after this fix.

coarse-grain mutation operators: *append* candidate snippet, *replace* the buggy region with the candidate snippet, and *delete* the buggy region. These mutation operators can also be combined to build multi-edit patches [113]. *Fine-grain* mutation operators correspond to a set of *templates* constructed based on a manual inspection of a large set of developer edits to open source projects.

Finally, selecting the tests to be executed to evaluate a candidate patch defines a repair technique's *test strategy*. When an automated technique is able to generate a variant of source code by following this process that passes all the test cases in the guiding test suite, a plausible patch is generated. These plausible patches might actually be *correct patches* that fully fix the bug, or they might be *low-quality plausible patches* which do not meet the expected program behavior.

The possibility of creating low-quality plausible patches is a fundamental problem automatic program repair approaches face [122, 172, 190]. This phenomenon occurs when the approach finds a variant that satisfies the partial specification provided by the guiding test suite, but it fails to *generalize* to the intended program behavior. Guiding test suites are intrinsically incomplete, since they describe discrete samples in the behavior space, such that plausible patches can satisfy all provided tests but fail to satisfy an independent evaluation. This evaluation can take the form of an independent human evaluator or any kind of formal specification. Our implemented work aims to improve the generated patches' quality, making these approaches a more powerful tool usable in real-world systems.

Concretely, the generate-and-validate process is a versatile set of steps that several approaches have instantiated [99, 116, 169, 210], and some of its components can be enhanced to increase the probability of the plausible patches generated by these tools to be *correct patches*.

Following the examples presented in Section 1.1 APR approaches are able to create a patch for the bug in Figure 1.1 by modifying the source code as presented in Figure 1.4. In this example, a new line was added after line 2 to handle the case where the variable x contains the value '1'. This patch is described as a low quality patch given that it can satisfy the correct program description

presented by the test suite, but the patch does not *generalize* to further cases where both inputs contain the same value and that value is different from '1'.

```
1 public int max(int x, int y){
2    int ret = 0;
+    if(x == 1) ret = x;
3    if(x>y)
4       ret = x;
5    else if(x < y)
6       ret = y;
7    return ret;
8 }
```

Test	Return
assertEquals(**greater(1,2)**,2)	2
assertEquals(**greater(2,1)**,2)	2
assertEquals(**greater(1,1)**,1)	1

Figure 1.4: Example of a low quality patch created by an automatic program repair technique. The patch satisfies the test cases presented but does not generalize to further cases of the same bug (where both parameters contain the same value).

1.3 Thesis

The goal of the research presented in this thesis is to improve patch quality in automatic program repair techniques by enhancing key components of the *generate-and-validate* process. I have identified three segments in this APR methodology that can benefit from specialized improvements leading to higher quality patches, recognized as Phases 1, 3, and 5 in Figure 2.1. Previous studies [130,190,194] show the potential impact these components have in the quality of generated plausible patches and how enhancing the techniques used within these components can significantly improve patch quality. This thesis will not focus on the remaining two Sections described in Figure 2.1 given that these components have either been extensively analyzed in the past (e.g., fault localization techniques [2, 3, 91, 107, 164] in Phase 2), or they focus on improving patch finding speed, not quality (e.g., test prioritization [210] and reduction of test redundancy [103] in Phase 4). The three main topics this thesis will focus on are therefore summarized below:

- Guiding test suite:
  Heuristic-based generate-and-validate repair relies on a partial specification of the desired fixed program, commonly taking the form of a test suite (a set of test cases describing the expected program behavior; I will refer to this test suite as the "guiding" test suite). Our work analyzes fundamental test suite characteristics such as test suite coverage, size, provenance, and number of triggering test cases to determine how these characteristics impact the quality of the resulting plausible patches generated by APR. Different from previous studies, the experiments performed in this Section use a corpus of real-world open source popular projects.

- Mutation operator selection:
  State-of-the-art heuristic-based generate-and-validate APR techniques select between and instantiate various mutation operators to construct candidate patches, informed largely by heuristic probability distributions, which may reduce effectiveness in terms of both efficiency and output quality due to the inaccurate nature of heuristics. In practice, human developers use some edit operations far more often than others when fixing bugs manually. I, therefore, implemented an approach to guide the mutation operator selection mechanism in automatic program repair by analyzing and mimicking human developer behavior thus improving the quality of generated patches. To obtain the distribution of mutation operators selected by humans I analyzed the last 100 bug-fixing commits from the 500 most popular Java projects in GitHub and matched the changes performed in these commits to the analyzed APR mutation operators. Finally I added the mined distribution to an APR technique and compare the quality of the resulting plausible patches to the plausible patches generated by comparable APR techniques.
- Diversity-driven repair:
  Several generate-and-validate approaches rely on stochastic processes [116, 169, 210], therefore, it is common to obtain several plausible patches for a single defect. I implemented a repair approach that incentivizes diversity in the patch finding process to optimize for diversity and correctness to increase the diversity of patches found. A diverse pool of implementations reduces the probability of identical faulty behavior therefore providing higher fault tolerance [16]. This approach is composed of a group of diversity-driven techniques such as slicing the program specification and using multi-objective search to increase diversity.

Hence, the following thesis statement summarizes the principal claim of this research:

> **Thesis statement:** Automatic program repair approaches may create low-quality plausible patches that overfit to the guiding test suite. Improving key components of the automatic program repair process (specifically, test suite quality, mutation operator selection, and patch diversity) leads to an improvement in the quality of the produced patches.

This thesis is aimed at enhancing key components of automatic program repair with the goal of improving patch quality and reducing low quality plausible patches generated by APR approaches. In this dissertation we answer the following research questions:

RQ1 Do generate-and-validate techniques produce patches for real-world Java defects?

RQ2 How often and how much do the patches produced by generate-and-validate techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

RQ3 How do the coverage and size of the test suite used to produce the patch affect patch quality?

RQ4 How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

RQ5 How does the test suite provenance (whether written by developers or generated automatically) influence patch quality?

RQ6 Do real-world developers edit certain statement kinds more frequently than others in the bug-fixing process?

RQ7 What is the distribution of edit operations applied by human developers when repairing errors in real world projects?

RQ8 How does a human-informed automatic program repair tool compare to the state-of-the-art in APR?

RQ9 What are the most common multi-edit modification rules in practice?

RQ10 Can overfitting be mitigated by exploiting randomness in the repair process? Do different patches overfit in different ways?

RQ11 How does software diversity impact patch quality in APR?

RQ12 How does patch diversity relate to patch quality when plausible patches are consolidated in relation to their non-consolidated counterparts?

1.3.1 Contributions

The work described in this dissertation explores in-depth a set of techniques to increase patch quality in the APR process of Generate-and-Validate approaches building on prior studies on smaller programs and other target languages [31, 112, 189]. We create JaRFly, a framework for Java generate-and-validate program repair techniques where we create Java versions of GenProg [113] and TrpAutoRepair [169] and reimplement PAR [99]. JaRFly is open-source and available at `https://github.com/squaresLab/genprog4java/`. We further use state-of-the-art automated test generation to generate high-quality test suites for real-world defects in Defects4J used in our study, and create a methodology for generating more such test suites for other defects. Our data, test suites, and scripts are all available at `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

Overall, our work has identified techniques to improve patch quality in automated repair when applied to real-world defects, and will drive research toward improving the quality of program repair. The major contributions of this dissertation are detailed below:

- **Patch quality analysis from test suite characteristics:** An analysis of the role test suites play in the context of automatic program repair. We analyze fundamental characteristics of test suites and the impact these characteristics have in the obtained patches' quality measures.
- **Empirical evaluation of APR approaches in real-world defects:** We evaluate three APR approaches in a large set of real-world defects from open-source projects. This outlines shortcomings and establishes a methodology and dataset for evaluating quality of new repair techniques' patches and promote research on high-quality repair.
- **Dataset of independent evaluation test suites for Defects4J defects:** We created an extensive set of test suites independent of the developer-generated guiding test suite used to evaluate the quality of repairs. Similarly, we outline a methodology for generating such test suites. Augmenting existing Defects4J defects with two, independently created test suites can aid not only program repair, but other test-based technology.

- **Mutation operator analysis of bug-fixing commits by human developers:** This thesis provides a deeper understanding of human developer edits when fixing errors in source code, and how do the analyzed APR mutation operators relate to the human edits.
- **Creation of developer-informed repair approach:** We conducted a mining study and constructed an empirical model of single-edit repairs from a substantial corpus of open-source projects. We later used this knowledge to inform an APR approach favoring mutation operators which human developers more commonly use and analyze the quality of the repairs created by said approach.
- **Study of software diversity in the context of APR:** We propose, implement and analyze a set of techniques to improve diversity in the context of APR. These techniques include slicing different components of the program specification (fault locations, guiding test cases, and mutation operators) in the repair process; and a multi-objective measurements based on program specifications (e.g. test suites) to approximate program diversity. This is beneficial, both to find more diverse patches thus avoiding similar faulty behavior, and to exploit these diverse patches to consolidate solutions into a higher quality fix.
- **Creation of multi-objective repair approach:** The implementation and analysis of a multi-objective repair approach which can optimize for several goals in the repair process. In this thesis, this approach was used in search space traversal to optimize for correctness and diversity, therefore finding a more diverse pool of patches.
- **Java Repair Framework (JaRFly):** A publicly released, open-source framework for building Java generate-and-validate repair techniques, including our reimplementations of GenProg, PAR, and TrpAutoRepair. JaRFly is designed to allow for easy combinations and modifications to existing techniques, and to simplify experimental design for automated program repair on Java programs. All the code and data produced in this thesis to run our experiments is made publicly available to support reproducibility and extension[1].

1.3.2 Potential Applications

The research implemented in this dissertation has applications that extend to real-world industrial software systems using APR techniques. The popularity of APR in the research environment continues to grow, and similarly, applications in industrial environments using APR have started to emerge [134]. The majority of patches generated by current approaches [99, 116, 169, 192, 220] are still low-quality patches therefore making this further research in this direction essential for further adoption of APR tools in industry.

This thesis proposes a set of techniques to increase the quality of plausible patches found in the APR process, therefore diminishing the gap existing between state-of-the-art APR approaches and their broader adoption in real-world applications. Potential applications of the research outlined in this thesis include error repair in all kinds of software, error detection, human-assisted debugging and error fixing, dynamic error repair, etc.

The rest of this document focuses on a series of techniques designed to improve patch quality in APR, it proceeds as follows: Chapter 3 describes a common methodology taken in the different

[1] https://github.com/squaresLab/genprog4java

experiments and studies that comprise this thesis. Chapter 4 analyzes the role of test suites in automatic program repair and how modifying test suite quality characteristics leads to higher quality patches; Chapter 5 depicts a study of developer behavior to inform the distribution of program edits and statement kinds as selected by APR approaches, and Chapter 6 explains the benefits of increasing diversity in the automatic program repair process as a means to increase patch quality.

Chapter 2

Background and Related Work

Automatic program repair techniques can be classified broadly into three classes [74]: (1) *Heuristic-based* techniques create candidate patches (often via search-based software engineering [78]) and then validate them, typically through testing (e.g., [9, 43, 51, 53, 96, 99, 123, 139, 153, 172, 187, 197, 210, 211]. Overall, *generate-and-validate* techniques refers to the process of creating variations of source code and validating their correctness. (2) *Constraint-based* techniques use constraints to build patches via formal verification, inferred or programmer-provided contracts, or specifications (e.g., [89, 165, 209]). (3) *Learning-based* techniques where fixes (code transformations, ranking models, buggy code models, etc.) are learned from a large corpus of patches commonly using deep learning (e.g., [19, 30, 77, 119, 125, 130, 204]). This paper focuses on heuristic-based techniques.

Test-driven heuristic-based techniques (a subset of generate-and-validate) are a particularly interesting subject of exploration, as they have been shown to repair defects in large, real-world software [122, 148] (e.g., Clearview [166], GenProg [113], PAR [99], and Debroy and Wong [51]). Constraint-based techniques (also known as synthesis-based) show great promise for new or safety-critical systems written in suitable languages, and adequately enriched with specifications. However, the significance of defects in existing software demands that research attention be paid at least in part to techniques that address software quality in existing systems written in legacy languages. Since legacy codebases are often idiosyncratic to the point of not adhering to the specifications of their host language [25], it might not be possible even to add contracts to such projects.

2.1 Heuristic-Based Automatic Program Repair

The generate-and-validate process is described in Figure 2.1. This repair approach takes as input a program with one or more bugs and a specification of correct behavior of the program, which typically takes the form of a test suite with passing and failing test cases (Phase 1 in Figure 2.1). The passing test cases describe correct program functionality that should be maintained, while failing test cases specify incorrect program behavior. All test cases in the test suite are assumed to be correct.

Generate-and-validate approaches then identify the locations of the program with higher probability of being buggy (Phase 2) by applying mechanisms from fault localization literature

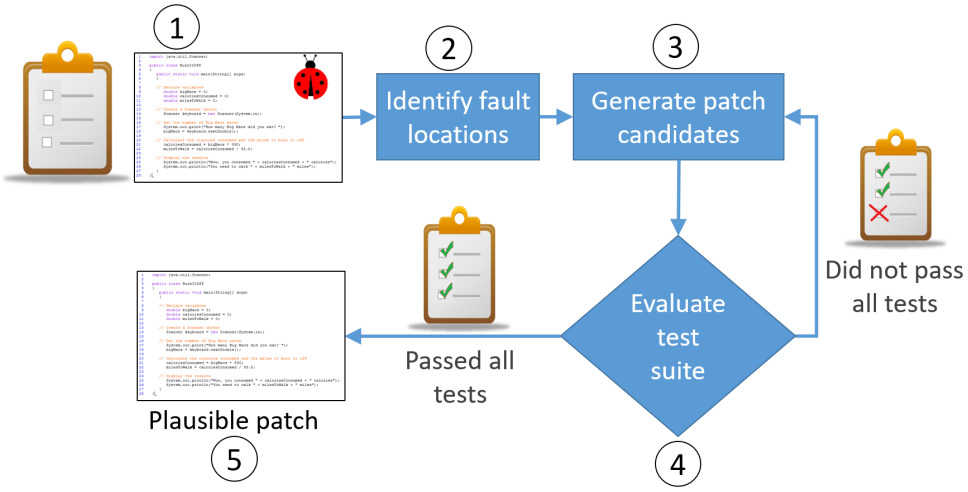


Figure 2.1: The automatic program repair *generate-and-validate* family to produce plausible patches starting from source code containing a bug and a test suite that describes the desired program behavior with passing and failing test cases.

(e.g., spectrum-based fault localization techniques such as Tarantula [91]). The information gained from the analysis of all test case traces is used to identify possible buggy locations. These approaches then *generate* variants of the original source code (Phase 3), usually referred to as *patch candidates*. APR techniques within the heuristic-based family use heuristics to guide the traversal of the search space to generate patch candidates. These heuristics commonly include stochastic components and the candidates created depend on their available *mutation operators* and search strategy. There is a broad diversity of such operators used in automatic program repair, including deleting or inserting statements [116, 169], applying templates [99], transformation schemas [127, 130], or semantically-inferred code [139, 140, 219].

Finally, the approaches *validate* the patch candidates' compliance to the correct program specification. In practice, the most common form of validation is testing. set of test cases (Phase 4). If a variant is found that satisfies the behavior described by the test suite, this variant is considered a *plausible patch* (Step 5), where "plausible" indicates that the variant passes all test cases. These patches have therefore a higher probability of being a *correct patch* for the bug [172], making the program follow the program's expected behavior. These approaches have been successful in patching bugs for real-world software systems [19, 99, 116, 134, 192], as well as simpler software created for educational purposes [190]. Researchers have proposed several instances of this family of approaches as successful exponents of APR achievements [35, 89, 99, 113, 139, 153, 166, 210].

The concept of *plausible* patch is directly related to the core theme of this dissertation which is *patch quality*. We call the patches generated by APR approaches *plausible* patches because the way they are generated ensures that they behave correctly *in the specification provided* (which we refer to as the "guiding test suite" in this dissertation. It is worth clarifying that there exist other ways to provide program specification and it does not necessarily need to be a test suite). The guarantee of *correct* behavior is therefore limited only to the behavior described in the guiding test suite and there is no guarantee that these plausible patches will behave correctly in cases

outside of the cases described in the guiding test suite.

Thus, to evaluate if a patch generated by APR is "better" than another, we require a measurement to evaluate patch quality that is independent from the guiding test suite and that measures if the generated plausible patch behaves as expected in a large spectrum of possible intended behavior, not just in the behavior described in the guiding test suite.

We perform this quality evaluation by using held-out test suites as described in Section 3.3 where the main idea is to generate a specification of the program behavior which is independent of the guiding test suite and evaluates how much of that independent specification is the patch able to generalize to. In this dissertation we refer to this independent specification (which also does not necessarily need to be a test suite) as a "held-out test suite" .

The process of a patch behaving correctly in the cases described in the guiding test suite but not in further cases is called "overfitting". This has a negative connotation and implies low-quality patches. The opposite behavior, when a patch behaves correctly in the cases described by the guiding test suite *and* also in further cases, is commonly referred as the patch "generalizing" to a broader spectrum of cases. This has a positive connotation and relates to high-quality patches. Automatic program repair approaches are prone to generating low-quality patches that *overfit* to the guiding test suite [122, 148, 190] therefore the work described in this thesis becomes of exceptional importance given its portfolio of methods to increase patch quality in automatic program repair. Thus, the main purpose of this thesis is to find ways to improve the quality of plausible patches, meaning therefore that they will behave correctly in more cases than the ones described in the guiding test suite and therefore generalize to more cases in its expected behavior.

2.2 State of the Art Heuristic-Based Automatic Program Repair

There are previous program repair techniques that, similar to the work described in this thesis, target the Java programming language. Search-based approaches can be categorized within two major families: template-based or statement-edit [192] to which we directly compare our work against. PAR [99], for example, searches for common patterns used by developers to generate fix templates. These templates are single-edit modifications of the source code that are used often to fix bugs. SOFix [124] also uses predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions and answers for fine-grained modifications. ARJA [224] is a Java-focused repair technique that implements multi-objective search and uses NSGA-II to look for simpler repairs. Genesis [125] is a repair approach that processes human patches to automatically infer code transforms for automatic patch generation. The authors use patches and defects collected from 372 Java projects. QACrashFix [71] is a repair approach that extracts fix edit scripts from StackOverflow and attempts to repair programs based on them. Part of this dissertation uses certain functionality from QACrashFix to account for replacements in the human behavior study (Section 5.2).

Previous tools have also tried to modify the objective function of APR approaches following different methodologies from ours. HDRepair [111] modifies the fitness function based on fix

history to assess patch suitability. The fitness of patch candidates is determined by how closely the changes in a patch occur in the analyzed corpus using a graph-based representation of the patches. Prophet [130] uses a probabilistic model built on the history of 8 different projects to rank candidate patches. It learns model parameters via maximum likelihood estimation. Unlike this work, we apply mined knowledge when actually instantiating patch candidates rather to rank the already created patch candidates, which reduces the search space at creation time.

GenProg [116], PAR [99], and TrpAutoRepair [169] are examples among a family of syntactic-based automatic program repair approaches seeking to generate patch candidates by modifying program syntax (while semantic-based approaches use code synthesis to construct fix code). These repair approaches are widely used in this dissertation and represent a variety of search-based techniques that vary in mutation operator kind and search traversal, therefore we directly compare against them in this thesis. GenProg [116] is an APR approach that leverages genetic programming while modifying software syntax using coarse-grained mutation operators such as delete, append, and replace. TrpAutoRepair [169, 171] traverses the search space using random search and restricts its pool of possible patches by applying a single edits to its candidate patches. The authors of this approach evaluate their tool against GenProg and suggest that TrpAutoRepair outperforms GenProg in a 24-bug benchmark. PAR [99] uses a set of templates mined from human behavior to modify source code (e.g., check if a variable is null before using it). Test suite behavior in the context of automatic program repair has been studied in the C language with a corpus of programs written by novices [190].

There are state-of-the-art repair techniques that extend the approaches we compare against or are contained within the same families we compare to. Similar to PAR, LASE [144] learns edit scripts from a pool of bug-fixing examples, finds the appropriate edit locations and applies the customized edit to the selected location. The scripts consist of a sequence of operations (insert, delete, update, and move) applied to the nodes of an abstract syntax tree representation of the program. These scripts are learned from examples changed in syntactically similar ways. SPR [127] combines staged program repair and condition synthesis to find repairs in programs. This repair approach introduces a set of parameterized transformation schemas to generate and search a diverse space of program repairs. Their evaluation in 69 bugs from 8 open source programs indicate an improvement over previous approaches. DLFix, similarly, creates code transformations based on a technique which applies deep learning applied to previous bug fixes [119].

Several state-of-the-art techniques also resemble or extend GenProg, for example, AE [210] uses adaptive search to improve the order in which test cases and candidate patches are evaluated to improve the usage of resources in the APR process. Because of this prioritization technique, AE reduces the search space size by an order of magnitude as compared to GenProg. JAFF [12, 14] is a repair technique based on co-evolution where programs and test cases co-evolve together. These components influence each other with the aim of fixing errors in programs automatically. Kali [172] is a tool that focuses in the removal of source code statements as a means to pass all test cases from a test suite.

There exist also state-of-the-art techniques that target domain-specific defects are therefore less directly comparable to our approaches. LeakFix [70] is a safe memory-leak fixing tool for C programs. It uses pointer analysis to build procedure summaries. Based on these, it generates fixes by freeing memory in key program points. Schlute et al. [181] developed an automatic program repair system for arbitrary software defects in embedded systems. It targets mostly systems with

limited memory, disk and CPU capacities. It does not require the program source code.

Similarly, ErrDoc [202] uses insights obtained from a comprehensive study of error handling bugs in real-world C programs to automatically detect, diagnose, and repair the potential error handling bugs in C programs. JAID [39] uses automatically derived state abstractions from regular Java code without requiring any special annotations and employs them, similar to the contract-based techniques to generate candidate repairs for Java programs. DeepFix [77] and ELIXIR [178] use learned models to predict erroneous program locations along with patches. ssFix [215] uses existing code that is syntactically related to the context of a bug to produce patches. CapGen [213] works at the AST node level and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. SapFix [134] and Getafix [19], two tools deployed on production code at Facebook, efficiently produce correct repairs for large real-world programs. SapFix [134] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in realtime. Getafix [19] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook's in-house static analysis tool. SimFix [87] considers the variable name and method name similarity, as well as structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. SketchFix [84] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution.

In addition to repair, search-based software engineering has been used for developing test suites [145, 207], finding safety violations [8], refactoring [183], and project management and effort estimation [21]. Good fitness functions are critical to search-based software engineering. Our findings indicate that using test cases alone as the fitness function leads to patches that may not generalize to the program requirements, and more sophisticated fitness functions may be required for search-based program repair.

2.3 Constraint-Based Techniques

Different from the work performed in this dissertation, there exists another important family of approaches referred to as constraint-based repair which is a family of techniques that uses constraints to build patches that satisfy an inferred specification. These constraints may take the form of developer-generated specifications, formal verification, invariants, among others [89, 165]. Such techniques typically use synthesis to construct repairs, using a different mechanism than our approach for both constructing and traversing the search space, therefore our approach is less immediately comparable. Some examples of this family of approaches are SemFix [153], DirectFix [139], QLOSE [47], Angelix [140], S3 [110], ACS [217], and Nopol [220] which use SMT or SAT constraints to encode test-based specifications.

SimFix [87] mines code patterns from frequently occurring code changes from developer-written patches. Then, in the project of the defect, SimFix identifies code snippets that are similar to the code SimFix has localized the defect to. SimFix ranks the code snippets by the number of times the mined patterns have to be applied to the snippet to replace the buggy code and then

selects the snippets (one at a time) from the ranked list.

SemFix [153] generates repair constraints using symbolic execution and the guiding test suite, it then solves the constraints using an SMT solver. DirectFix [139] uses partial maximum SMT constraint solving and component-based program synthesis building simpler and safer patches than SemFix. Angelix [140] focuses on a repair constraint to reduce the search space named "angelic forest" independent of program size, which represents a considerable improvement in scalability over its predecessors [139,153]. Recently a Java version was proposed called JFix [109]. QLOSE [47] is an approach which finds plausible patches by minimizing an objective function based on semantic and syntactic distances from the buggy version.

S3 [110] focuses on a programming-by-examples methodology which uses code synthesis to find plausible patches. ACS [217] targets `if conditions`, using dependency-based ordering and predicate mining. Nopol [220] is a Java-focused approach which targets `if conditions`. It uses an SMT solver and angelix fix localization to create plausible patches for the buggy programs. Part of the work performed in this dissertation was evaluated against Nopol. The original publication [192] describes a full description of the comparison. It is worth mentioning that semantic-based techniques do not pick mutation operators based on heuristics, therefore the work performed in the publication is not directly applicable to that family of techniques.

SemGraft [138] infers specifications by symbolically analyzing a correct reference implementation instead of using test cases. Genesis [125], Refazer [176], NoFAQ [48], Sarfgen [208], and Clara [76] process correct patches to automatically infer code transformations to generate patches. SearchRepair [97] blurs the line between generate-and-validate and semantic-based techniques by using constraint-based encoding of the desired behavior to replace suspicious code with semantically-similar human-written code from elsewhere.

Some automated repair techniques focus on a particular defect class, such as buffer overruns [186, 188], unsafe integer use in C programs [43], single-variable atomicity violations [89], deadlock and livelock defects [121], concurrency errors [123], and data input errors [9] while other techniques tackle generic bugs. Our evaluation has focused on tools that fix generic bugs, but our methodology can be applied to focused repair as well.

2.4 Empirical Studies On Automatic Program Repair

Prior work has argued the importance of evaluating the types of defects automated repair techniques can repair [150], and evaluating the generated patches for understandability, correctness, and completeness [146]. Yet many of the prior evaluations of repair techniques have focused on what fraction of a set of defects the technique can produce patches for (e.g., [35, 46, 56, 89, 117, 135, 210, 211]), how quickly they produce patches (e.g., [113, 210]), how maintainable the patches are (e.g., [67]), and how likely developers are to accept them (e.g., [1, 99]).

However, some recent studies have focused on evaluating the quality of repair and developing approaches to mitigate patch overfitting. For example, on 204 Eiffel defects, manual patch inspection showed that AutoFix produced high-quality patches for 51 (25%) of the defects, which corresponded to 59% of the patches it produced [165]. While AutoFix uses contracts to specify desired behavior, by contrast, the patch quality produced by techniques that use tests has been found to be much lower. Manual inspection of the patches produced by GenProg, TrpAutoRepair

(referred to as RSRepair in that paper), and AE on a 105-defect subset of ManyBugs [172], and by GenProg, Nopol, and Kali on a 224-defect subset of Defects4J showed that patch quality is often lacking in automatically produced patches [135]. An automated evaluation approach that uses a second, independent test suite not used to produce the patch to evaluate the quality of the patch similarly showed that GenProg, TrpAutoRepair, and AE all produce patches that overfit to the supplied specification and fail to generalize to the intended specification [31, 189]. This work has led to new techniques that improve the quality of the patches [97, 126, 129, 215, 216, 223].

For example, DiffTGen generates tests that exercise behavior differences between the defective version and a candidate patch, and uses a human oracle to rule out incorrect patches. This approach can filter out 49.4% of the overfitting patches [215]. Using heuristics to approximate oracles can generate more tests to filter out 56.3% of the overfitting patches [216]. UnsatGuided uses held-out tests to filter out overfitting patches for synthesis-based repair, and is effective for patches that introduce regressions but not for patches that only partially fix defects [223]. Automated test generation techniques that generate test inputs along with oracles [27, 72, 149, 196] or use behavioral domain constraints [10, 69, 90, 201], data constraints [61, 151, 152], or temporal constraints [22, 23, 24, 57, 155] as oracles could potentially address the limitations of the above-described approaches.

Using independent test suites to measure patch quality is a technique used throughout this thesis and it has also been applied to previous studies. Even though this measurement is imperfect, as test suites are a partial specification and may identify some incorrect patches as correct, it provides with an scalable and less biased way to measure patch quality. On a dataset of 189 patches produced by 8 repair techniques applied to 13 real-world Java projects, independent tests identify fewer than one fifth of the incorrect patches, underestimating the overfitting problem [108]. However, on other benchmarks, the results are much more positive. For example, on the QuixBugs benchmark, combining test-based and manual-inspection-based quality evaluation could identify 33 overfitting patches, while test-based evaluation alone identified 29 of the 33 (87.9%) [221].

While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [163]. Further, using independently generated test suites instead of using the subset of the original test suite to evaluate patch quality ensures that we do not ignore regressions a patch is most likely to introduce. Poor-quality test suites result in patches that overfit to those suites [148, 172].

Studying the improvement of patch quality of the patches generated by Angelix on the IntroClass [115] and Codeflaws [198] benchmarks of defects in small programs finds results consistent with the work presented in this dissertation. By contrast, this thesis focuses on real-world defects in real-world projects and generate-and-validate repair. Further, prior work has shown that the selection of test subjects (defects) can introduce evaluation bias [26, 168]. The evaluation technique presented in this thesis focuses on the limits and potential of patch quality improvement on repair techniques when evaluated in a large dataset of defects, and controls for a variety of potential confounds.

2.5 Defect Benchmarks

Several benchmarks of defects have evolved recently. Throughout this dissertation we use Defects4j [92] version 1.1.0 which consists of 357 defects observed and patched by developers during the development of five popular real-world open-source Java projects. Besides Defects4J, many other defect benchmarks have been released and published for different programming languages, sizes, expertise and proficiency levels. The ManyBugs benchmark [115] consists of 185 C defects in real-world software. The IntroClass benchmark [115] consists of 998 C defects in very small, student-written programs, although not all 998 are unique.

The Codeflaws benchmark [198] consists of 3,902 defects from 7,436 C programs mined from programming contests and automatically classified across 39 defect classes. The DBGBench benchmark [29] (based on the CoREBench benchmark [28]) contains a collection of 70 real regression errors in four open-source C projects. The QuixBugs benchmark [120] consists of 40 programs from the Quixey Challenge, where programmers were given a short buggy program and one minute to fix the bug. The programs are translated to Python and Java, and each bug is contained on a single line.

The Defects4J benchmark [93], originally designed for testing and fault-localization studies, consists of defects in real-world software, and has become a popular benchmark for evaluating automated program repair [56, 135, 150, 219]. We elected to use Defects4J because it contains real-world defects in large, complex projects, it supports reproducibility and test suite generation, and is increasingly a testbed for evaluating automated program repair.

Most prior evaluations of generate-and-validate repair techniques demonstrate by construction that the technique is feasible and reasonably efficient in practice [43, 97, 117, 123, 139, 153, 165, 166, 197, 209, 211]. Some show that the resulting patches withstand red team attacks [166], some illustrate with a small number of examples that generate-and-validate-generated patches for security vulnerabilities protect against exploits and fuzzed variants of those exploits on typical user workloads [117], and some consider the fraction of a set of bugs their technique can repair [56, 97, 99, 113, 153].

These evaluations have demonstrated that generate-and-validate techniques can repair a moderate number of bugs in medium-sized programs, as well as evaluated the monetary and time costs of automatic repair [113], the relationship between operator choices and test execution parameters and success [114, 210], and human-rated patch acceptability [1, 99] and maintainability [67]. However, these evaluations have generally not used an objective metric of correctness independent of patch construction. The evaluation used in this dissertation measures patch correctness independently of patch construction. This quality evaluation is designed to permit controlled evaluations that isolate particular features of the inputs, such that we can examine their effects on automatic repair and patch quality improvement in a statistically significant way.

Concurrent research is starting to evaluate repair techniques in terms of overfitting [172, 197]. Evaluating the degree to which *relifix* and GenProg introduce regression errors [197] is a step toward the independent correctness evaluation we advocate here, where we use independent test suites to measure patch quality. Poor-quality test suites result in patches that overfit to those suites [148, 172]. Our evaluation goes further, demonstrating that high-quality, high-coverage test suites still lead to overfitting, and identifying other relationships between test suite properties and patch quality.

Finally, prior and concurrent human evaluations of automatically-generated patches have measured acceptability [56, 99] and maintainability [67]. While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [163].

Our work evaluates automated repair so that it can be improved. Empirical studies of fixes of real bugs in open-source projects can also improve repair by helping designers select change operators and search strategies [93, 226]. Understanding how automated repair handles particular classes of errors, such as security vulnerabilities [117, 166] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [186, 188], unsafe integer use in C programs [43], single-variable atomicity violations [89], deadlock and livelock defects [121], concurrency errors [123], and data input errors [9]. Other techniques tackle generic bugs. For example, the ARMOR tool replaces buggy library calls with different calls that achieve the same behavior [35], and *relifix* uses a set of templates mined from regression fixes to automatically patch generic regression bugs. Our evaluation has focused on tools that fix generic bugs, but our methodology can be applied to focused repair as well.

User-provided code contracts, or other forms of invariants, can help to synthesize patches, e.g., via AutoFix-E [165, 209] (for Eiffel code) and SemFix [153] (for C). DirectFix [139] aims to synthesize minimal patches to be less prone to overfitting, but only works for programs using a subset of C language features, and has only been tested on small programs. Synthesis techniques provide the benefit of provable correctness for patches, but require contracts, so they are unsuitable for legacy systems. Synthesis techniques can also construct new features from examples [42, 75], rather than address existing bugs. Our work has focused on generate-and-validate approaches, and investigating overfitting and patch quality in synthesis-based techniques is a complementary and worthwhile pursuit. Our findings may extend to other search-based or test suite-guided repair techniques (e.g., [14, 51, 99, 139, 153, 158, 166, 210]).

2.6 Software Diversity

In this dissertation we propose a set of ways to improve diversity in the automatic program repair process. Similar to our proposed approach, there have been previous attempts to improve the quality of software by incentivizing diversity. One of the biggest motivations in this direction is N-Version Software (NVS), which is a way to take advantage of different implementations of code created following the same specification [18]. It was first introduced in 1977 as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification [16]. One of the major justifications for NVS was that it would be able to provide online tolerance for software faults, following the intuition that the independence of programming efforts will reduce the probability of identical software fault behavior. Our approach takes this same intuition applied in the context of APR where program fixes are created independently by construction, removing the risk of human bias and how humans tend to introduce similar errors in different software versions.

Some key experimental hands-on studies that have researched NVS are, for example, Avizienis [16] and Chen [38], where they implemented NVS systems using 27 and 16 independently

written versions; Ram [173] and Vog [205] have studied real-time software by developing six different implementations (programming languages) from the same requirements. Different from our study, "Diversity" in this context usually refers to the diversity of components (e.g., different compilers, programming languages, versions of the specifications [98], or different algorithms [38]). Even when software diversity is enforced through the variation of programming languages, developers tend to follow a "natural" sequence even when coding independent computations that could be performed in any order [17]. In this thesis dissertation, we avoid having this restriction since our patches are not generated by human developers and therefore do not follow any sequence that may seem "natural" to human programmers.

Another impediment to encouraging software diversity in software systems is when software specifications describe 'how' to implement portions of the code [17]. Because of this, identical errors were found in various versions. In this thesis, our specification is given by test cases, which describe examples of correct behavior. APR tools do not follow any instructions on "how" to build the patch, and the nature of the specification varies, removing these indications human developers have. Further research has advanced in relationship to software diversity metrics in the context of NVS [37, 132]. These studies focus mostly in the diversity of fault behavior. Robustness of software has also been analyzed in the context of operating systems using similar diversity metrics [102]. In this dissertation, we are not interested in focusing on fault behavior among versions but we are interested in creating diverse patches for the same bug.

Similarly, previous work has tried to improve diversity in genetic algorithms by implementing multi-objective search with goals different than increasing patch quality in APR. Panichella et al. [161] use multi-objective genetic algorithms (MOGA) to for test case selection as a means to reduce the cost of regression testing. Szubert et al. [195] describe how increasing diversity in genetic algorithms might lead to antagonism between behavioral diversity and fitness in the context of symbolic regression. Also previous work [32] has implemented techniques to maintain high-level search quality while increasing diversity.

More recently, artificial diversity has been proposed to improve the correct location of errors in software using "Mutation-Based Fault Localization" (MBFL) approaches [147, 162]. The intuition behind these techniques is that when mutants are generated at the faulty location, the test suite should exhibit different behavior than when mutants are generated in non-faulty locations. Further studies [164, 203] have suggested that MBFL techniques do not significantly distinguish between faulty and non-faulty locations. Smith et al. [190] compare the performance of single patches to hypothetical N-version patches, where the behavior of the N-version patches is described by a voting system. This paper introduces the usage of a held-out independently created test suite as a means for measuring software quality. The work introduced in this dissertation leverages these previous ideas to create real N-version patches with actual compilable code to be executed by the test cases.

Chapter 3

Experimental Approach Overview

This dissertation is comprised of a series of experiments highlighting ways to increase patch quality in automatic program repair. These experiments have a set of components in common to evaluate the hypothesis formulated in this thesis. These components are shared among the different studies described throughout this thesis and are shown in Figure 3.1.

Figure 3.1: Experimental approach is comprised of three components: The corpus of bugs to test our APR approaches, the APR tools we implemented and compared against, and the methodology to evaluate patch quality

The first component of our experimental approach is a corpus of defects used through this dissertation (Section 3.1). We chose Defects4J, a dataset and extensible framework containing 357 real bugs built to support software testing research. Bugs in Defects4J are comprised from five open-source Java projects: JFreeChart (26 bugs), Closure Compiler (133 bugs), Apache Commons Lang (65 bugs), Apache Commons Math (106 bugs), and Joda Time(27 bugs).

The second component of our approach overview is the set of APR approaches and techniques used to improve patch quality and execute our experiments. For this, we created JaRFly, an open-source extensible framework for Java repair[1]. It currently implements three APR approaches (GenProg, TrpAutoRepair, and PAR). GenProg [116] is an APR approach that uses coarse-grained mutation operators and genetic programming to generate patches; TrpAutoRepair [169, 171] uses single-edits and traverses the search space using random search to create candidate patches; and

[1] `https://github.com/squaresLab/genprog4java/`

identifier	project	description	KLoC	defects	tests	test KLoC
Chart	JFreeChart	Framework to create charts	85	26	222	42
Closure	Closure Compiler	JavaScript compiler	85	133	3,353	75
Lang	Apache Commons Lang	Extensions to the Java Lang API	19	65	173	31
Math	Apache Commons Math	Library of mathematical utilities	84	106	212	50
Time	Joda-Time	Date- and time-processing library	29	27	2,599	50
total			302	357	6,559	248

Figure 3.2: The 357 defect dataset created from five real-world projects in the Defects4J version 1.1.0 benchmark. We used `SLOCCount` to measure the lines of code (KLoC) counts (`https://www.dwheeler.com/sloccount/`). The **tests** and **test KLoC** columns refer to the developer-written tests.

PAR [99] is an APR technique that uses a set of templates to produce patches.

Finally, the third component of our experimental approach is the evaluation of patch quality. We created a methodology for creating high-quality held-out test suites to evaluate patch quality in an scalable manner. We have made this methodology publicly available and have created test suites used for evaluating the quality of a set of Defects4J bugs. Both JaRFly and the generated held-out test suites are publicly available for extension and scrutiny.

3.1 Real-World Defects and Test Suites

The first component in our experimental approach overview is the corpus of defects used throughout this thesis to evaluate error repair. To increase patch quality we require defects in which we can test our hypothesis. For the experiments described in this dissertation we used Defects4J version 1.1.0 [92], which consists of 357 defects made by developers during the development of five real-world open-source Java projects. Figure 3.2 describes the Defects4J defects and the projects they come from.

Each defect comes with (1) the source code necessary to replicate each bug (including the defective version of code containing the bug) and the code after the developer repaired the error; (2) a set of developer-written tests, all of which pass on the developer-repaired version and at least one of which shows the defect by failing on the defective version; and (3) the infrastructure to generate tests using modern automated test generation tools. Each defective version is a real-world version of the code.

The defective version was submitted to the project's repository by the developers actively working on the project under analysis. The developer-repaired version is a subsequent version of that code submitted by the project's developers that passes all the tests, minimized to only include changes relevant to repairing the defect.

Defects4J has been used to evaluate program repair in terms of how often techniques produce patches [55], what types of defects the techniques are able to patch [150], and the quality of the produced patches [108, 135, 216, 218]. These existing evaluations that measure patch quality use manual inspection [108, 135, 216] or automatically-generated evaluation test suites [108, 215, 218].

3.2 JaRFly: The Java Repair Framework

The second component of this experimental approach overview is the automatic program repair tools used throughout this thesis. For this purpose we have created JaRFly, an open-source framework for implementing techniques for automatic repair of Java programs. The implementation includes reimplementations of GenProg [113] and TrpAutoRepair [169] for Java (original releases of these tools were for C programs), and releases the first public implementation of PAR [99]. JaRFly is publicly available at `https://github.com/squaresLab/genprog4java/` to facilitate researchers and practitioners building automatic program repair approaches for Java programs.

JaRFly, as a framework, separates fundamental elements of APR and allows developers to modify those elements as necessary to create new approaches, leaving the rest of the implementation as default. These elements are problem representation, fitness function, mutation operators, and search strategy [79]. JaRFly provides an extensible set of interchangeable pieces for each of these elements. This differentiates our framework from prior work in this area [137].

JaRFly simplifies the process of implementing automatic program repair approaches for Java programs by parsing Java programs into a specified representation. It allows users to specify mutation operators, search strategy, and fitness function by selecting from a set of already implemented options, or by extending to their own custom made versions. Different from previous implementation of Java-based repair techniques [137] JaRFly makes APR elements explicitly interchangeable and this facilitates the extension and modification of said components. Following, we will detail these components of search-based repair and how JaRFly handles their implementation and extension.

Figure 3.3: JaRFly is an extensible automatic program repair framework for Java programs available at `https://github.com/squaresLab/genprog4java/`

3.2.1 Problem Representation

The way an APR approach represents the problem space affects its success and efficiency [114]. This problem representation therefore becomes a crucial part of the architectural design of an automated repair tool.

JaRFly represents patch candidates in a way that allows convenient manipulation and evaluation. This includes functionality to obtain information specific to each particular candidate, such as:

1. Localization information
2. Fitness evaluation

3. Serialization
4. Compilation
5. Test case execution
6. Program transformation using mutation operators

JaRFly represents plausible patches as sequence of edits to the original program [113, 114] minimizing the patch representation without losing syntactic information. Previous approaches [64, 211] represented patches as an abstract syntax tree (AST) making the patch representation much larger and therefore losing efficiency in the repair process. In addition to Java, variations of this representation can target Python [4] and C [156, 157] programs.

JaRFly provides a *Representation* abstract class and a patch representation for Java programs. Patches include mutation operators, location and, if needed, statement numbers as described in its internal representation, i.e. "Insert statement *S* at location *L*" where *Insert* represents the mutation operator being used, *L* represents a location within the program, and *S* indicates a statement from a prebuilt statement bank. When creating a new patch candidate by applying such mutations, JaRFly will add this mutation at the end of the patch representation.

Different from program repair tools that handle C code [113, 156, 157] JaRFly acknowledges that Java compilers are less permissive in allowing semantically incorrect code. Therefore JaRFly restricts the creation of patch candidates that would typically be permitted to compile in C, but if attempted in Java would create an exception. For example, Java compilers typically consider the addition of dead code to be an error, therefore if APR tools append an arbitrary statement after a *return statement*, such a patch candidate will not compile. Similarly, a *super method* can only be called as the first statement of a constructor otherwise it will show a compilation error. JaRFly handles cases as such to diminish the probability of creating non-compiling program variants.

3.2.2 Fitness Function

Search-based algorithms and particularly genetic algorithms use a *fitness function* to guide the traversal of the search space when evolving source code. This function determines the overall goal to achieve by the patch candidates.

The most common way in which a fitness function is designed in automated repair is by guiding the repair approach towards the *correctness* of the variants, which is usually measured by passing test cases from the guiding test suite. Alternative approaches [49, 54, 63, 111] have also proposed multi-objective fitness functions where the repair approach combines correctness with other kinds of incentives such as similarity to previous successful patches or learned invariants.

JaRFly provides a configurable and extensible `Fitness` class including several fitness strategies such as method and class level JUnit test execution. Similarly it includes the functionality for test sampling, which is used to execute a subset of the test suite to test variants' fitness before running the full suite to determine patch plausibility. The Fitness class also includes test selection which previous approaches [169, 210] have used to improve APR efficiency by prioritizing the execution of test that are more likely to fail.

This Fitness module can be easily extended to other potentially beneficial fitness computations such as the diversity-driven fitness analyzed in Chapter 6 of this thesis where I extend the

Fitness function of JaRFly to create a multi-objective fitness function to incentivize diversity and correctness in the automatic program repair process.

3.2.3 Mutation Operators

To generate patch candidates APR approaches need ways in which a program can be transformed into a variation of its original form. JaRFly provides the *EditOperation* abstraction which can be extended to transform Java programs into patches candidates. *EditOperation* is instantiated at a particular *Location* and depending on which mutation operation is being selected, it modifies the location accordingly. For example, an `Append` operation can be instantiated at any program point in a Java program and it will insert a selected statement at a particular *Location*.

JaRFly implements all statement-level operations used by GenProg [113] and TrpAutoRepair [169]:

1. Append Statement
2. Delete Statement
3. Replace Statement

Similarly JaRFly implements all PAR templates, including the optional templates[2] not included in the original paper [99]:

1. Null Checker
2. Parameter Replacer
3. Method Replacer
4. Parameter Adder and Remover
5. Object Initializer
6. Sequence Exchanger
7. Range Checker
8. Collection Size Checker
9. Lower Bound Setter
10. Upper Bound Setter
11. Off-by-one Mutator
12. Class Cast Checker
13. Caster Mutator
14. Castee Mutator
15. Expression Changer
16. Expression Adder

These mutation operators typically use code within the program to construct patch candidates, either by modifying targeted statements in the program or using information from the program to

[2] `https://sites.google.com/site/autofixhkust/home/`

generate new code. JaRFly provides information on legal *Location*s for each possible mutation operator to be applied and objects within scope of the *Location* to successfully create compiling patch candidates.

The code bank to chose statements from in JaRFly is by default a segmentation of the original program being modified. Previous studies show that programming languages are repetitive [81] and fix code is more likely found within the same module and project than from foreign modules and projects. However, JaRFly's code bank can be easily extended to include portions of code from external sources and therefore extend the reach of possible statements used to generate patch candidates.

JaRFly also includes a static legality checker which is composed of heuristics to reduce the possibility of creating non-compiling variants. All *EditOperation*s cannot be applied at all *Location*s. For example, the mutation operator *Parameter Replacer* modifies the parameters in a method call for a different set of parameters. This mutation operator can not be implemented in types of statements that do not include parameters such as a *Break Statement* (keyword used to terminate the execution of a loop or switch case). Checks as such improve efficiency in program repair by augmenting the number of compiling variants to validate and mutate while decreasing the time spent working over potentially non-compiling variants.

The abstract class *EditOperation* can be extended and instantiated to further produce new sets of mutation operators or to modify the way in which these mutation operators are selected to generate patch candidates [192].

3.2.4 Search Strategy

The *search strategy* defines the way in which APR approaches traverse the search space to find plausible patches, typically by using the fitness function to look for the a variant that satisfies its optimization goal. Common search strategies include local search, random search, and genetic programming. JaRFly implements an interface and a set of options to chose from, including random search, weighted brute force search, oracle search, genetic programming, and NGSA-II [50], a multi-objective evolutionary search strategy. Similar to previous elements of JaRFly, the *search strategy* can easily be extended to include more search strategies or to compare against the predefined ones.

3.2.5 Population Manipulation

JaRFly implements crossover and selection strategies common in evolutionary program manipulation. JaRFly includes one-point crossover, uniform crossover [211], and crossback crossover [211]. It also includes a default tournament selection strategy. In Chapter 6 of this thesis we extended this functionality to select variants with a higher diversity score for crossover in further generations.

Additionally, JaRFly is parameterized to allow for a configurable population size and mutation rate. More crossover and selection strategies can be easily added by extending the current implementations.

3.2.6 Localization and Code Bank Management

Fault localization is a known and broadly studied field within search-based program repair. A common and well-known family of approaches that make an effort to pinpoint the localization of errors in source code is called "spectrum-based fault localization" techniques. In these approaches, the main idea is to analyze the execution of passing and failing test cases through the targeted program, and create a weighted sum of the statements executed by each of these tests to understand which statements are more likely to contain the error (Jaccard [40], Ochiai [2], Ample [225], Tarantula [91], Wong [214]).

JaRFly implements common spectrum-based weighted path localization with configurable path weights, and an extensible abstract class for further extension to alternative localization strategies. JaRFly uses JaCoCo, an off-the-shelf library to compute coverage in Java programs for the purposes of fault localization [60].

The decoupling of these fundamental APR elements allows for a flexible and extensible program repair framework. In this thesis we have used this repair framework and extended several of the fundamental elements described above, which allows for ongoing research and experimentation of ways to enhance APR components to increase patch quality of the generated plausible patches.

3.3 Quality Evaluation

Previous studies [122, 190] have shown that automated program repair is prone to producing patches that overfit to the guiding test suites. Within the space of possible program modifications, many patches can be created where the variant passes all the supplied tests. Guiding test suites describe a partial description of the desired behavior and therefore it is common that the generated plausible patches fail to generalize to the full intended specification and result in low-quality patches. This phenomenon of automated program repair producing patches that satisfy the partial specification of the supplied test suite, but failing to generalize is called overfitting [122, 190].

Since then, research has measured the degree to which generate-and-validate patches overfit and what factors affect that overfitting on small C programs [190], how often these patches disagree with developer-written patches [172], how often overfitting happens in Java repair [56, 135], and what is the concentration of correct patches [128].

Additionally, research has attempted to improve on the quality of the patches produced by using semantic search to increase the granularity of repair [96], condition synthesis [126], learning patch generation patterns from human-written code [129], and automated test case generation [215]. Further, research has found that overfitting occurs in APR tools targeting different programming languages and repair families [112, 122] given their reliance on a partial specification. Even when repair uses manually-written contracts as the desired behavior specification, which are more complete than tests, APR approaches still overfit, producing correct patches for only 59% of the analyzed defects [165].

The main goal of this thesis is to create higher quality patches by enhancing key components of the automatic program repair process. In this context, **patch quality** becomes a fundamental concept that must be measurable and quantifiable. Since software system functionality is described

by subjective human requirements, determining whether one patch is "better" than another is often difficult to assess. A perfect oracle would be able to check the program formally against a full specification, but given the nonexistence of such specifications and oracles in practice, we are forced to find alternatives.

Given these restrictions, there are two established methods for evaluating quality of program repair, using an independent test suite not used during the construction of the repair [31, 190], and manual inspection [135, 172].

The two methodologies are complementary. The methodology that uses an independent test suite is more objective, whereas manual inspection is more subjective and can be subject to subconscious bias, especially if the inspectors are authors of one of the techniques being evaluated. Manual inspection has been used to measure how maintainable the patches are [67] and how likely developers are to accept them [99]). However, a recent study found that manual-inspection-based quality evaluation can still be imprecise [108].

3.3.1 Evaluating Patch Quality Through Held-out Test Suites

Held-out-test-suite based quality evaluation is inherently partial, as the independent test is a partial specification. Therefore, the results of this technique can also be inaccurate by mislabeling a patch as correct when there might exist untested cases that show the incorrectness of the patch.

In this thesis, we will use the test-suite-based quality evaluation method because (1) it is objective and reproducible in a fully automated manner, (2) can scale to complex, real-world defects in real-world systems, which are the focus of our work (manual inspection would require using the projects' developers with domain knowledge), (3) remove the possibility of subconscious bias [108] in potential human evaluators.

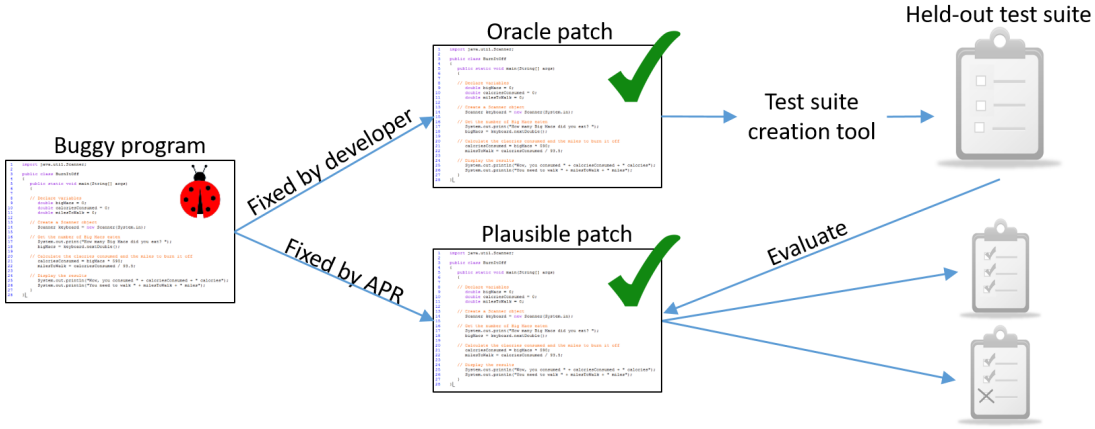


Figure 3.4: Evaluating the quality of generated plausible patches based on a held-out test suite generated from a developer patch. For each buggy program we create *plausible patches* using APR techniques. To evaluate their quality we generate a held-out test suite from the human-generated patch (i.e., the *oracle patch*) and execute that held-out test suite in the plausible patch. The quality is based on the percentage of passed test cases from the held-out test suite.

For the experiments described in this dissertation we use two independent test suites that

specify the desired behavior of the program being repaired. One test suite can be used by the automated program repair techniques to produce a patch for a defect (which we call *guiding* test suite). The second, independent test suite we called it the *held-out* or *evaluation* test suite; this test suite is used to measure the patch's quality. As already mentioned, each Defects4J defect comes with a developer-written test suite that evidences the defect. To create the evaluation test suite, for each defect, we generated test inputs using an off-the-shelf automated test input generator on the developer-repaired code.

Figure 3.4 shows a buggy program that is later patched by a human developer (provided as a developer patch from Defects4J) as an approximation to an "oracle" patch. We then use an off-the-shelf test suite creation tool [65]) to generate a held-out test suite that describes the behavior of the oracle patch.

3.3.2 Analyzing Test Generation Tool Behavior

Evaluating patch quality through a held-out test suite is only effective if the evaluation test suite is of high quality. Coverage is widely-used in industry to estimate test-suite quality [86]. Using statement-level code coverage as a proxy for test suite quality, our goal was to generate, for each defect, a high-coverage test suite, thus implying that a big portion of the functionality of the inspected class is being evaluated. Specifically, we focused on the statement coverage of the methods and classes modified by the developer-written patch and designed a test generation methodology aimed to maximize that coverage.

Ideally, we want the evaluation test suite to have perfect coverage, but modern automated test generation tools cannot achieve perfect coverage on all large real-world programs, in part because of limitations of such tools such as possible infinite recursion in the creation process or impreciseness of method signatures such as Java generics [66]. Thus, we set as our goal to generate, for each defect, a test suite that achieves 100% coverage on all developer-modified methods, and at least 80% coverage on all developer-modified classes. The choice of coverage criteria is a compromise between a reasonable measure of covering all the developer changes and the modern automated test generation tools' ability to generate high-coverage test suites.

To achieve this coverage threshold we first compared the effectiveness of two modern off-the-shelf automated test generators Defects4J supports, Randoop [160] and EvoSuite [66], in a controlled fashion, and found that EvoSuite consistently produced test suites with higher coverage on Defects4J defects' code. This finding is consistent with prior analyses [184]. Accordingly, we elected to use EvoSuite as our test suite generator.

EvoSuite uses randomness in its test generation and continues to generate tests up to a given time budget, so we experimented with different ways to run EvoSuite to maximize coverage. We ran EvoSuite using branch coverage as its target maximization search criterion (the default option) twenty times per defect, with different seeds, ten times for 3 minutes and ten times for 30 minutes. We found low variance in the coverage produced by the generated test suites: the 3-minute test suites had a variance in statement coverage of 0.6% and the 30-minute test suites of 0.8%.

We also found that the improvement between the mean statement coverage of the 3-minute test suites and the mean statement coverage of the 30-minute test suites was low (from 68% to 72%), suggesting that longer time budgets would not significantly improve coverage. Merging ten 3-minute test suites resulted in higher statement coverage than a single average 30-minute

test suite (77% vs. 72%). Finally, merging ten 30-minute test suites resulted in 81% statement coverage, on average, the highest we observed. We thus used the ten merged 30-minute test suites as preferred combination mechanism to optimize test suite coverage.

3.3.3 Creating High-Quality Held-Out Test Suites

We executed the following automated process for generating the test suites: For each defect, we ran EvoSuite (v1.0.3) ten times (on different seeds) with a 30-minute time budget and merged the ten resulting test suites, removing duplicate tests. We then checked if the resulting test suite covered 100% of the statements in the developer-modified methods, and at least 80% of the statements in each of the developer-modified classes. For 30 out of the 68 defects, this algorithm generated test suites that satisfied the coverage criterion. This process is described in Algorithm 1 where $coverage(T_{eval}, covm, covc) == true\ iif\ methodCoverage = covm \wedge classCoverage \geq covc$. As detailed in our description for our evaluation test suites we used the values $covm := 100$ and $covc := 80$.

Algorithm 1 to generate evaluation test suite using EvoSuite

1: **procedure** GENERATETESTSUITE($covm, covc$)
2: ▷ For a given defect, generate a test suite which covers at least *covm* percent of the developer-modified method and at least *covc* percent of the developer-modified class
3: $T_{eval} \leftarrow \{\}$ ▷ initialization an empty set
4: $runs \leftarrow 10$ ▷ number of times EvoSuite is run
5: $timebudget \leftarrow 30$ ▷ time budget for each run (mins)
6: $criterion \leftarrow branch/line$ ▷ criterion to optimize for in each run
7: **while** $runs > 0$ **do**
8: $runs \leftarrow runs - 1$
9: $T \leftarrow$ genTestEvosuite($timebudget$) ▷ generate tests by running EvoSuite for 30 mins
10: $T_{eval} \leftarrow Distinct(T_{eval} \cup T)$ ▷ merge the generated suite into evaluation suite after removing duplicate tests
11: **if** $coverage(T_{eval}, covm, covc) == true$ **then**
12: **return** T_{eval} ▷ generated suite satisfies the coverage requirements
13: **else**
14: $T'_{eval} \leftarrow ManuallyAugment(T_{eval})$ ▷ augment generated test suite with manually written tests to satisfy the coverage requirements
15: **if** $coverage(T'_{eval}, covm, covc) == true$ **then**
16: **return** T'_{eval} ▷ augmented suite satisfies the coverage requirements
17: **return** "cannot generate test suite" ▷ coverage criterion cannot be met

In the course of our study, a new version of EvoSuite was released. We attempted to augment the test suites by using this later version of EvoSuite (v1.0.6), but this new version did not produce better-coverage test suites than v1.0.3 on its own. However, using statement-coverage as the target maximization search criterion (instead of the default branch coverage) did produce test suites that, when combined with the previous v1.0.3-generated test suites, improved coverage. This

defect set	# of defects	statement coverage of patch-modified	mean	median
at least one patch	68	methods	89.7%	100.0%
		classes	88.5%	97.5%
adequate test suite	45	methods	100.0%	100.0%
		classes	97.1%	98.7%

Figure 3.5: Statement coverage of the EvoSuite-generated test suites for the 68 Defects4J defects patched by at least one repair technique in our study, and for the 45-defect subset for which our generated test suites covered 100% of all developer-modified methods and at least 80% of all developer-modified classes.

process resulted in test suites that satisfied the coverage criterion for a total of 37 defects (7 Chart, 3 Closure, 8 Lang, 17 Math, and 2 Time defects).

We then examined the generated test suites that met one, but not both of the coverage criteria and attempted to manually augment them to fully meet the other criterion. Examining these cases, we found that EvoSuite often was unable to cover statements that required the use of specific hard-to-generate literals present in the code. For example, covering some portions of code from the Closure project (a JavaScript compiler) required tests that take as input specific strings of JavaScript source code, such as an inline comment. Meanwhile covering some exceptional Lang code required specific strings to trigger the exceptions. The probability of the random strings generated and selected by EvoSuite to match the necessary strings to cover these portions of the code is negligibly small.

We, therefore, manually examined the source code and created test cases using the necessary literals. Augmenting the EvoSuite-generated test suites with these manually-written tests resulted in test suites for 8 more defects (1 Chart, 2 Closure, 4 Lang, and 2 Math, defects) that satisfied the coverage criteria.

In total, this process produced test suites that satisfied the coverage criterion for 45 of the 68 defects (8 Chart, 5 Closure, 11 Lang, 19 Math, and 2 Time defects). We restrict our study to these 45 defects. An additional 5 defects had 80% or higher coverage on the developer-modified classes, but did not have 100% coverage on the developer-modified methods. The mean statement coverage for the developer modified classes for these 45 defects is 97.1% and median is 98.7% (with means and medians for the modified methods both 100%, as required by the coverage criterion). Figure 3.5 summarizes these statistics for the 45 defects used in our study and the 68 defects patched by at least one repair technique.

Chapter 4

Analyzing the Role of Test Suites in the APR Process

Automatic program repair has the ability to generate plausible patches given a program with an error and a guiding test suite describing the desired program behavior. This guiding test suite is a fundamental component in the automatic program repair process since it is the main component the APR approach has to reason about the expected behavior of the program. This test suite works as a partial specification of the desired program describing both correct behavior to maintain (positive test cases), and erroneous behavior to modify (negative test cases). The triggering criterion to declare a patch candidate as a plausible patch is when all test cases in this test suite pass. A low-quality guiding test suite might easily lead the APR approach to create plausible but incorrect patches [190] which generate correct outputs for all test cases but where the repair does not fully address the underlying error needed to be fixed [122, 130]. This raises concerns about the usability of automated repair approaches, and outlines possible paths toward building techniques that produce higher-quality patches [96, 126, 129, 141, 192].

Prior work [190] introduced the methodology for evaluating patch quality described in Section 3.3 for a group of small programs written by students in an introductory course to programming. This study outlined the importance of overfitting in automatic program repair and consequently the influence that the guiding test suite has on the quality of generated patches. This study analyzed the behavior of APR in small programs and had several limitations (only considered two generate-and-validate approaches, did not control for confounding factors, and used test suite size as a proxy for coverage). Building up on previous work, in this thesis chapter we answer five research questions [1]:

RQ1 Do generate-and-validate techniques produce patches for real-world Java defects?

Answer: The generate-and-validate approaches executed in our experiment were able to generate patches for 68 out of 357 real-world defects.

RQ2 How often and how much do the patches produced by generate-and-validate techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite,

[1] Portions of the work described in this Chapter have been published in the journal IEEE Transactions on Software Engineering [148]

and thus ultimately to the program specification?

Answer: We conclude that tool-generated patches on real-world Java defects often overfit to the test suite used in constructing the patch, often breaking more functionality than they repair.

RQ3 How do the coverage and size of the test suite used to produce the patch affect patch quality?

Answer: We conclude that both test suite size and test suite coverage have extremely small but statistically significant correlations with patch quality (positive for test suite size in all techniques and negative for test suite coverage in two techniques) produced using automatic program repair techniques.

RQ4 How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

Answer: We conclude that the number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the patches produced using automatic program repair techniques and that this finding depends on the fault localization strategy used by the repair techniques.

RQ5 How does the test suite provenance (whether written by developers or generated automatically) influence patch quality?

Answer: We conclude that test suite provenance has a significant effect on repair quality, though the effect may differ for different techniques. For GenProg and TrpAutoRepair, patches created using automatically-generated tests had lower quality than those created using developer-written test suites. For a smaller, perhaps non-representative number of defects, PAR-generated patches showed the opposite effect.

In the following Sections we evaluate generate-and-validate repair techniques and the resulting patch quality of plausible patches generated by performing a series of experiments using the Defects4J dataset described in Section 3.1 and the quality evaluation test suites described in Section 3.3. Section 4.1 describes an overview of how successful are the techniques from JaRFly at producing patches on real-world defects. Finally, Section 4.2 further analyzes the quality of these patches and provides insight at what attributes from the guiding test suites have a larger impact in patch quality.

4.1 Ability to Produce Plausible Patches

Research Question 1: Do generate-and-validate techniques produce patches for real-world Java defects?

The first step in this study is to get a general sense of how effective are automated repair approaches in fixing real-world defects.

Methodology:

We used each of the three repair techniques included in JaRFly to attempt to repair the 357 defects in the Defects4J benchmark providing the developer-written test suite as the guiding test

suite to all the techniques. For GenProg, PAR, and TrpAutoRepair, which select random mutation operators to generate a patch, we attempt to repair each defect 20 times with a timeout of 4 hours each time, using a different seed each time, for a total of $357 \times 20 = 7{,}140$ attempted repairs, per each repair technique.

We ran these techniques using a cluster of 50 compute nodes, each with a Xeon E5-2680 v4 CPU with 28 cores (2 processors, 14 cores each) running at 2.40GHz. Each node had 128GB of RAM and 200GB of local SSD disk. We launched multiple repair attempts in parallel, each requesting 2 cores on one compute node. The 20 repair attempts provided a compromise between the likely ability to make statistically significant findings, and the computational resources necessary to run our experiments. The computational requirements are substantial: Repairing a single defect 20 times with a 4-hour timeout can take 80 hours per defect per repair technique. If we were to run this experiment sequentially for the 357 defects and 3 repair techniques, it would take 10 CPU-years.

The repair techniques' parameters affect how they attempt to repair defects. For GenProg, PAR, and TrpAutoRepair, we used the parameters from prior work that evaluates these techniques on C programs [99, 113, 169]. We set the population size (`PopSize`) to 40 and the maximum number of generations to 10 for all three techniques. For GenProg and TrpAutoRepair, we uniformly equally weighted the mutation operators `Append`, `Replace`, and `Delete`. For PAR, we uniformly equally weighted the mutation operators `FUNREP`, `PARREP`, `PARADD`, `PARREM`, `EXPREP`, `EXPADD`, `EXPREM`, `NULLCHECK`, `OBJINIT`, `RANGECHECK`, `SIZECHECK`, and `CASTCHECK`.

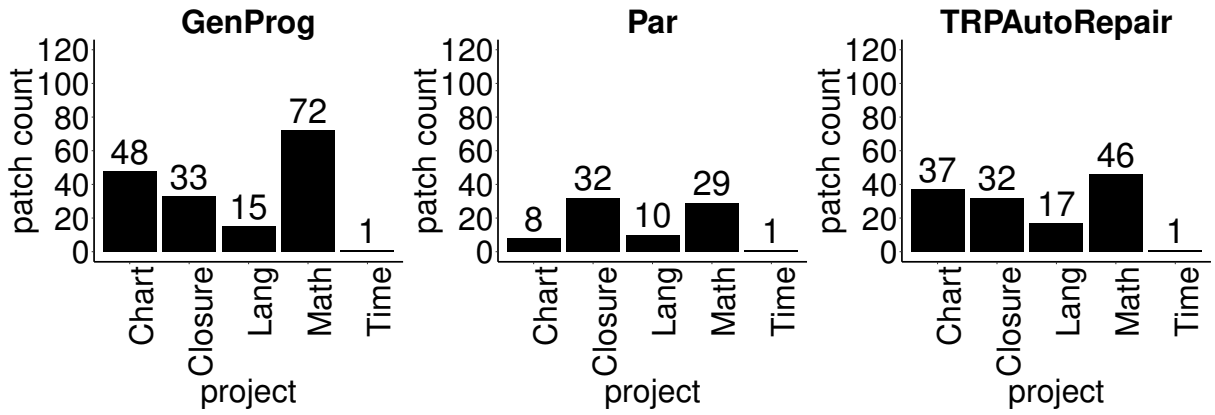
For GenProg and PAR, we set `SampleFit` to 10% of the test suite. For fault localization, all three techniques apply a simple weighting scheme to assign values to statements based on their execution by passing and failing tests. For PAR and TrpAutoRepair, we set `negativePathWeight` to 1.0 and `positivePathWeight` to 0.1, based on prior work [99, 169]. For GenProg, we set `negativePathWeight` to 0.35 and `positivePathWeight` to 0.65 [114]. For all remaining parameters, we use their default values from prior work [99,113,169]. We describe the complete set of parameters at `https://github.com/LASER-UMASS/JavaRepair-replication-package/wiki/Configuration-parameter-details/`.

technique	patches total	patches unique	defects patched
GenProg	585 (8.2%)	255	49 (13.7%)
PAR	288 (4.0%)	107	38 (10.6%)
TrpAutoRepair	513 (7.2%)	199	44 (12.3%)
total	1,298 (6.1%)	561	68 (19.0%)

(a) Produced patches

Figure 4.1: GenProg, PAR, and TrpAutoRepair produce patches 1,298 times (6.1%) out of the 21,420 attempts. At least one technique can produce a patch for 68 (19.0%)) of the 357 real-world defects.

Results:



(a) Unique patch distributions, per technique

Figure 4.2: The distributions of unique patches produced by the three techniques are similarly shaped.

Figure 4.1 reports the results of the repair attempts. GenProg patches 49 out of 357 defects (6 Chart, 15 Closure, 9 Lang, 18 Math, and 1 Time) and creates a total of 585 patches, out of which 255 are unique. Search-based approaches are able to find several patches per each defect, thus we report the total number of patches; and given that these approaches use an stochastic approach to find repairs, some of the patches found can be repeated among the different seeds, therefore we also report the number of unique (non-repeated) patches. PAR patches 38 out of 357 defects (3 Chart, 12 Closure, 7 Lang, 15 Math, and 1 Time), and produces a total of 288 patches, out of which 107 are unique. TrpAutoRepair patches 44 out of 357 defects (7 Chart, 12 Closure, 8 Lang, 16 Math, and 1 Time) and produces a total of 513 patches, out of which 199 are unique.

Overall, at least one technique produced at least one patch for 68 out of the 357 defects. All techniques produced at least one patch for 18 defects. GenProg most often produced patches (8.2% of the 7,140 attempts) and produced patches for the most defects (13.7%). Figure 4.2 shows the distributions of unique patches, per project, generated by each of the three techniques.

Compared to prior studies on C defects [189], [115, 169], the Java repair mechanisms produce patches on fewer repair attempts and for fewer defects. On C defects, GenProg produced patches for between 47% (ManyBugs defect dataset) and 60% (IntroClass defect dataset) and TrpAutoRepair produced patches for between 52% (ManyBugs) and 57% (IntroClass) defects. These previous studies evaluated smaller and simpler programs, therefore a lower repair rate when using more complex and larger real-world defects is expected.

Our findings are also consistent with prior work applying generate-and-validate repair to Java defects, which found techniques to produce patches for 9.8%–15.6% of the defects [135]. In a prior study on Java defects, PAR produced patches for 22.7% of the defects [99]. Some of the prior study's defects came from Lang and Math, projects that are also part of Defects4J (though a different set of defects), and our results on those projects are similar to those in the prior study [99].

Answer to Research Question 1: The generate-and-validate approaches executed in our experiment were able to generate patches for 68 out of 357 real-world defects.

4.2 Analyzing Plausible Patch Quality

Section 4.1 showed that generate-and-validate techniques are able to patch 19.0% of the real-world defects in Defects4J. This Section explores the quality of the produced patches and measures the factors that affect it. These experiments are based on the 45 defects for which we are able to generate high-quality evaluation test suites (recall Section 3.3). These 45 defects are a subset of the 68 defects for which at least one repair technique produced at least one patch.

4.2.1 Patch Overfitting

Research Question 2: How often and how much do the patches produced by generate-and-validate techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

	patch quality				100%-quality
technique	**minimum**	**mean**	**median**	**maximum**	**patches**
GenProg	64.8%	95.7%	98.4%	100.0%	24.3%
PAR	64.8%	96.1%	98.5%	100.0%	13.8%
TrpAutoRepair	64.8%	96.4%	98.4%	100.0%	19.5%

Figure 4.3: The quality of the patches the repair techniques generated when using the developer-written test suite varied from 64.8% to 100.0%.

Methodology:

To measure the quality of a produced patch, we start with the defective code version, apply the patch to that code, and execute the generated evaluation test suite. We call the total number of tests executed in the evaluation test suite T_{total} and the number of tests the patched version passes T_{pass}. The quality of a patch is $\frac{T_{pass}}{T_{total}}$, as defined by prior work [189]. A patch that passes all the tests in the evaluation test suite has 100% patch quality.

Similarly, we measure the quality of the defective code version by executing the evaluation test suite prior to applying the patch. We can therefore measure the quality improvement when the patch is applied.

Results:

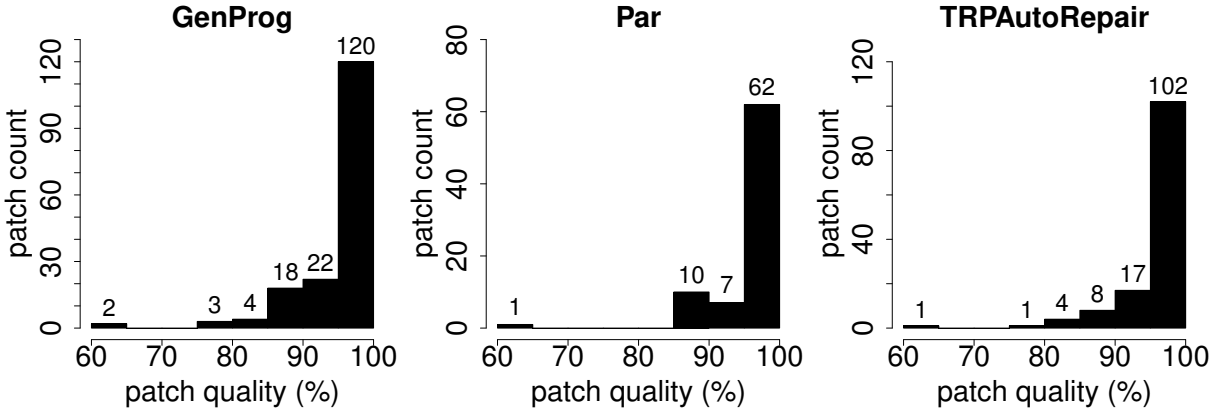


Figure 4.4: The distributions of patch quality is skewed toward the 100% end. On average, 80.8% (GenProg: 75.7%, PAR: 86.2% and TrpAutoRepair: 80.5%) of the patches failed at least one test.

Figures 4.3, and 4.4 show the distributions of the quality of the patches produced by each technique. It is possible for all techniques to find the same patch for some defects, which, in this case, resulted in all the three techniques displaying the same overall minimum patch quality.

Overall, 80.8% of the patches (GenProg: 75.7%, PAR: 86.2%, and TrpAutoRepair: 80.5%), on average, failed at least one test, thus overfitting to the specification and failing to fully repair the defect. The mean quality of the patches varied from 95.7% to 96.4%. It is noteworthy to understand throughout this dissertation that the relatively high fraction of tests passed in the quality evaluation is not necessarily a proportional indication of the quality of repair. Defective code versions already pass 98.3% of the tests on average, thus a patch that passes 96.0% of the tests may not even be an improvement over the defective version. The reason for the high percentage numbers in quality evaluation is because of the high number of tests generated by the test-suite-generation tools used to create the held-out test suites. Held-out test suites evaluate that previously erroneous behavior was fixed, however they also evaluate that previously correct functionality is maintained which is a considerable portion of the test cases that pass in the patched version.

Next, we consider whether patches improve program quality. Figures 4.5,4.6, and 4.7 show, for each of the patched defects, the change in the quality between the defective version and the patched version. A negative value indicates that the *patched* version failed more evaluation tests than the *defective* version. When a technique produced multiple distinct patches for a defect, for this comparison, we used the highest-quality patch.

In Figures 4.8, and 4.9 we aggregated the results per APR approach. For GenProg, 33.3% of the defects' patches improved the quality, 42.5% showed no improvement, and the remaining 24.2% decreased quality. For PAR, 20.0% improved, 40.0% showed no improvement, and 40.0% decreased quality. For TrpAutoRepair, 32.3% improved, 25.8% showed no improvement, and 41.9% decreased quality. For PAR and TrpAutoRepair, more patches broke behavior than repaired it, and the decrease in quality was, on average, larger than the improvement. For all the techniques, the majority (63 out of 89, 70.7%) of the patches decrease or fail to improve quality: 22 patches from GenProg, 20 from PAR, and 21 from TrpAutoRepair; and more than a third (31 out of 89, 34.8%) of the patches break even more tests than they fix: 8 patches from GenProg, 10 from PAR,

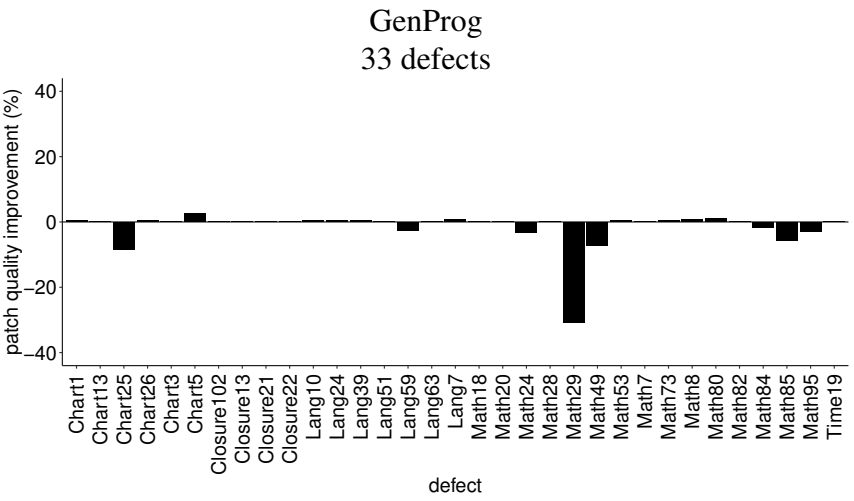


Figure 4.5: Change in quality between the defective version and the patched version of the code per each defect. GenProg created repairs for 33 defects

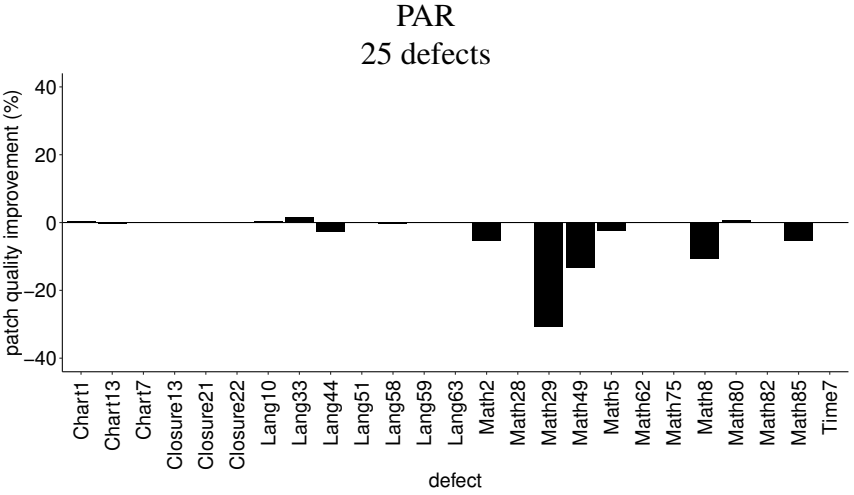


Figure 4.6: Change in quality between the defective version and the patched version of the code per each defect. PAR created repairs for 25 defects

and 13 from TrpAutoRepair.

These results are consistent with the previous findings obtained using C repair techniques on small programs, where the median GenProg patch passed only 75% (mean 68.7%) of the evaluation test suite and the median TrpAutoRepair patch passed 75.0% of the evaluation test suite (mean 72.1%) [189].

Answer to Research Question 2: We conclude that tool-generated patches on real-world Java defects often overfit to the test suite used in constructing the patch, often breaking more functionality than they repair.

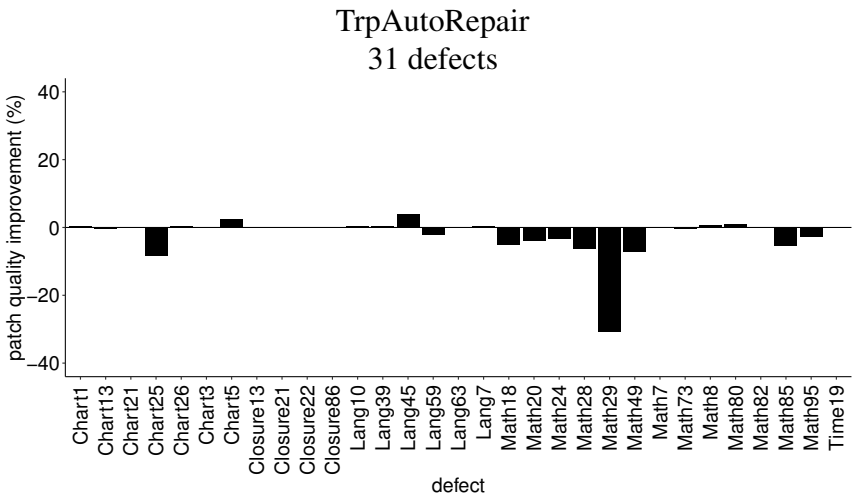


Figure 4.7: Change in quality between the defective version and the patched version of the code per each defect. TrpAutoRepair created repairs for 31 defects

4.2.2 Test Suite Coverage and Size

Research Question 3: How do the coverage and size of the test suite used to produce the patch affect patch quality?

Previous work [189] used test suite size to approximate test suite coverage. In this study we measure the actual statement-level code coverage of the used guiding test suites, and control for confounding factors, such as test suite size, defects' project, and the number of failing tests. For our dataset, we found statistically significant weak positive correlation ($r = 0.14$) between test suite size and statement-level coverage of the developer-written tests (guiding test suite) on the defective code version. This is consistent with the prior studies [92].

Methodology:

To measure the relationship between test suite coverage and repair quality, we attempted to create subsets of the guiding (developer-written) test suite of varying coverage while controlling for test suite size, number of failing tests, and the defects themselves. Test suite coverage and test suite size are positively correlated, therefore analyzing their association with repair quality individually would not be appropriate. We used multiple linear regression to identify the relationship between two independent variables (test suite coverage and test suite size) and their corresponding dependent variable (patch quality).

For this analysis, we considered the 46 defects for which we created high-quality evaluation test suites. For each of the defects, we created subsets of the developer-written test suite of varying coverage. Each subset contains all the tests that evidence the defect, and randomly selected subsets of the rest of the tests. We then used the repair techniques included in JaRFly to produce patches using these test suite subsets using the methodology from Section 4.1. Finally, we computed the quality of the patches produced for each defect using the automatically-generated high-quality evaluation test suites. We excluded defects for which we could not generate test suites

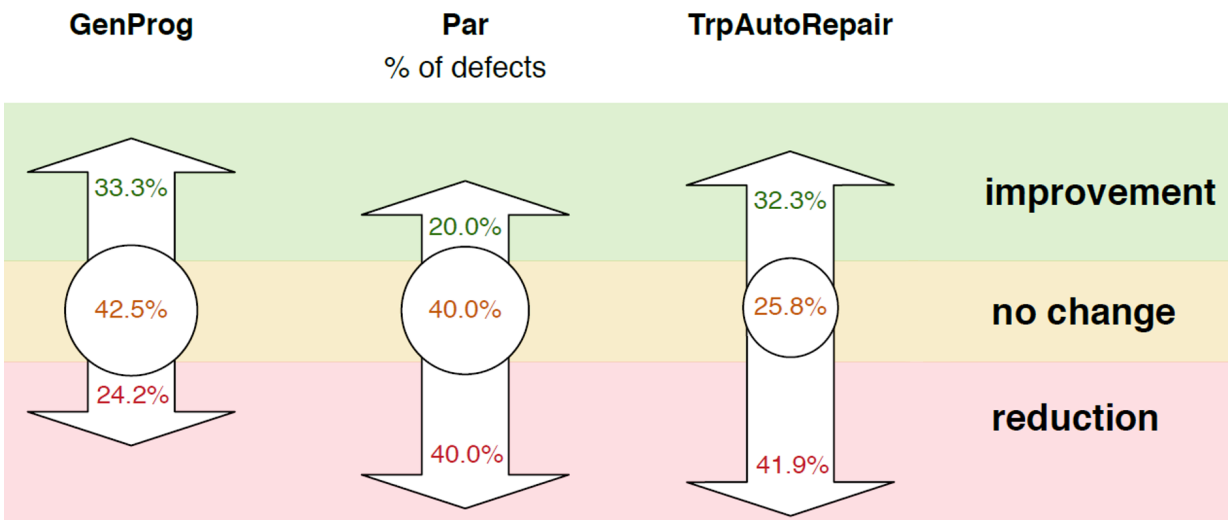


Figure 4.8: Patch overfitting. Aggregated change in quality between the defective version and the patched version of the code. The median patch neither improves nor decreases quality. While more GenProg patches improve the quality than decrease it, the opposite is true for PAR and TrpAutoRepair patches, and, on average, patches break more functionality than they repair.

	change in quality due to patch			
technique	**minimum**	**mean**	**median**	**maximum**
GenProg	−30.9%	−1.7%	0.0%	2.6%
PAR	−30.9%	−2.8%	0.0%	1.5%
TrpAutoRepair	−30.9%	−2.1%	0.0%	3.8%

Figure 4.9: The data presented is for the 46 defects with high-quality evaluation test suites, of which GenProg produced patches for 33, PAR for 25, and TrpAutoRepair for 31.

with sufficient variability in coverage, and for which we did not have sufficiently high-quality evaluation test suites.

To generate the test suite subsets for each defect, we first compute the minimum and the maximum code coverage ratio of the developer-written test suite of that defect. The *minimum code coverage ratio* (cov_{min}) of a developer-written test suite is the statement coverage on the defective code version when executing only the failing tests. The failing tests are the minimum number of tests necessary to run our APR techniques, thus we include them in every subset we generate. The *maximum code coverage ratio* (cov_{max}) is the statement coverage on the defective code version of the entire developer-written test suite (the largest possible subset).

For example, for Chart 1, there is 1 failing test and 245 passing tests that execute the developer-modified class `AbstractCategoryItemRenderer`. The minimum coverage, (cov_{min}), for Chart 1 is the statement coverage of the single failing test on the developer-modified class. This test covers 18 out of the 519 lines, (3.5%). The maximum coverage, (cov_{max}), for Chart 1 is the statement coverage of the full test suite (246 tests) on the developer-modified class. This test suite covers

300 out of the 519 lines, (57.8%).

We then compute the guiding test suite coverage variability as the difference between the minimum and the maximum: $\Delta_{cov} = cov_{\max} - cov_{\min}$ following the procedure described in Algorithm 2. Defects whose $\Delta_{cov} < 25\%$ lack sufficient variability in statement coverage to be used in this study and we discard them. In our study, we discarded 18 defects for this reason (1 Chart, 7 Closure, and 10 Math) out of the 68 defects that had at least one repair technique produce at least one patch.

Algorithm 2 to produce a test suite subset

```
1:  procedure SAMPLETESTSUITECOVERAGE(T, c)
    ▷ Produce a test suite with coverage c that is a subset of T
2:      P ← allPassingTests(T)
3:      S ← allFailingTests(T)                          ▷ Start with all failing tests
4:      attempt ← 0
5:      while coverage(S) < (c − 0.05) do
          attempt ← attempt + 1
6:          if attempt = 500 then return "could not generate suite"
7:          p ← a uniformly randomly selected test in P, without replacement
8:                                  ▷ If adding p does not overshoot coverage c, add p:
9:          if coverage(S ∪ {p}) < (c + 0.05) then
10:             S ← S ∪ {p}
11:     return S
```

For each of the 50 defects, we chose five target coverage ratios evenly spaced between the minimum and the maximum and try to generate subsets of tests that exhibit this coverage ratio: $cov_{\min} + \frac{1}{5}\Delta_{cov}$, $cov_{\min} + \frac{2}{5}\Delta_{cov}$, $cov_{\min} + \frac{3}{5}\Delta_{cov}$, $cov_{\min} + \frac{4}{5}\Delta_{cov}$, and $cov_{\min} + \Delta_{cov} = cov_{\max}$.

Given that there are multiple ways to achieve each target coverage, we attempt to generate 5 different subsets per each target ratio, therefore creating a total of 25 distinct sub test suites per each defect. In the subset generation process we allowed a 5% margin of error given that it is commonly difficult (or sometimes impossible) to achieve the exact target ratio.

For each sub test suite, we started with all tests that fail on the defective code version and pass on the developer-repaired code version. We then iteratively attempted to add a uniformly randomly selected passing test case, without replacement, one at a time, as long as it did not make the subset's coverage exceed the target by more than 5%, stopping if the subset's coverage was within 5% of the target.

If we attempted to add a randomly selected test 500 times and failed to reach the target, we stopped as detailed in Algorithm 2. For 5 of the 50 defects (1 Chart, 1 Closure, 2 Lang, and 1 Math), the sampling algorithm was unable to generate five distinct test suite subsets, so we discard these five defects. We consider the remaining 45 defects for the analysis.

Results:

For each of the 45 defects, we had 25 test suite subsets, and we attempted each repair 20 times using GenProg, PAR, and TrpAutoRepair on different seeds. In total, these 22,500 repair attempts produced 8,371 patches. Figure 4.10 shows the distribution of these patches. GenProg produced

at least one patch for 29 out of the 45 defects, PAR 25, and TrpAutoRepair 29. (GenProg: 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time; PAR: 5 Chart, 1 Closure, 8 Lang, 10 Math, and, 1 Time; and TrpAutoRepair 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time.)

Figure 4.11 shows the statistics of the quality of the patches for those defects, created using the varying-coverage test suites. The quality varied, with GenProg even producing some patches that failed *all* evaluation test cases. Overall, 81.2% of the patches, on average, failed at least one test in the evaluation test suite.

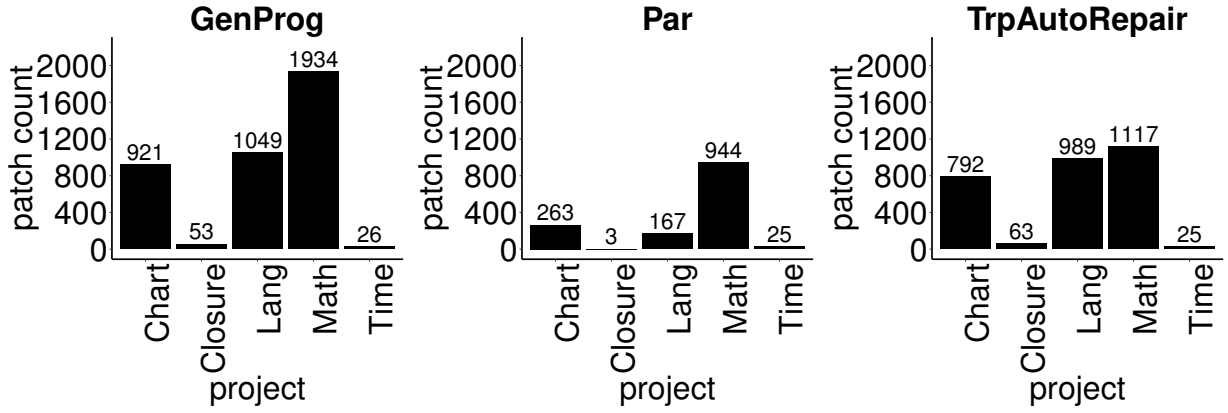


Figure 4.10: Distribution of patches generated using varying-coverage test suites. Distribution of the number of patches produced using developer-written test suite subsets of varying code coverage on the defective code version.

	patch quality				100%-quality
technique	**minimum**	**mean**	**median**	**maximum**	**patches**
GenProg	0.0%	94.8%	98.4%	100.0%	16.2%
PAR	51.8%	91.2%	95.5%	100.0%	13.3%
TrpAutoRepair	62.9%	95.5%	99.0%	100.0%	19.0%

Figure 4.11: Quality of patches generated using varying-coverage test suites. The quality of the patches generated using varying-coverage test suites varied from 0.0% to 100.0%. On average, 83.83% (GenProg: 83.8%, PAR: 86.7%, and TrpAutoRepair: 81.0%) of the patches failed at least one test.

Next, for each technique, we created a multiple linear regression model to predict the quality of the patches based on the test suite coverage and size. Figure 4.12 shows, for each technique, the results of the regression model. All three regression models are strongly statistically significant ($p < 0.001$) though with low R^2 values. Test suite size was a statistically significant predictor for patch quality for all three techniques. This suggests that larger test suites lead to higher-quality patches; however, with an extremely small effect size. Test suite coverage was a less clear predictor: for TrpAutoRepair, the association was not statistically significant ($0.1 < p < 1$), and the relation was positive for TrpAutoRepair, but negative for PAR and GenProg. We further detail each technique's regression results next.

technique	model quality		test suite	p
	p	R^2		
GenProg	7.2×10^{-13}	0.013	size	6.7×10^{-13}
			coverage	8.5×10^{-4}
PAR	5.2×10^{-12}	0.035	size	4.2×10^{-5}
			coverage	7.6×10^{-11}
TrpAutoRepair	6.9×10^{-5}	0.0057	size	1.6×10^{-5}
			coverage	0.96

Figure 4.12: Multiple linear regression relating coverage and size to patch quality. A multiple linear regression reports that test suite size and test suite coverage are strongly significantly associated with patch quality ($p < 0.01$) except for coverage for TrpAutoRepair).

The regression function for GenProg's patch quality (on a 0–100 scale) is:

$$\text{genprog\_patch\_quality} = 94.82 - 0.02(\textit{coverage}) + 0.02(\textit{size})$$

where coverage is $100\times$ the fraction of code in the defective code version covered by the test suite, and size is the normalized number of tests in the test suite used to generate the patch. Thus, the quality of the patch produced by GenProg decreases by 0.02% for each 1% increase in the test suite coverage and increases by 0.02% for each additional test in the test suite. While both associations of test suite coverage and size with the patch quality were statistically significant ($p < 0.001$), the magnitude is extremely small. We conclude that test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for GenProg.

For PAR, patch quality is described by the function:

$$\text{par\_patch\_quality} = 91.18 - 0.10(\text{coverage}) + 0.03(\text{size})$$

Thus, the quality of the patch produced by PAR decreases by 0.10% for each 1% increase in the test suite coverage and increases by 0.03% for each additional test in the test suite. Again, while both associations of test suite coverage and test suite size with patch quality are strongly statistically significant ($p < 0.001$), the magnitude is extremely small. We conclude that both test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for PAR.

For TrpAutoRepair, the quality of the patch is equal to:

$$\text{trpautorepair\_patch\_quality} = 95.80 + 0.0003(\text{coverage}) + 0.006(\text{size})$$

The equation implies that the quality of the patch produced by TrpAutoRepair increases by 0.0003% for 1% increase in the test suite coverage and increases by 0.006% for each additional test in test suite. The association of test suite size with patch quality is strongly statistically significant ($p < 0.001$), but that is not the case for test suite coverage ($0.1 < p < 1$). The magnitude of the

association is extremely small. We conclude that test suite size is a significant predictor of patch quality, but the magnitude of the effect is extremely small, for TrpAutoRepair.

Answer to Research Question 3: We conclude that both test suite size and test suite coverage have extremely small but statistically significant correlations with patch quality (positive for test suite size in all techniques and negative for test suite coverage in two techniques) produced using automatic program repair techniques.

4.2.3 Defect Severity

Research Question 4: How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

The intuition behind this research question is that if a defect is triggered by a large number of failing test cases, the APR approach will have more information (in the form of restrictions) that it has to satisfy when creating a plausible patch, therefore the quality of such patches should be higher than patches generated using a lower number of failing test cases which have to satisfy a smaller number the restrictions described by a the test cases.

Methodology:

To measure the effect of the number of failing tests in the test suite used to guide repair, we selected those defects that had at least 5 failing tests in the developer-written test suite and for which we are able to create high-quality evaluation test suite (recall Section 3.3). There were only 5 such defects in the 68-defect subset of Defects4J.

For each of the five defects, we created 21 test suites subsets. We did this by first computing five evenly distributed target sizes s: $\frac{1}{5}f$, $\frac{2}{5}f$, $\frac{3}{5}f$, $\frac{4}{5}f$, and f, where f is the number of failing tests in the developer-written test suite (rounding to the nearest integer). Notice that there is a unique superset of failing test cases, unlike Section 4.2.2 where there are potentially several subsets to achieve maximum coverage. Therefore in this Section we create 21 test suite subsets, different from the 25 subsets in Section 4.2.2. For each s (except $s = f$), we created 5 test suite subsets by including every passing test from the developer-written test suite, and uniformly randomly sampling, without replacement, s of the failing tests. This created 20 test suite subsets. We also included the entire developer test suite as a representative of the $s = f$ target, for a total of 21 test suite subsets. We then used the three automated repair techniques to attempt to patch the defects using each of the test suite subsets, following the methodology described in Section 4.1. Our methodology controls for the number of passing tests, unlike the prior study [189].

Finally, we used Pearson correlation coefficient to assess the linear relationship between patch quality and the number of failing tests in the test suite used to guide repair.

Results:

Figure 4.13 shows the frequency distribution of failing tests across the 68 defects for which at least one of the three techniques produced at least one patch, and for which we were able to create a high-quality evaluation test suite. Of these 68 defects, only 5 defects, Chart 22, Chart 26, Closure 26, Closure 86, and Time 3, have at least five failing tests.

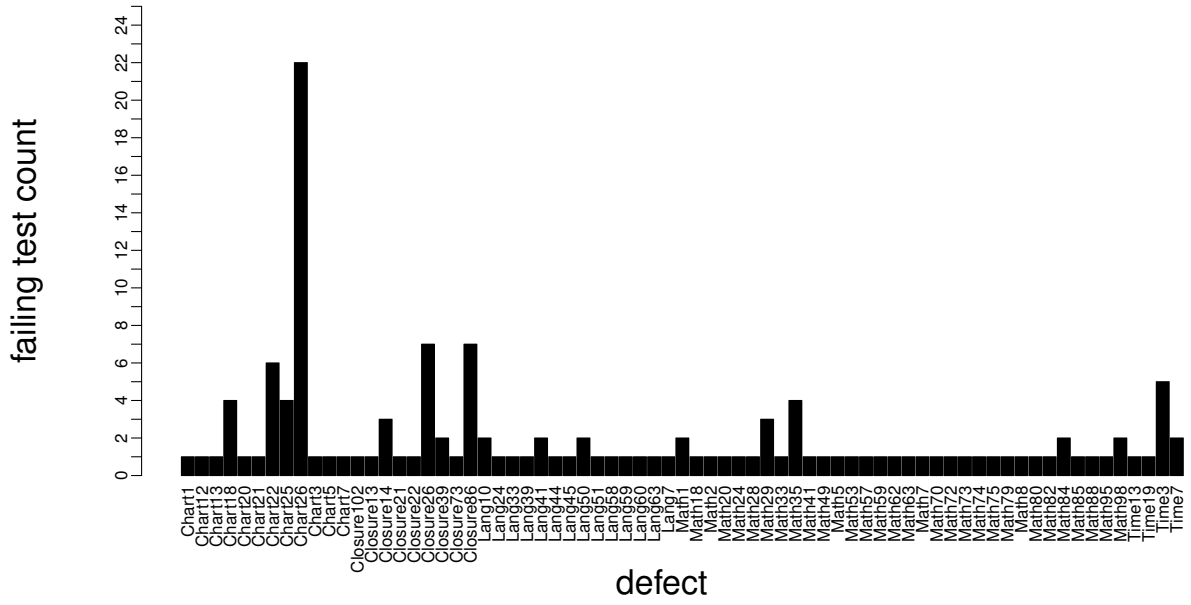


Figure 4.13: Defect severity. The distribution of the number of failing tests in the 68 defects for which at least one repair technique produces at least one patch and has a high-quality evaluation test suite.

Figure 4.14 shows, for each technique, the quality of the patches produced, as a function of the fraction of the failing tests in the test suite used to guide repair. For GenProg and TrpAutoRepair, we observe statistically significant ($p < 0.05$) positive correlations (GenProg: $r = 0.18$, $p = 0.006$; TrpAutoRepair: $r = 0.19$ $p = 0.008$) between patch quality and the number of failing tests in the test suite. PAR did not produce any patches for any of the 5 defects considered for this analysis.

Answer to Research Question 4: We conclude that the number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the patches produced using automatic program repair techniques and that this finding depends on the fault localization strategy used by the repair techniques.

4.2.4 Test Suite Provenance

Research Question 5: How does the test suite provenance (whether written by developers or generated automatically) influence patch quality?

Prior work has suggested that using automatic test generation might improve program repair quality by increasing the coverage of the test suite used to produce the repair [189, 215, 223]. Augmenting a developer-written test suite with automatically-generated tests requires an oracle that specifies the expected test outputs. Several approaches have been proposed regarding the usage of different program specifications as possible oracles, which include other implementations

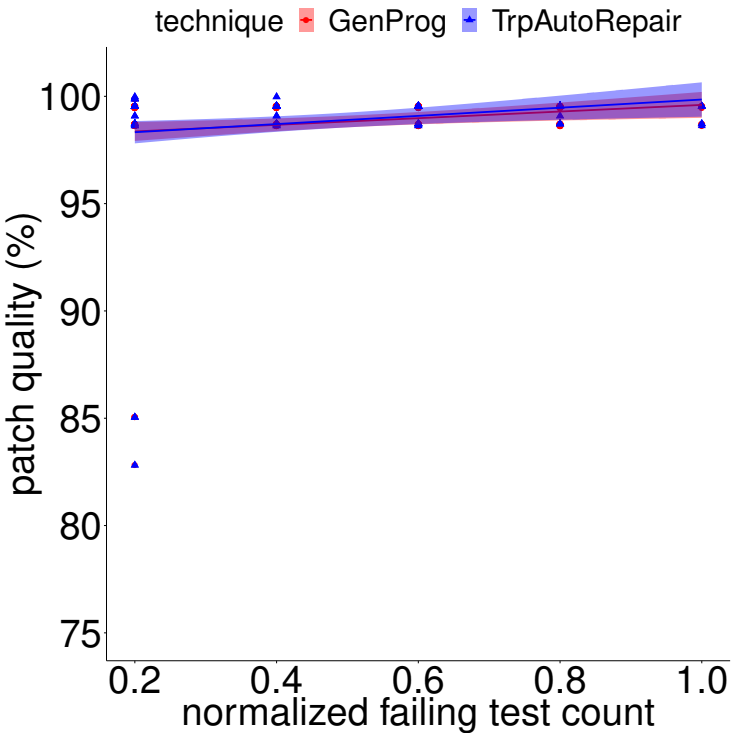


Figure 4.14: Defect severity. Linear regression between patch quality and the number of failing tests and Pearson's correlation show statistically significant positive correlations for GenProg and TrpAutoRepair.

of the same specification [138], or extracted from comments or natural language specifications (e.g., Swami [149], Toradacu [72], Jdoctor [27], or @tComment [196]).

However, a previous study [189] found that even when a perfect oracle exists, using automatically-generated tests for program repair resulted in much lower quality patches than using developer-written tests (about 50% vs. about 80% quality) on small, student-written programs. Thus, in this research question we evaluate the effectiveness of using tests generated using EvoSuite as described in Section 3.3 to produce patches using generate-and-validate repair when used in real-world defects.

Methodology:

In this experiment, we compared the patches generated using developer-written test suites from Section 4.1 to patches generated using the EvoSuite-generated test suites. A technical challenge in executing repair techniques using EvoSuite-generated tests is a potential incompatibility between the bytecode instrumentation of EvoSuite-generated tests with the bytecode instrumentation done by code-coverage-measuring tools employed by repair techniques for fault localization. JaRFly uses JaCoCo [82] for fault localization and resolves instrumentation conflicts by updating the runtime settings of EvoSuite-generated tests (following official EvoSuite documentation[2]). The EvoSuite-generated tests are compatible with JaCoCo, Cobertura [41], Clover [15], and PIT [44] code coverage tools. To create the EvoSuite-generated tests we used the developer-written patches

[2] `http://www.evosuite.org/documentation/measuring-code-coverage/`

as the oracle of expected behavior.

To control for the differences in the defects, properly measuring the association between test suite provenance and patch quality should be done using defects that can be patched using both kinds of test suites. If the set of defects patched using developer-written test suites differs from the set of defects patched using the automatically-generated test suites (as was the case in the earlier study [189]), then the defects can be a confounding factor in the experiment. For example, it is possible that more of the defects patched using one of the types of test suites are easier to produce high-quality patches for, unfairly biasing the results.

We thus started with the 68 defects for which at least one of the three repair techniques (GenProg, PAR, and TrpAutoRepair) was able to produce a patch when using the developer-written test suites to guide repair, and first discarded those defects for which the EvoSuite-generated test suites did not evidence the defect. To evidence the defect, at least one test in the test suite has to fail on the defective code version. (By definition, all automatically-generated tests pass on the developer-patched version, since that version is the oracle for those tests.)

For 31 out of the 68 defects, automatically-generated test suites did not evidence the defect. This left 37 defects (5 Chart, 4 Closure, 11 Lang, 16 Math, and 1 Time). We next executed each of the three repair techniques on each of the 37 defects using the EvoSuite-generated test suites, using the methodology from Section 4.1, thus executing $37 \times 20 = 740$ repair attempts per technique. Note that comparing repair techniques' behavior with different test suites on these 37 defects is unfair because one of the criteria they satisfied to be selected is that at least one repair technique produced at least one patch for the defect using the developer-written test suite. Thus, for each technique, we identified the set of defects that were patched both using developer-written and using automatically-generated test suites. We call these the *in-common* populations. Note that these populations are, potentially, different for each technique.

To compare the quality of the patches on the in-common patch populations, we use the nonparametric Mann-Whitney U test. We choose this test because the two populations may not be from a normal distribution. This test measures the likelihood that the two populations came from the same underlying distribution. We compute Cliff's delta's δ estimate to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95% confidence interval (CI) of the δ estimate.

Results:

Figures 4.15, 4.16, and 4.17 summarize our results. Figure 4.15 reports data for the 37 defects for which both test suites evidence the defect. As expected, because of the aforementioned bias in the selection of the 37 defects, using EvoSuite-generated test suites produced fewer patches and patches for fewer defects than using developer-written test suites. Using developer-written test suites produced a patch on between 10.1% and 21.4% executions, while using EvoSuite-generated test suites produced a patch on between 2.3% and 13.9% of the executions. Using developer-written test suites produced a patch for between 54.1% and 81.1% of the defects, while using EvoSuite-generated test suites produced a patch for between 5.4% and 45.9% of the defects.

In addition to the bias in defect selection, another possible reason that EvoSuite-generated test suites resulted in fewer patches could be differences in the test suites. Figure 4.16 shows the distributions of the number of failing (defect-evidencing) tests across the 37 defects for the two types of test suites. EvoSuite-generated test suites typically had more failing tests,

technique	test suite	generated patches	defects patched	patch quality minimum	mean	median	maximum	100%-quality patches
GenProg	developer	158 (21.4%)	29 (78.4%)	77.4%	94.9%	98.0%	100.0%	17.8%
	EvoSuite	98 (13.2%)	14 (37.8%)	6.3%	65.3%	54.3%	100.0%	8.2%
PAR	developer	75 (10.1%)	20 (54.1%)	98.1%	98.4%	98.1%	99.7%	0.0%
	EvoSuite	17 (2.3%)	2 (5.4%)	97.2%	99.6%	99.9%	100.0%	41.2%
TrpAutoRepair	developer	128 (17.3%)	30 (81.1%)	77.4%	96.8%	98.1%	100.0%	24.6%
	EvoSuite	103 (13.9%)	17 (45.9%)	6.3%	65.2%	54.3%	100.0%	10.4%

Figure 4.15: Patching results for the 37 Defects4J defects whose developer-written and EvoSuite-generated test suites have at least one failing test each. Using EvoSuite-generated test suites, automated program repair techniques were able to produce patches for 37 of the the 68 defects.

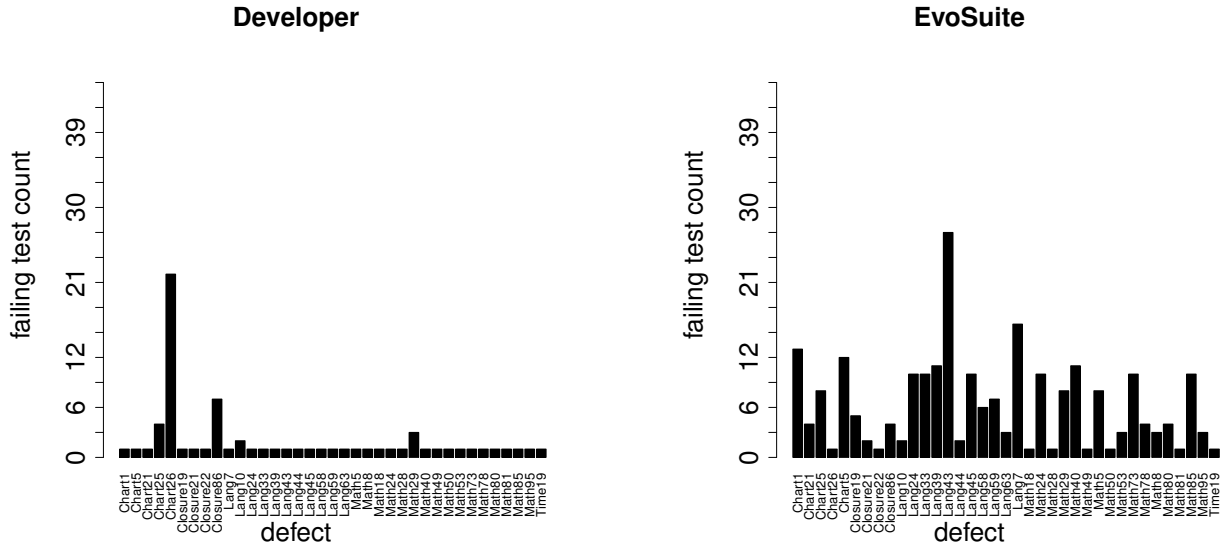


Figure 4.16: Distributions of failing tests in the 37 Defects4J defects' test suites. The EvoSuite-generated test suites typically have more failing tests than the developer-written ones.

perhaps contributing to it being more difficult to produce patches when using those test suites. Prior work has shown that having a larger number of failing tests correlated with lower patch production [150, 189].

We compared the quality of the patches produced using the two types of test suites on the in-common populations. Figure 4.17 shows that for GenProg and TrpAutoRepair, the mean and median quality of the patches produced using the developer-written test suites are higher than of those produced using EvoSuite-generated test suites. These differences are statistically significant (Mann-Whitney U test, $p = 1.3 \times 10^{-11}$ for GenProg, and $p = 5.8 \times 10^{-11}$ for TrpAutoRepair). The δ estimate computed using Cliff's delta shows a large effect size for the median patch quality of the patches produced using EvoSuite-generated test suites being lower for GenProg and TrpAutoRepair. The 95% CI does not spans 0 for both techniques, indicating that, with 95% probability, the two populations are likely to have different distributions.

For GenProg, this comparison is on the 12 in-common defects (Chart 5, Closure 22, Lang 43,

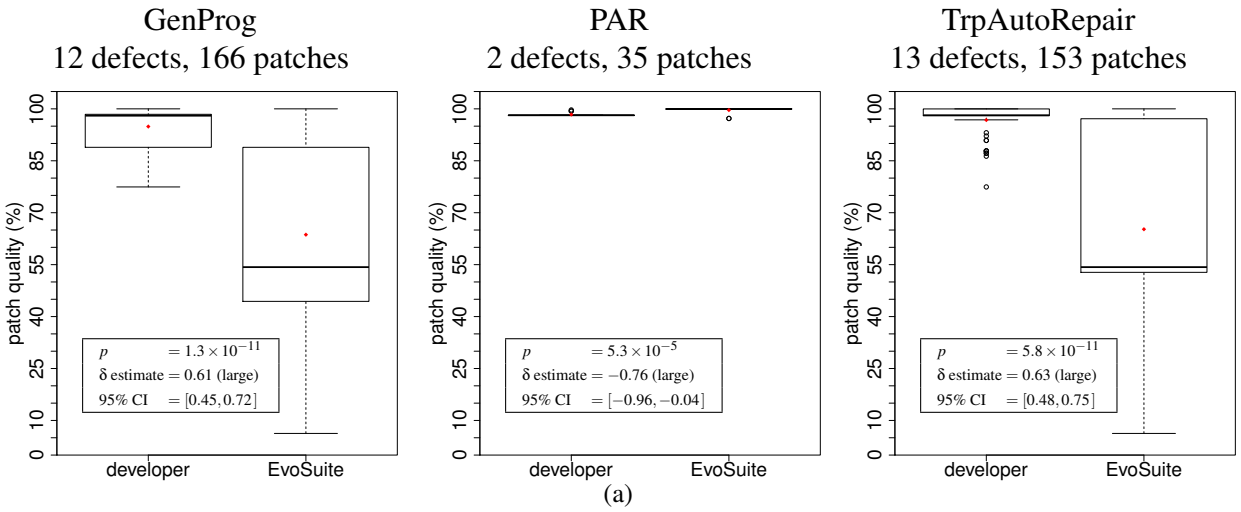


Figure 4.17: Test suite provenance. Patch quality comparison on the in-common (patched using both types of test suites) defect populations. The box-and-whisker plots compare patch quality on the in-common defect populations, showing the maximum, top quartile, median, bottom quartile, and minimum values, with the mean as a red diamond. The quality of patches produced by GenProg and TrpAutoRepair using the EvoSuite-generated test suites is statistically significantly (Mann-Whitney U test) lower that those produced using developer-written test suites. For PAR, the effect is reversed.

Math 24, Math 40, Math 49, Math 50, Math 53, Math 73, Math 80, Math 81, and Time 19). On these defects, GenProg produced 73 patches using developer-written test suites and 93 patches using EvoSuite-generated test suites (166 patches total). For TrpAutoRepair, this comparison is on the 13 in-common defects (Chart 5, Closure 22, Closure 86, Lang 43, Lang 45, Math 24, Math 40, Math 49, Math 50, Math 73, Math 80, Math 81, and Time 19). On these defects, TrpAutoRepair produced 57 patches using developer-written test suites and 96 patches using EvoSuite-generated test suites (153 patches total).

Because the results for GenProg and TrpAutoRepair are derived from 12 and 13 defects, respectively, there is hope that these results will generalize to other defects. The same cannot be said for PAR. PAR produced patches using both types of test suites for only 2 out of the 37 defects (Closure 22 and Math 50). Figure 4.17 shows that the mean and median quality of the patches produced using the developer-written test suites are lower than those produced using EvoSuite-generated test suites. This result is statistically significant because PAR produced 18 patches using developer-written test suites and 17 patches using EvoSuite-generated test suites, with $p = 5.3 \times 10^{-5}$ and the 95% CI interval does not span 0. However, while significant for these 2 defects, we cannot claim (nor do we believe that) this result generalizes to all defects from this 2-defect sample.

Our finding is consistent with the earlier finding [189] that provenance has a significant effect on repair quality, and that for GenProg and TrpAutoRepair, developer-written test suites lead to higher quality patches. Surprisingly, the finding is opposite for PAR (which was not part of the earlier study), with automatically-generated tests leading to higher-quality patches. Our study

improves on the earlier work in many ways: We control for the defects in the two populations being compared, we use real-world defects, and we use a state-of-the-art test suite generator with a rigorous test suite generation methodology. The earlier study used a different generator (KLEE [33]) and aimed to achieve 100% code coverage on a reference implementation, but the generated test suites were small.

Answer to Research Question 5: We conclude that test suite provenance has a significant effect on repair quality, though the effect may differ for different techniques. For GenProg and TrpAutoRepair, patches created using automatically-generated tests had lower quality than those created using developer-written test suites. For a smaller, perhaps non-representative number of defects, PAR-generated patches showed the opposite effect.

The results obtained show that, as hypothesized, enhancing key characteristics of the guiding test suite can lead to an improvement of the produced patches. Concretely, this chapter shows that guiding test suite size and provenance are strong indicators for produced patch quality followed by coverage, and therefore enhancing these components in automatic program repair leads to higher quality plausible patches.

We think that test suite size by itself might be a proxy for an underlying not-analyzed quality attribute, therefore further inspection into test suite attributes which correlate with test suite size and their corresponding patch quality might be needed in the future to expand the analysis on patch quality.

This Chapter helps us understand that tool generated patches often overfit to the test suite used, often breaking more functionality than they repair. Therefore, we analyze the role of guiding test suites and the corresponding quality of the patches generated when using said test suites in the automatic program repair process. We analyzed quality attributes of guiding test suites that can be optimized to maximize the quality of the patches generated by APR and , therefore, show how patch quality in the APR process can be improved by correctly choosing and optimizing these quality attributes such as test suite coverage, size and provenance.

Chapter 5

Analyzing Developer Software Changes to Inform APR Selection Mechanisms

A key component in the automatic program repair process is the selection mechanism APR approaches use to choose which edits will be applied to faulty locations when creating patch candidates. This selection is particularly difficult because the search space of possible edits that can be applied to each location is infinite, an infinite number of changes can be applied to a program creating a transformation of the original version that after evaluation becomes a plausible patch.

The goal of automatic program repair techniques is to modify a program P containing at least one error and create P': a transformed version of P where the correct functionality of P is maintained but the incorrect functionality of P is modified to no longer manifest the error(s) in P. To create P' from the original P it is necessary to apply certain code changes to P. We broadly refer to these change types as *mutation operators*.

There is a broad diversity of such operators used in automatic program repair, including deleting or inserting statements [113], applying templates [100], transformation schemas [127, 130], or semantically-inferred code [140, 153, 220]. Given a potentially-faulty location (typically identified using off-the-shelf fault localization, e.g., Tarantula [91]), these approaches then use heuristics or heuristically-informed probability distributions to select between mutation operators to construct candidate patches. These heuristics are mostly based on general approximations of reasonable behavior that have not been carefully calibrated. Therefore, even with these efforts, the search space for possible edits remains vast and many of the patches produced using these techniques are of low quality.

For example, applying a single line change that modifies the status of the program (e.g. *"variable++;"*) in a continuous manner can be repeated infinitely creating a different program version every time (*variable* would hold a different value in each transformation and therefore the program behavior would likely be different in each version).

Human developers use certain mutation operators much more frequently than others when

fixing errors in software (e.g., deleting a line is much more common than creating a new class). Therefore, our intuition is that since developers have a wide understanding of what edits and statements need to be selected to fix errors, analyzing developer behavior to fine-tune the selection decisions in APR would increase the produced patches' quality. We can thus use that knowledge to drive our search.

In this chapter, we study and then simulate the behavior of human developers to create patches. Our key intuition is that our approach can navigate the search space guided by human-learned mutation operator selections making it therefore more likely to produce high-quality patches. Similar ideas have been used in the past to create more human-acceptable mutation operators [100] and to inform patch *ranking* (rather than construction) [130, 219].

We mine bug fixing commits from the 500 most popular GitHub Java projects to model the selection probability of the possible mutation operators based on empirical data that describes how human programmers fix their code. We thus compare and validate a superset of mutation operators in use in a number of state-of-the-art approaches [100, 113, 130, 210].

We then use this model to guide a repair approach that chooses from the set of possible operators based on these human-learned probabilities. As a result, our work goes beyond prior work that leverages human bug fixes in a program repair context [100, 130, 219] by generalizing to a broader set of mutation operators, and using a developer-learned model when patch candidates are created.

We evaluate the predictive power of our mined model in terms of its accuracy in predicting the operators used in real-world bug fixes. We demonstrate the quality improvement of patches generated by this approach with a full set of mutation operators on a subset of real-world single-line defects [92] in comparison to several previous state-of-the-art techniques.

In the following Sections of this chapter we analyze the behavior of developers when fixing bugs by inspecting the types of statements they modify and the edits they perform to the source code (Section 5.1). We then mine a corpus of popular open source projects and create an empirical probabilistic model of the edits developers use (Section 5.2). Finally we create an APR approach which uses this probabilistic model to select which transformations to apply when creating patch candidates (Section 5.3).

In this chapter we answer the following research questions [1]:

RQ6 Do real-world developers edit certain statement kinds more frequently than others in the bug-fixing process?

Answer: Yes, Expression statements are added in 25.7% of the studied cases while Type Declaration statements only in 0.2%. The most commonly deleted statements are Expression statements (13.6% of the cases) while Type Declaration statement only a 0.2%.

RQ7 What is the distribution of edit operations applied by human developers when repairing errors in real world projects?

Answer: The distribution of mutation operators is described in Figure 5.3. The most common mutation operator is "Append" and the least common operator is "Off by One".

[1] Portions of the work described in this Chapter have been published in the International Conference on Software Analysis, Evolution and Reengineering [192] and in the International Conference on Mining Software Repositories, Challenge Track [191, 194]

RQ8 How does a human-informed automatic program repair tool compare to the state-of-the-art in APR?

Answer: An APR technique using a mutation operator selection mechanism informed by developer behavior is able to generate less but higher-quality patches as compared to the other APR techniques.

RQ9 What are the most common multi-edit modification rules in practice?

Answer: The most common multi-edit rules are described in Table 5.5, the most common consequent is "Append".

5.1 Analysis of Developer Changes in Java Projects

> **Research Question 6:** Do real-world developers edit certain statement kinds more frequently than others in the bug-fixing process?

For this research question we are interested in understanding if there is an actual difference between how human developers modify source code and how APR techniques modify source code, and how substantial is this difference with the goal of later creating an APR goal that approximates human code changes.

Methodology:

To answer this question, we use the Boa framework [59, 85]. Boa is a domain-specific language and infrastructure that eases mining software repositories. Boa's infrastructure leverages distributed computing techniques to execute queries against hundreds of thousands of software projects [58] Boa provides the infrastructure to query 4,590,679 bug-fixing commits from a database of 554,864 Java projects.

Bug-fixing commits are identified by the Boa framework using the *isfixingrevision* function which uses a list of regular expressions to match against the revision's log [58]. The Java language specification classifies statements into statement kinds (e.g., For Loop, While Loop, Variable Declaration, Assignment, etc.). Since our intuition is that APR can benefit from mimicking the edit behavior of developers, we start by analyzing if and how developers apply common APR edits. Some automatic repair approaches seek generality by using higher-granularity mutation operators such as statement-level addition, deletion and replacement. To support the generation of high-quality patches, we analyze how developers mutate source code to fix bugs at this granularity level.

Because direct diffs are difficult to identify on this dataset, we heuristically approximate the extent to which one statement type appears to be "replaced" by another. For each modified file, we count the number of appearances of each statement type in the file pre- and post-commit. We then compare the results to see how many of each statement type was removed, and how many inserted, to roughly characterize the types of replacement that happen at a per-file level. Note that this analysis doesn't distinguish the replacement of the same statement kind, since we are counting the amount of appearances of each statement kind. We follow a similar approach to

	Assert	Break	Continue	Do	For	If	Label	Return	Case	Switch	Synch	Throw	Try	TypeDecl	While
Assert	-	7.48	3.76	0.53	8.30	23.05	0.31	20.04	4.90	4.62	1.30	13.50	7.23	0.03	4.95
Break	1.00	-	4.08	0.60	9.93	26.03	0.13	25.39	2.48	1.57	1.79	8.39	11.73	0.10	6.77
Continue	1.74	9.42	-	1.28	11.39	18.25	0.35	22.60	3.80	2.85	2.17	8.98	9.42	0.11	7.63
Do	0.81	5.26	6.60	-	9.44	14.21	0.18	15.86	3.73	1.67	1.97	5.88	6.39	0.03	27.98
For	0.86	6.28	3.19	0.79	-	22.89	0.09	21.08	5.01	3.34	1.87	10.01	10.71	0.08	13.79
If	1.64	8.43	2.87	0.60	13.49	-	0.24	26.46	7.45	4.80	2.85	9.89	15.11	0.08	6.11
Label	1.30	8.33	7.86	1.11	5.18	22.85	-	15.17	3.05	2.04	14.62	10.45	4.16	0.09	3.79
Return	1.13	9.41	3.11	0.49	13.33	27.24	0.24	-	5.59	3.65	2.55	14.91	12.61	0.12	5.61
Case	0.78	2.84	2.84	0.39	10.27	31.79	0.16	22.40	-	0.46	2.07	7.37	11.69	0.08	6.87
Switch	1.14	2.72	3.80	0.55	11.07	34.14	0.13	21.86	0.75	-	1.53	8.65	9.02	0.05	4.58
Synch	0.80	6.57	2.28	0.43	10.21	24.18	0.05	19.77	6.35	2.07	-	9.16	12.16	0.04	5.93
Throw	2.11	6.57	2.58	0.48	11.87	18.84	0.17	32.28	4.64	3.30	2.74	-	10.08	0.07	4.27
Try	0.71	7.41	3.02	0.66	11.73	27.75	0.11	23.24	5.63	2.65	2.58	8.99	-	0.09	5.42
TypeDecl	0.00	4.51	7.52	1.00	10.28	21.05	0.50	17.79	6.02	1.75	2.01	9.27	11.53	-	6.77
While	0.72	8.02	3.82	1.96	23.16	19.78	0.12	16.48	6.56	3.09	1.64	6.81	7.80	0.04	-

Table 5.1: This Table describes the likelihood for developers in the analyzed corpus to replace a statement type (row) by a statement of another type (column). The diagonal is empty given that this replacement chart is based on an incidence count of statement, therefore it does not account for statements that replace other statements of its same kind.

approximately count deletions and insertions. For each bug fixing revision r and each statement kind k, we compare the count of statements of kind k in revision r and $r-1$.

Results:

Table 5.1 shows the replacement likelihood for our dataset (each cell shows the percent of the time that the statement in the *row* was replaced by a statement of the type in the *column*). For example, the corresponding to the `Do` row (row 5) and `Assert` column (column 2) shows 0.81, indicating that `Do` statements were replaced by `Assert` statements 0.81% of the times.

Additional analysis (raw numbers not shown) show that the most common replacement replaces `Return` statements with `If` statements (in 30,489 files). The second most common replacement replaces an `If` statement with a `Return` (28,536 incidences). By contrast, the least common replacement was an `Assert` statement replacing a `TypeDecl`, which we did not observe. The second least common replacements were replacing `Do` statements or `Labels` for a `TypeDecl`; we observed these once each.

The most common *replacer* statement (the statement kind that most commonly replaces others) is the `If` statement (101,366 appearances). The least common *replacer* is the c (447 appearances). This makes the ratio of most to least commonly used replacer to be 227:1. In other words, per each time that a `TypeDecl` statement was used by a developer to replace another program statement, there were 227 occasions where an `If` statement replaced another program statement. The most common *replacee* was the `Return` (111,938 appearances); the least common *replacee* was again the `TypeDecl` (399 appearances). This makes the ratio of most used replacee to least used replacee to be 281:1.

From Figure 5.1, we can see that `expression`, `If`, `Return`, `for` and `Try` statements are both added and deleted most often as compared to the other statement times. These findings indicate that most bugs were fixed by changing control flow.

An important study that complements ours is the repair model approach proposed by Martinez and Monperrus [136], which proposes a probability distribution suggesting when to apply which

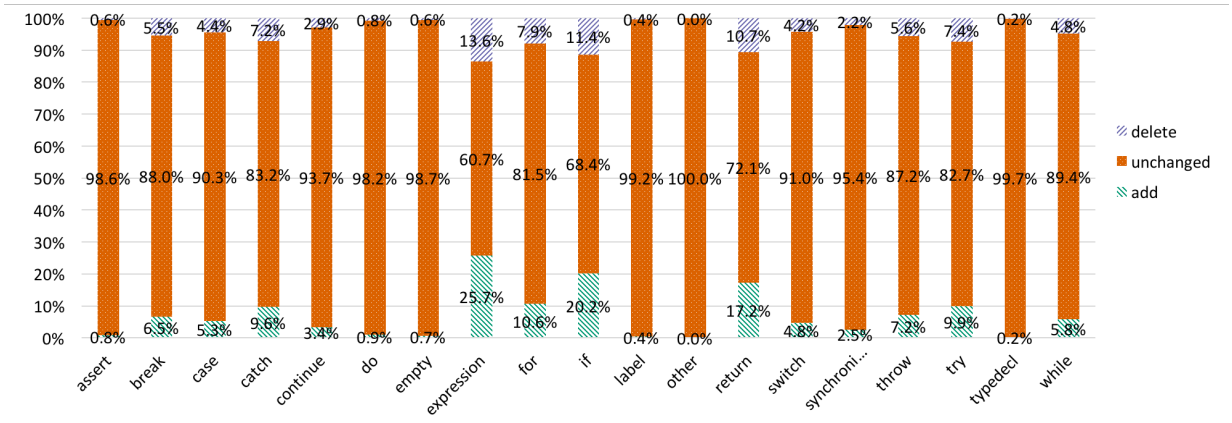


Figure 5.1: This Figure shows, per each bug-fixing revision, the percentage of statements that get added, deleted and unmodified per each statement kind. Expression is the statement kind most added in the analyzed bug-fixing revisions and also the most deleted.

kind of edit. Although their approach can trace more fine-grain AST-level changes, our results are consistent with their findings. For example, their empirical analysis [136] shows that method invocations, if statements, and variable declarations are added/deleted/updated most often, which is also illustrated in Figure 5.1 (Boa groups method invocations and variable declarations into the `Expression` category). Our study complements there at a much larger scale (we study 380,125 repositories with 23,229,406 revisions as compared to the 14 repositories in the prior work).

Similarly, we analyzed deletions and insertions of program statements in bug fixing commits. The most commonly added statement kind is *Expression Statement*, added in 25.7% of the studied cases, followed by *If Statement* (17.2%). The least added was the *Type Declaration*, (0.2%). The most commonly deleted statement kinds are *Expression Statement*, deleted 13.6% of the cases studied; and the least deleted statement kind is the *Type Declaration* (0.2%).

Answer to Research Question 6: Yes, Expression statements are added in 25.7% of the studied cases while Type Declaration statements only in 0.2%. The most commonly deleted statements are Expression statements (13.6% of the cases) while Type Declaration statement only a 0.2%.

Our analysis shows that human developers indeed use some program edits more frequently than others in the bug-fixing process. We focused on the analysis of high level coarse-grain mutation operators (replacement, insertion and deletion of program statements). The analyzed data shows that there is a considerable gap between the most commonly used statement for insertions (25.7%) and least used (0.2%), similarly the most commonly deleted statement (Expression statement 12.6%) to the least common (Type Declaration statement 0.2% of analyzed cases). Furthermore, an analysis of replacements shows that the ratio between most common to least common replacer is 227:1, and most common to least common replacee is 281:1.

5.2 Corpus Mining from Popular GitHub Projects

In Section 5.1 we validated our intuition that when human developers fix program errors, some program statements are used more often than others, and therefore the distribution of edits necessary to fix bugs is not equally distributed. This distribution varies broadly between edits, including how often statements get added, deleted and replaced.

In this next research question we performed an analysis to understand the distribution of *mutation operators* (types of edits) used by developers when fixing bugs with the goal of creating a repair approach that using this distribution can build repairs in a similar way to how humans create repairs. Therefore our next research question is:

> **Research Question 7:** What is the distribution of edit operations applied by human developers when repairing errors in real world projects?

In Section 5.1 we used a powerful publicly-available code mining framework [59, 85] to understand how human developers use different types of edits to patch errors in source code. Given the way in which this framework outlays their program representation, we are not able to track each specific program statement, therefore we counted the aggregate number of each statement type between the versions of code before the developer fixed the code and the version after and create an analysis based on this difference. This approach therefore allows for some inaccuracies in our calculation.

In this next Section, we describe how we create our own mining approach to minimize these inaccuracies by using code differencing tools [62, 71] to match statements in the versions of code before the fix took place and after the fix took place, and by focusing on mutation operators used by APR approaches instead of statement types. We mine a model of human bug-fixing edits from a large set of popular Java projects.

The intuition is to use this model to apply human knowledge to the automatic program repair process, creating patches inspired by what human developers do; the model is used explicitly in the patch creation step of a generate-and-validate repair process. To do this, we select a corpus of popular GitHub projects and identify their most recent bug fixing commits. We identify mutation operator and replacement incidence in this dataset to construct a two level probabilistic model used in a novel repair technique).

Methodology:

The first step towards analyzing how often human developers use each type of program edit is to understand what types of edits (mutation operators) do APR approaches use and how do these match to the changes performed by developers. We thus categorize mutation operators used from a cross section of state-of-the-art approaches into two groups:

5.2.1 Categorizing Mutation Operators

One family of repair approaches [113, 170, 210] creates candidate patches by applying coarse-grained mutation operators (e.g. *append*, *delete*, or *replace*) at the statement level. These prior techniques historically target the C programming language, where a statement is a grammar nonterminal corresponding intuitively to blocks, simple statements that terminate with a semicolon,

or compound statements corresponding to control flow or loops. In Java, statements conceptually map to similar program elements, e.g. blocks, while loops, or single-line method calls. In these approaches, the statements being appended or replaced typically come from within the project being modified. This is grounded in the notion that source code has a high level of redundancy [81].

Another family of approaches [100, 127, 130] instantiates predetermined templates, more complex than those in the first family, at applicable code locations and typically at a finer level of modifications.

PAR is the product of a study of a large number of human created patches, from which human annotators abstracted a set of different templates to cover the most commonly-used changes in bug-fixing practice. These considered templates are detailed in the top section of Figure 5.2. In the interest of completeness, we also include six extra templates mentioned on the PAR website.[2] These extra templates provide new mutation operators drawn from human edits, that help us compare to and generalize the other approaches; they are shown in the middle segment of Figure 5.2. SPR and Prophet use a set of transformation schemas, shown in the bottom section of Figure 5.2.

PAR fix templates	
Null Checker	Parameter Adder and Remover
Parameter Replacer	Expression Adder and Remover
Method Replacer	Collection Size Checker
Expression Replacer	Range Checker
Object Initializer	Class Cast Checker
PAR "extra" templates	
Caster Mutator	Lower Bound Setter
Castee Mutator	Upper Bound Setter
Sequence Exchanger	Off-by-one Mutator
SPR transformation schema	
Condition Refinement	Insert Initialization
Copy and Replace	Condition Control Flow Introduction
Value Replacement	Condition Introduction

Figure 5.2: (Top) PAR fix templates. (Middle) PAR "extra" templates. (Bottom) SPR transformation schemas. We use the templates in the top and middle portions of this Table as representative of the class of Template-based mutations.

The SPR/Prophet transformation schemas can be mapped to a subset of the PAR templates. For example, *Condition Introduction* can be seen as a superset of *Range Checker*, *Collection Size Checker*, *Class Cast Checker*, and *Null Checker*. *Condition Refinement* includes *Expression Adder and Remover*. *Insert Initialization* can be generalized from *Object Initializer*, *Upper Bound Setter* and *Lower Bound Setter*; *Conditional Control Flow Introduction* can be seen as a subset of

[2] `https://sites.google.com/site/autofixhkust/home/fix-templates`

Sequence Exchanger; *Value Replacement* can be seen as a superset of *Method Replacer*, *Parameter Replacer*, *Castee Mutator* and *Expression Changer*; and *Copy and Replace* can be matched to *Expression Adder*. These operators similarly generalize those used in semantics-based approaches, which replace expressions used either in conditions or on the right-hand-side of assignments (the operators are the same; the difference lies in how the fix code is selected/constructed).

The templates used in the program modification tool Kali [172] also correspond to subsets of certain PAR templates or their extensions. For example, *Redirect Branch* can be seen as a subset of *Expression Changer*, and *Insert Return* and *Remove Statement* are subsets of *Expression Adder and Remover* accordingly. Similarly, many other operators from the field of mutation testing [154], as used in APR [51, 219] can be seen as subsets or extensions of the PAR templates.

To summarize, these approaches have significant similarities between them. We use the PAR templates to represent this category because PAR (1) broadly includes the other techniques' mutation operators, (2) provides a concrete description of how the code is changed, enabling replication, and (3) explicitly targets Java (SPR, Prophet and Kali target C), reducing the extent to which we must apply subjective judgment to re-implement and use in our context.

5.2.2 Building Probabilistic Model

Given the categorization of Section 5.2.1 we can now mine a corpus of developer changes to understand how these human edits match the APR mutation operations targeted in this study.

We first cloned the 500 most-starred Java projects on GitHub and identified the most recent 100 bug fixing commits per each project. If the project had fewer than 100 bug fixing commits, we analyzed as many as found. Identifying such commits is a difficult problem [26], we followed previous approaches [45, 99, 180] to identify bug-fixing commit by applying a regular expression to each commit message that looks for words such as *"fix", "bug", "issue", "problem",* etc.

We further only include commits that exclusively modify Java source code, since we focus on such bugs. We restrict attention to commits that modify a maximum of three files to exclude big merges, and because large commits are more likely to include changes unrelated to a bug fix [80, 95].

For each considered commit, we refer to the code before the fix as the *"before-fix"* version and the code after as the *"after-fix"* version. We seek to identify the changes performed between the before- and after-fix versions, match them to our considered mutation operators, and count how often each operator is used in the edits in our corpus. We used Gumtree [62], a source code tree differencing framework to identify deletions and insertions. Similarly, we used components of QACrashFix [71] which allows to more accurately account for replacements. These tools create an AST representation of each program file, both before- and after-fix, and produce a set of changes performed between them.

The list of changes generated by these tools is then matched to the studied mutation operators starting by the fine-grain mutation operators, and then defaulting to the coarse-grain mutation operators if no fine-grain mutation operator is found. We seek each of the mutation operators that can match a given set of edits. For example, to identify a *Null Checker* application, for each action describing a commit, we check if the manipulated node is an *IfStatement*. If so, we check whether the action is a node insertion. If so, we check if the condition in the inserted *IfStatement* is an *InfixExpression* that compares an *Expression* to a *NullLiteral*. In this case, we count this sequence

of actions as an instance of a *Null Checker* mutation operator. We created such an automated procedure for all the mutation operators. These strategies are necessarily heuristic, and we do not claim perfect soundness in our matching, instead aggregating results over a large dataset.

Results:

Figure 5.3 shows the distribution of edits used by developers when fixing a bug in the analyzed corpus in order of most common to least common. The most common mutation operators used by human developers are Append (61% of the edits, followed by Sequence Exchange (15 % of the analyzed cases); the least used mutation operators are Upper Bound Set and Off by One, both of which only appeared a couple of times through our analysis, therefore when represented as a percentage of the corpus in Figure 5.3, its value is close to zero. These results detail in a granular way what is the distribution of edit operators analyzed from APR approaches that human developers use when repairing errors. The full list can be found in the publication [192].

Mutation operator	Edits found (%)
Append	61.03
Sequence Exchange	15.76
Delete	9.10
Param Replacer	5.93
Param Add/Rem	3.15
Expression Repl	1.28
Method Replacer	1.12
Null Check	0.76
Castee Mutator	0.61
Replacement	0.60
Expression Add/Rem	0.37
Cast Check	0.11
Size Check	0.07
Range Check	0.04
Object Initializer	0.03
Caster Mutator	0.03
Lower Bound Set	0.01
Upper Bound Set	$\sim$ 0.00
Off by One	$\sim$ 0.00

Figure 5.3: Distribution of mutation operators mined from the selected corpus of analysis. The most commonly used mutation operator by human developers is the “Append” operator, and the least used operator is “Off by One” according to our analysis.

Answer to Research Question 7: The distribution of mutation operators is described in Figure 5.3. The most common mutation operator is “Append” and the least common operator is “Off by One”.

Using this mined distribution we then designed and executed an experiment (Section 5.3) to compare the quality of the patches generated using an APR approach based on this distribution against approaches using heuristic-based distributions.

The results highlighted in this Section have been published in the IEEE International Conference on Software Analysis, Evolution and Reengineering [192].

5.3 APR Tool Informed by Developer Edit Distribution

Finally, we created an APR approach informed by the distribution of Section 5.2 and compared its performance to other APR techniques. We augmented JaRFly, a publicly available tool described in Section 3.2 to include a selection mechanism informed by the distribution described in Section 5.2. Our following research question is:

> **Research Question 8:** How does a human-informed automatic program repair tool compare to the state-of-the-art in APR?

Methodology:

Because we compare to single-edit techniques, we restrict attention to the subset of the Defects4J bugs with single-line human patches (63 buggy programs). We compare our approach against three APR approaches: GenProg [116], PAR [99], and TrpAutoRepair [169].

We implement a novel syntactic generate-and-validate repair technique that differs from prior work first, in the range of mutation operators considered (Section 5.2.1) and second, in how it chooses between those operators and instantiates them. We created this technique by extending JaRFly an open-source implementation several automatic program repair approaches for Java described in Section 3.2. We extended the framework by adding a mechanism that allows the tool to select between the mutation operators according to the probabilities described by a model. We used the model described in Section 5.2.2 and both categories of mutation operators analyzed in Section 5.2.1.

Our approach uses a two-level model (Figure 5.4) for operator selection/instantiation. The first level informs the selection of the given mutation operator, from a set of legal operators at a given potentially-faulty location (e.g. *Parameter replacer* cannot be applied to a *BreakStatement*). If the operator selected is a *Replace* operation, the second level informs the selection of the replacement code. To build both models, we perform an incidence count of each mutation operator and replacement observed in our dataset, matched as described in Section 5.2.2. We then apply Laplace smoothing [177] with $\alpha = 1$ to account for zero occurrences in the replacements lower level model. These two models in detail are as follows:

The *Mutation Operator Probabilistic Model* describes the probabilities of choosing between the several different mutation operators at a particular fault location. The model is built by analyzing the incidence of each mutation operator observed in our dataset and matched as described in Section 5.2.2. As described in Figure 5.3, Template-Based and Statement-Edit mutations contribute 29.26% and 70.74% of the studied edits, respectively.

If the "Replacement" mutation operator is selected, the *Replacements Probabilistic Model* describes the probability of replacing one statement (*"replacee"*) with another (*"replacer"*), thus

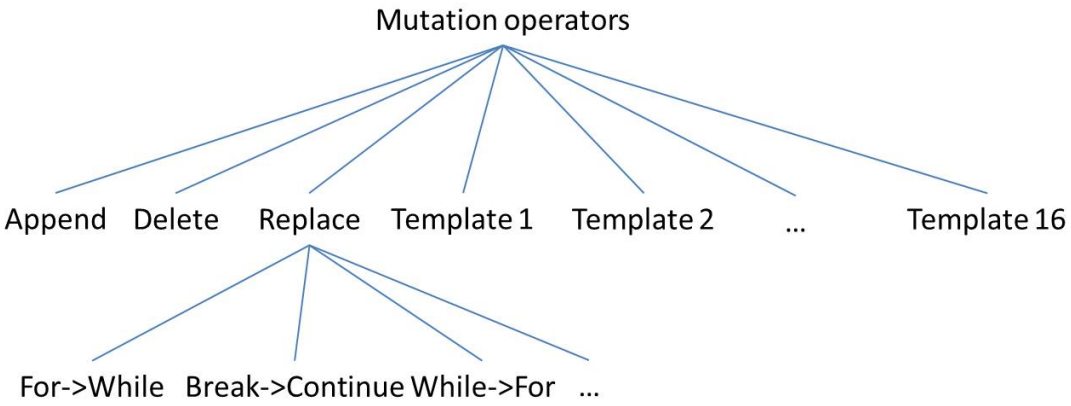


Figure 5.4: This Figure describes the two level probabilistic model used in our study. The first level selects among three coarse-grain mutation operators and 16 fine-grain templates. If the "Replace" mutation operator is selected, then a second level of the model is used to select which statements replace the selected fault location. This probabilistic model is then used to inform operator selection and instantiation in the context of an automatic program repair approach.

informing the selection of replacement fix code. For the *Replacements Probabilistic Model*, we consider the 22 different statement types detailed by Eclipse JDT as direct subclasses of the class `Statement`:

1. AssertStatement
2. Block
3. BreakStatement
4. ConstructorInvocation
5. ContinueStatement
6. DoStatement
7. EmptyStatement
8. EnhancedForStatement
9. ExpressionStatement
10. ForStatement
11. IfStatement
12. LabeledStatement
13. ReturnStatement
14. SuperConstructorInvocation
15. SwitchCase
16. SwitchStatement
17. SynchronizedStatement

Bug ID	Prob. Model		GenProg		TrpAutoRepair		PAR	
	Found	Generalize	Found	Generalize	Found	Generalize	Found	Generalize
Chart # 1	5	×	6	×	4	×	2	×
Closure # 10	2	✓		—		—		—
Closure # 18	1	✓		—		—		—
Closure # 86	2	✓		—	1	✓		—
Lang # 33	1	✓		—		—	1	✓
Math # 2	1	✓		—		—	1	✓
Math # 75	1	✓		—		—	1	✓
Math # 85	4	×	8	×	3	×	8	×
Time # 19	2	✓	1	✓	1	✓		—

Table 5.2: Comparison of patches generated using the probabilistic model-based repair and other state-of-the-art approaches. "–" indicates no patch found. The "Found" column indicate the number of patches found per bug over the multiple random trials. "Generalize" indicates whether all produced patches generalize to the held-out test suites (✓) or not (×). In these results, all produced patches for a bug, technique pair either all generalized, or none did.

18. ThrowStatement
19. TryStatement
20. TypeDeclarationStatement
21. VariableDeclarationStatement
22. WhileStatement

Given 22 statement types, there are 484 possible combinations of replacements. Note that the observed probabilities are not reciprocal, e.g. that the probability of a `For` loop replacing a `While` loop is different from the probability of a `While` loop replacing a `For` loop. This model is built analogously to the mutation operator model, based on replacer/replacee statement incidence.

We then used both of these models to inform the selection mechanism of our extended version of JaRFly. Finally, we compared the patches generated using our new APR approach against the patches generated by GenProg, PAR and TrpAutoRepair as implemented in JaRFly.

Results:

Table 5.2 shows a comparison between the patches found when using our probabilistic model to guide the selection of mutation operators in the context of APR against the patches found on the described bugs using off-the-shelf state-of-the-art approaches. Column 1 shows the defect ID as labeled by Defects4J. The remaining columns show the number of patches found on the left and if all the patches generated by that tool for that bug fully generalized (✓) or not (×) to the held-out test suite. A patch fully generalizes to a held-out test suite when it passes all the test cases contained in the test suite.

From the 19 distinct patches created by our approach, 10 pass all held-out test suite (52.6%); For these same bugs, GenProg generated 15 patches, one of which generalized (6.6%). Similarly 22.2% of TrpAutoRepair's; and 23.1% of PAR's patches generalize to the held-out test suite. Figure 5.5 shows our technique's patch for the Closure #18 bug; it is identical to the human patch.

Regarding how many bugs each APR approach was able to patch, our technique patched 9

```
1288 -if(options.dependencyOptions.needsManagement() &&
        options.closurePass){
1289 +if(options.dependencyOptions.needsManagement()){
1290   for (CompilerInput input : inputs) {
1291  // Forward-declare all the provided types, so that
1292  // they are not flagged even if they are dropped.
1293     for (String provide : input.getProvides()) {
1294       getTypeRegistry().forwardDeclareType(provide);
1295     }
1296   }
```

Figure 5.5: A patch generated using the probabilistic model, identical to the developer patch. No other approach found this identical patch.

APR Technique	Bugs Patched	Patches Generated	Patches Generalize (#)	Patches Generalize (%)
Probabilistic Model	9	19	10	52.6%
GenProg	9	46	5	10.9%
TrpAutoRepair	16	30	4	13.3%
PAR	8	34	11	32.4%

Table 5.3: The left column describes the APR techniques under comparison. The second column, the number of bugs patched by each technique, the third column, the number of patches generated in total by each technique. The fourth column outlines the subset of patches that fully generalized to the held-out test suite. Finally the last column illustrates the number of high-quality patches (that fully generalize to the held-out test suite) generated by each technique as a percentage of the total number of patches generated by that approach.

of the 63 bugs in our evaluation (shown in Figure 5.2). From these 63 bugs, GenProg was able to patch 9; PAR, 16; and TrpAutoRepair, 8. There are 37 bugs for which at least one approach produced at least one patch. patch. From these, 19 were patched by only one approach; 10 were patched by two; 3 patched by three; 5 patched by all four. These are described in Table 5.3.

Regarding how many patches each APR approach generated, our approach created 19 distinct patches for the aforementioned 9 bugs (remember that these techniques can create several patches per each bug). Of these, 10 (52.6%) pass the held-out test suites. Genprog created 46 distinct patches; 5 of them (10.9%) pass the test suites. TrpAutoRepair created 30 patches; 4 of them (13.3%) pass the test suites. Finally, PAR created 34 patches; 11 of them (32.4%) pass the test suites.

Based on the overall results shown in Table 5.3, we conclude that a mutation operator selection mechanism informed by developer behavior and included into an APR technique is able to generate a smaller number of patches as compared to the other APR techniques (described in column "Patches Generated"). However, the patches generated by our approach are of higher quality than the patches generated by other APR techniques (detailed in column "Patches Generalize (%)").

Answer to Research Question 8: An APR technique using a mutation operator selection mechanism informed by developer behavior is able to generate less but higher-quality patches as compared to the other APR techniques.

5.4 Multi-Edit Rules to Inform Automatic Program Repair

Although single-edit patches can repair many non-trivial bugs in real software, the majority of high-quality bug fixes in real software require multiple edits [194, 226]. The number of combinations of possible mutation operators to apply in a sequence increases exponentially with the number of combined source code changes.

Even though recent approaches have proposed initial ideas towards tackling the navigation of this vast search space (e.g., HERCULES [179]) the multi-edit repair field remains largely unexplored. Therefore as an initial step towards exploring multi-edit repair, we propose an analysis of multi-edit source code changes by mining a more expressive model of common changes. In particular, we extract *association rules* to model chains of several edits, capturing the way humans create these kinds of fixes.

Association rules are if/then statements that show relationships between elements in a dataset which happen frequently together. To create these association rules, we use the well established association rule mining algorithm Apriori [7].

In this Section, we describe and evaluate the mutation operator association rules produced by mining human patches to identify edits that commonly occur together in human-generated patches. The goal of these models is to provide intuition regarding how to form multi-edit source code changes. Note that we create these association rules using strictly the Mutation Operator model corpus; the Replacements operator corpus is only informative when the "Replace" operator is chosen, and thus does not apply to the question of chaining together edits to produce larger patches. We therefore ask the research question:

Research Question 9: What are the most common multi-edit modification rules in practice?

Methodology:

To answer this research question we first analyze what are the code edits that happen commonly together by analyzing the association rules with the highest confidence, and then examine the effectiveness of the association rules by reasoning about different confidence thresholds to determine understand the best parameters to create these multi-edit association rules.

First, we mine association rules for the Mutation operators model (Section 5.2.2) by analyzing mutation operator incidence in the studied commits. We develop rule sets at different *Confidence* levels. Confidence in the context of association rules is defined as:

$$conf(X \implies Y) = \frac{supp(X \cup Y)}{supp(X)}$$

Where X and Y are items in a transaction (mutation operators, in our context). Confidence is calculated according to its Support (*supp*), an indication of how frequently the set of mutation operators (item set) occurs in the corpus. Formally:

$$supp(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|}$$

Where X is the item set and t is each individual transaction in the database of transactions T. Apriori identifies the mutation operators that frequently happen together in a set of commits, iteratively extending them to larger item sets that appear often in the transactions as identified by these metrics.

Then, to evaluate the effectiveness of the association rules in the context of the automatic program repair process, we first remove from the corpus human patches with fewer than three edits. This is because our mined rules all require at least two antecedents and one consequent. This removed 62.83% of the corpus. We validated the rules on the remaining 37.17% of patches as follows. First, we divide our corpus in 10 folds. For each fold, we build association rules on the remaining nine, as described. Given the mined rules, we then we analyze how many testing patches (instances in the fold used as testing data) can be built by applying the learned rules. We categorize them as either *Fully covered*, *Partially covered*, or *Not covered*.

To illustrate via example, Table 5.4 shows three instances of patches in the testing data, and three rules. Instance 1 can be constructed by applying rules 1 and 3 and is thus classified as Fully covered. Rule 1 would cover the first three edits of Instance 2 instance, but there is no way to create the Replace ("Rep") mutation using the listed rules. This instance is classified as Partially covered. For Instance 3, even though Rule 3 contains two of the edits in the rule's antecedent, the instance does not contain the rule's consequent. The rules do not apply, and thus this instance is classified as Not covered.

Table 5.4: Transactions describing the statements used in patch instances on top. At the bottom, are a set of associated association rules generated.

	Instances
1	Del; App; NullCheck; ObjInit
2	Del; App; NullCheck; Rep
3	App; NullCheck; CastMut
	Rules
1	Del $\wedge$ App $\rightarrow$ NullCheck
2	App $\wedge$ ParamRep $\rightarrow$ Rep
3	App $\wedge$ NullCheck $\rightarrow$ ObjInit

Results:

Below, we list the top 10 rules identified with 100% confidence in the dataset. This means that in 100% of the cases observed, every transaction that contained the antecedent of a rule also

contained the consequent. A high threshold like 100% produces rules for APR that predict with high accuracy which edits to perform, given an initial set of edits. These rules are obtained with a 1% support, which means that each of these rules individually appear in at least 1% of all the transactions in the corpus. We show only the top association rules (the full set of rules is released with the code and data associated with this paper):

Table 5.5: Association rules with 100% confidence generated to detail most common code changes by human developers, using patches with over 3 mutation operators

Association rules with 100% confidence
Replace $\wedge$ Delete $\implies$ Append
Delete $\wedge$ AddNullCheck $\implies$ Append
Replace $\wedge$ SeqExchanger $\implies$ Append
Replace $\wedge$ ParamReplacer $\implies$ Append
Delete $\wedge$ CasteeMutator $\implies$ Append
Replace $\wedge$ Delete $\wedge$ ParamReplacer $\implies$ Append
Replace $\wedge$ AddNullCheck $\implies$ Append
Replace $\wedge$ Delete $\wedge$ SeqExchanger $\implies$ Append
Delete $\wedge$ ExpressionAdder $\implies$ Append
Delete $\wedge$ AddNullCheck $\wedge$ ParamReplacer $\implies$ Append

The key observation to draw from these rules is that "Append" is the most common single edit mutation operator applied by developers. This behavior is reflected in the fact that it is the consequent in all the top mined rules. Overall, association rules provide an intuition of which common patterns of code changes developers use. These rules tell us which edits happen frequently together, supporting understanding of multi-edit source code changes, an understudied area that covers the majority of real patches.

Finally, we performed this analysis at 6 different confidence thresholds (50%, 60%, 70%, 80%, 90%, 100%) to analyze the tradeoff between ruleset expressive power and size. A high confidence produces a small number of very accurate rules (when the antecedent is present, it is very likely that the consequent will be present as well). Setting the confidence lower produces the opposite trade-off: a large set of rules (covering more instances) where if the antecedent is present, it is less likely that the consequent will be present too. For each confidence threshold, we performed a standard 10-fold cross validation process with all the instances and all the rules for each fold and finally, we aggregate the results from all folds.

Figure 5.6 shows results. As expected, the number of rules created increases as confidence decreases. Note also that the number of Fully Covered instances increases as the confidence decreases, due to the fact that there are more rules, even though these rules are less accurate.

APR would benefit from having a small number of very accurate (high confidence) rules that would describe what edit to perform next after a series of edits, but at the same time, it needs rules that are flexible enough that they can generalize to a big portion of the patches. We find a good tradeoff at a confidence threshold of 90%. The 100% threshold provides very accurate rules, but

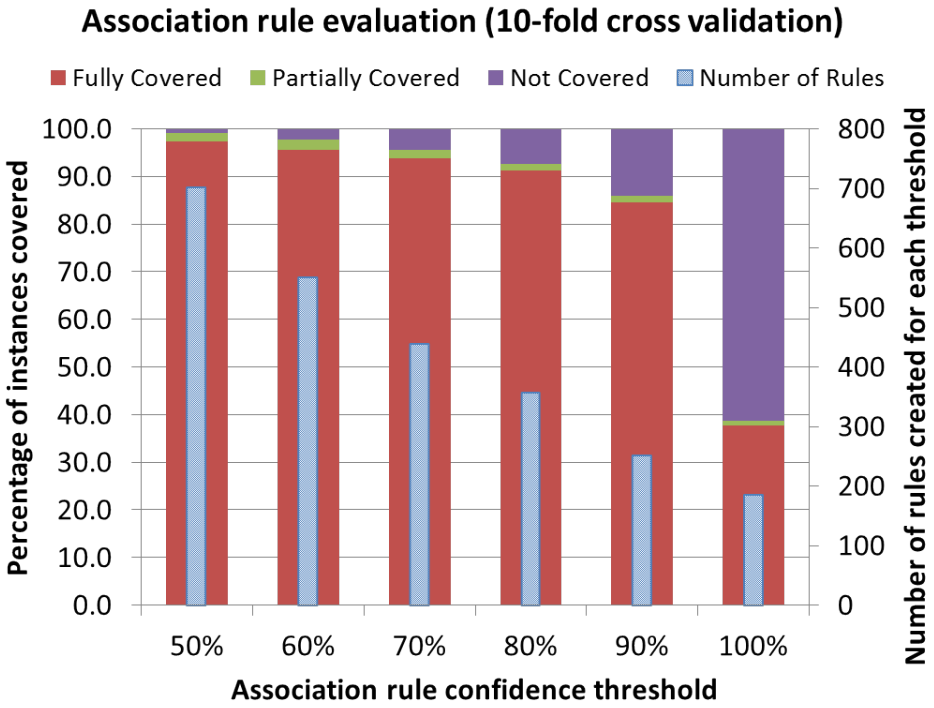


Figure 5.6: The wide bars use the left vertical axis to describe the percentage of patches covered (Fully, Partially or Not) by the association rules. The thin bars use the right vertical axis to describe the number of rules created for each confidence threshold.

can fully cover only 37.7% of the evaluation patches. By contrast, the 90% confidence threshold produces slightly more rules, but they are able to fully cover 84.6% of the patches.

Answer to Research Question 9: The most common multi-edit rules are described in Table 5.5, the most common consequent is "Append".

This Chapter shows how human developers modify certain portions of code much more often than others when fixing bugs, and having an APR approach behave in a similar manner to humans help the APR approach's possibility of generating higher quality patches.

We execute a series of experiments that allows us to analyze what code do human developers modify the most. We then create an APR approach that simulates how human developers modify code, and we analyze the quality of patches generated by said approach. Our results show that an APR approach that reflects the way human developers patch code leads to higher quality patches. Finally, our experiments also explore multi-edit patches, and shows association rules of edits that happen commonly together in human developer code changing patterns. These results open possibilities for future studies into how to apply this gathered knowledge of association rules of edits to be able to create higher quality mutli-edit patches.

Chapter 6

Incentivizing Patch Diversity in Automatic Program Repair

Generate-and-validate automatic program repair approaches have the ability to generate several patches for a single bug, this is in part because they aim to solve an underspecified problem as described by the guiding test suite, which by definition is a *partial* specification of the desired solution. These techniques can thus traverse the search space of code changes that lead to a program transformation that can satisfy the constraints described in the guiding test suite.

The partial specification — a test suite — therefore fails to distinguish between patches that pass the tests and implement the desired functionality, and the patches that pass the tests but fail to implement the desired functionality not encoded by the partial specification. The search space of possible patches is large [128] and navigating it in a way to improve the probability of finding a high-quality patch [97, 128, 129, 215] is at the heart of solving the repair quality problem.

The diversity of patches produced in such a way, even by a single technique, may be used to improve the overall quality of a patch [189]. In essence, given a set of *plausible* patches, either a subset of them are *correct* patches that accurately transform the program into a version that follows the intended behavior (e.g. Figure 5.5); or the patches fix the errors for the specified test cases, but fail to generalize this solution to a broader set of executions.

In the latter case, we can still find ways to use these lower-quality patches, for example to use them for N-Version software where a system can shift between patches depending on the executions that behave correctly; for patch consolidation, where we unify several lower-quality patches into a single higher-quality patch; or to provide human developers with a set of patches for they to decide which of these lower quality patches will better generalize to an intended program behavior.

In this Chapter we explore ways to incentivize patch diversity in the APR process and the impact diversity has in patch quality. The goal of incentivizing patch diversity is both to increase the probability of APR approaches to find *correct* patches and to create a broader set of *plausible* patches that can be used for the purpose of increasing software quality in contexts such as n-version software and patch consolidation as described below.

N-Version Software:

The creation of several different patches can be seen as a form of n-version programming [38],

and it is subject to the same constraints as n-version programming. However, there is a main difference we are leveraging in this Chapter between off-the-shelf n-version programming and n-version programming based on APR patches.

In standard n-version programming (or n-version software), different groups of developers are provided a single program specification, and the goal is for them to generate functionally equivalent programs [38]. The intuition being that independently generated programs would have enough diversity to provide online fault tolerance (e.g., if a version of software fails in an execution, another version might not fail for that particular execution and therefore a system might be able to shift between programs to provide fault tolerance).

Previous studies show that human program repair usually lacks the scale of diversity required to effectively achieve the goals that motivate n-version programming [101]. Correlations in faults of human-written programs prevent a quality improvement when implementing N-version programming.

To our advantage, automatic program repair techniques are freer of these human biases [182] that restrict manual efforts [5, 131]. Thus, testing if this approach works for automatically generated patches is, in some sense, a measure of whether human-written and automatically-generated patches differ in their diversity profiles. Our intuition is that automated repair patches minimize these fault correlations, especially if we are able to create an approach that actively incentivizes diversity in generated patches.

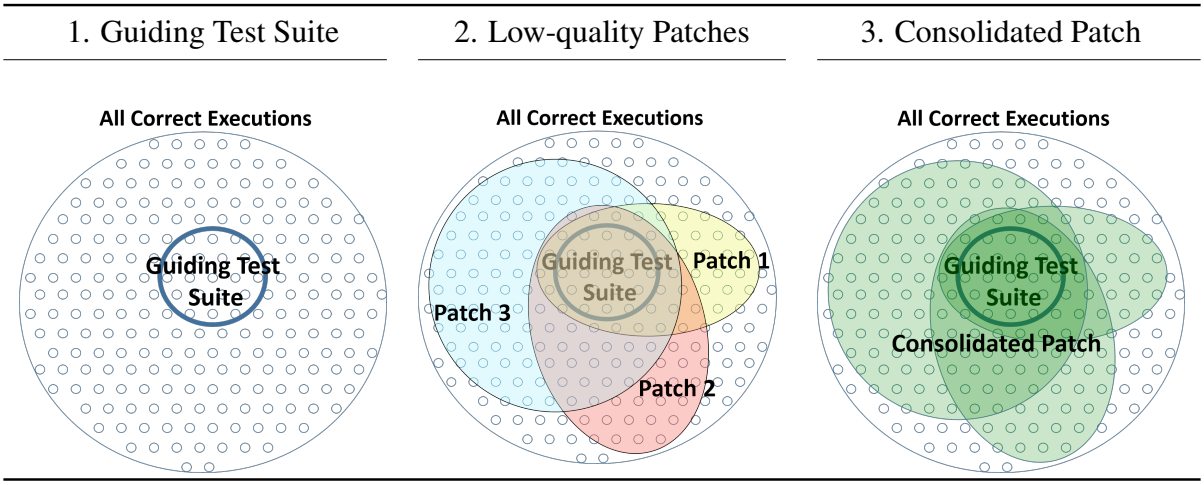


Figure 6.1: 1. Describes how the guiding test suite is a subset of all the possible correct executions of the program's intended behavior. 2. Depicts how low-quality patches can be created to behave correctly for the cases described in the guiding test suite but fail to generalize to all intended behavior. 3. Shows how a consolidated patch could cover a superset of the correct executions of its corresponding low-quality plausible patches. While it might still not fully generalize, it could cover a superset of correct executions that lower quality patches could not.

Consolidation:

Another possible use of patch diversity is by leveraging diverse patches to create a single higher-quality consolidated patch. The intuition behind this idea is that each plausible patch might be a partial solution that fails to cover all possible correct executions of the expected behavior

of the program, therefore consolidating several plausible patches might increase the number of correct executions and thus the quality of the overall solution.

Figure 6.1 shows a graphical description of how patch consolidation could potentially increase the number of executions covered by a consolidated patch. Considering the full set of correct executions of a program (which is potentially infinite) Image 1 of Figure 6.1 describes the guiding test suite as a limited subset of all possible correct executions. Image 2 shows how low-quality plausible patches implement functionality that covers at least the executions described by the guiding test suite. Image 3 shows how by consolidating low-quality patches we can cover a superset of the correct executions of its corresponding low-quality plausible patches.

Program that displays temperature in different measurement systems

```
1   /* Input:
2   * int degrees: degrees in Fahrenheit
3   * char system: 'c' for Celsius, 'f' for Fahrenheit, 'k' for Kelvin
4   * Output:
5   * String: detailing the converted temperature and the system used
6   * Empty string if invalid system */
7  public String convertedTemp(int degrees, char system){
8     String ret = "";
9     if(system == 'f'){
10       ret = Integer.toString(degrees);
11       ret += " °F";
12    }
13    if(system == 'c'){
14       degrees = (int)((degrees - 32) * 5/9);
15       ret += " °C"; // bug! "degrees" was never added to "ret"
16    }
17    if(system == 'k'){
18       degrees = (int)((degrees - 32) * 5/9 + 273.15);
19       ret = Integer.toString(degrees);
20       ret += " °K";
21    }
22    return ret;
23 }
```

Set of test cases that describe intended program behavior, referred to as the *guiding test suite*

```
Test1: assertEquals(convertedTemp(-87,'f'), "-87 °F")
Test2: assertEquals(convertedTemp(55,'f'), "55 °F")
Test3: assertEquals(convertedTemp(2,'k'), "257 °K")
Test4: assertEquals(convertedTemp(-23,'k'), "243 °K")
Test5: assertEquals(convertedTemp(32,'c'), "0 °C")
```

Figure 6.2: Illustrative example of a program that displays temperature in different systems. At the top there is a program that takes two variables as inputs: the degrees in Fahrenheit, and the system to display. It returns the temperature to be displayed. At the bottom, a test suite that partially describes the intended behavior of the program.

Motivating Software Diversity Example:

Consider Figure 6.2 which depicts a program that displays temperature in different measurement systems. This is composed of a method called `convertedTemp`. It receives two parameters: `degrees` which is an integer describing the degrees in Fahrenheit, and `system`, a char describing which measurement system should be displayed on. In line 8 a return variable is created. From line 9 to 12, the method handles the case of the Fahrenheit system. From line 13 to 16 the Celsius system and from line 17 to 21 the Kelvin system. The goal in each of these cases is to properly convert the degrees using the appropriate convertion formula, add that value to the degrees String, and add which system is being used before returning the corresponding String. The return value is a string with the correct degrees and the measurement system.

Notice that this program contains an error in the case of converting to Celsius ('c') where the variable `degrees` is converted to the correct number (line 14), but it is never added to the return String `ret` (line 15).

At the bottom section of Figure 6.2, there is a test suite partially describing the expected behavior of the program above. This test suite is composed of five test cases showing a range of examples of desired behavior for displaying different temperatures using the three measurement systems. Given the bug highlighted in line 15 of the program, Test5 from the test suite fails showing the erroneous behavior. The expected return value when calling `convertedTemp(32,'c')` is the String "`0 °C`" given that 32 degrees Fahrenheit equals 0 degrees Celsius. However, because of the bug in line 15, the actual return value of the function is "` °C`", causing Test5 to fail.

Samples of three low-quality plausible patches that overfit to the guiding test suite

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       ret += " °C"; //bug! "degrees" is never added to "ret"
10  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       if(degrees == 0)
10        ret = Integer.toString(degrees);
11      ret += " °C"; //bug! "degrees" is never added to "ret"
12  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       ret += " °C"; //bug! "degrees" is never added to "ret"
10  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       if(degrees <= 0)
10        ret = Integer.toString(degrees);
11      ret += " °C"; //bug! "degrees" is never added to "ret"
12  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       ret += " °C"; //bug! "degrees" is never added to "ret"
10  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       if(degrees >= 0)
10        ret = Integer.toString(degrees);
11      ret += " °C"; //bug! "degrees" is never added to "ret"
12  }
```

Figure 6.3: Example of three overfitting plausible patches. The first patch creates a fix only for the execution described by Test5 from Figure 6.2. The following two patches describe code changes that satisfy the description of Test5 and also a superset of executions (`degrees <= 0` in the second patch and `degrees >= 0` in the third patch correspondingly).

Figure 6.3 shows a diverse set of plausible patches which are able to pass all the test cases in the guiding test suite, but fail to generalize to an independent evaluation. The first patch creates a fix for the execution described by Test5 from Figure 6.2 by only checking the values described in the test case, specifically, creating a fix only for the case where `degrees == 0`. The following two patches describe code changes that satisfy the description of Test5 but also include a superset

of executions. The second patch handles the cases where `degrees <= 0` in the second patch and the third patch the cases where `degrees >= 0`.

Back to how patch diversity can be used to implement n-version software, the patches described in Figure 6.3 demonstrate a case where three diverse patches can be created by following the same program specification (the test suite in Figure 6.2). Thereafter, a system could shift between these three patches to provide fault tolerance (e.g., the system can use the first patch when $degrees == 0$, then shift between patches two and three depending whether the value of *degrees* is above or below zero).

Consolidated patch that generalizes to an independent evaluation

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       ret += " °C"; //bug! "degrees" is never added to "ret"
10  }
```

```
7   if(system == 'c'){
8       degrees = (int)((degrees - 32) * 5/9);
9       if(degrees == 0)
10        ret = Integer.toString(degrees);
11      if(degrees >= 0)
12        ret = Integer.toString(degrees);
13      if(degrees <= 0)
14        ret = Integer.toString(degrees);
15      ret += " °C"; //bug! "degrees" is never added to "ret"
16  }
```

Figure 6.4: Example of a high-quality consolidated patch created from the three plausible patches in Figure 6.3 that generalizes to the intended program behavior.

Finally, Figure 6.4 shows an example of how when these low-quality overfitting patches described in Figure 6.3 can be consolidated, the resulting patch is able to intuitively generalize to a superset of correct executions. This example shows a simple case where patch consolidation can be used to increase the quality of overfitting plausible patches. Patch consolidation is in itself an on going complex field of study.

This Chapter provides an in-depth study of patch diversity, how individual patches can be consolidated into n-version patches, and how does diversity impact patch quality. We start by evaluating the the idea of a hypothetical n-version patch and how would it behave in comparison to the individual patches it is generated from. We then propose and evaluate a set of diversity driven APR approaches and their corresponding patch quality evaluation. Finally, we move to creating consolidated patches and analyzing their quality in terms of each APR's ability to generate diverse patches. Therefore, in this Chapter we will answer the following research questions [1]:

RQ10 Can overfitting be mitigated by exploiting randomness in the repair process? Do different patches overfit in different ways?

Answer: The patches exhibit insufficient diversity to improve quality through some method of combining multiple patches.

RQ11 How does software diversity impact patch quality in APR?

[1]Portions of the work described in this Chapter have been published in the journal IEEE Transactions on Software Engineering [148], and the IEEE/ACM International Conference on Automated Software Engineering (ASE) Doctoral Symposium track [193]. The remaining unpublished portions are under submission for publication to the journal Empirical Software Engineering

Answer: Slicing mutation operators was specially favorable for patch quality by finding patches for a greater number of bugs, a higher number of unique patches, higher average patch quality, and highest quality patch for the majority of the groups of executions analyzed.

RQ12 How does patch diversity relate to patch quality when plausible patches are consolidated in relation to their non-consolidated counterparts?

Answer: Consolidated patch quality is proportional to the APR technique's ability to generate diverse plausible patches.

6.1 Quality of Hypothetical N-Version Patch vs. Individual Patches

In this Chapter we analyze the benefits of patch diversity as a means to increase patch quality in the automated program repair process. We frame a study where we look into the behavior of different patches when evaluated by individual tests of the held-out test suite.

> **Research Question 10:** Can overfitting be mitigated by exploiting randomness in the repair process? Do different patches overfit in different ways?

To answer this research question, we separate the question of how to combine patches from the question of whether it might be worthwhile to combine patches. We answer the latter question. The APR approaches described in this thesis use stochastic processes to create patches, therefore they can create several patches for a single bug. Given a set of patches for a defect, the majority of the patches passes an evaluation test, then it is possible that the n-version combination would pass that test. If the overall quality of an n-version patch across the entire evaluation test suite is higher than that of the individual patches, then perhaps it is worthwhile to attempt to combine them. Conversely, if the n-version patch quality is no better than the individual patches, combining is unlikely to improve quality.

Methodology:

For this research question we used the unique patches generated by the three repair techniques implemented in JaRFly described in Section 4.1. These patches were created by running these repair techniques on all 357 Defects4J defects using the developer-written test suites, with 20 different seeds per defect for GenProg, PAR, and TrpAutoRepair. This produced 561 unique patches (255 by GenProg, 107 by PAR, and 199 by TrpAutoRepair, recall Figures 4.2, and 4.1).

For each technique, we identified the defects for which that technique produced at least 3 distinct patches. For these defects, we then evaluated how the potential n-version patch would perform by executing the evaluation test suite on each patch and considering the n-version to pass the test if the strict majority of the patches passed the test. For GenProg, 30 defects qualified for this experiment, 9 for PAR, and 25 for TrpAutoRepair.

For each bug b_i, consider the set of patches that pass the guiding test suite P_i. P_i is composed of patches p_i^j where i describes the bug being fixed and j the patch number within the set. Similarly, acknowledge two test suites involved in the APR process, the guiding test suite G_i composed of

tests g_i^k, and the held-out test suite H_i comprised of tests h_i^l where k and l describe the test number accordingly. All tests g_i^k pass when evaluated in p_i^j by definition of plausible patches in the APR algorithm.

In this research question we analyze the quality of a hypothetical n-version patch y_i where the quality of y_i is defined by by having each test case $h_i^l \in H_i$ "vote" on whether the hypothetical n-version patch y_i would pass h_i^l if y_i existed. We consider the test h_i^l passed by y_i if the sum

$$\sum_{n=0}^{j} x \mid x = 1 \; iif \; h_i^l \; passes \; when \; evaluated \; in \; p_i^j$$

is greater than the mean of the number of patches in P_i. Otherwise, we consider the n-version patch y_i failed to generalize to test case h_i^l. We then aggregate the number of tests h_i^l that pass y_i in the form of a percentage. Finally, we compare the performance of the n-version patch to the performance of each individual patch p_i^j. We perform this analysis by executing each test h_i^l in H_i on all patches p_i^j in P_i for each bug b_i.

To compare the quality of the n-version and individual programs, we use the nonparametric Mann-Whitney U test. We choose this test because our data may not be from a normal distribution. We compute Cliff's delta's *delta estimate* to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95% confidence interval (CI) of the δ estimate.

Results:

Figures 6.5 and 6.6 compare the quality of the n-version patches to the individual patches that make up those n-version patches. The Mann-Whitney U test indicates the differences between the patch quality of the individual patches and the n-version patches are not statistically significant and the δ estimate suggests the differences are negligible.

Our results thus suggest that given the patches generated from our studied APR techniques, the difference in quality performance between a hypothetical n-version patch and individual patches is negligible. The reason for this similarity in quality is that patches generated for a single bug tend to be very similar themselves and behave similarly, thus analyzing their behavior when combined results in similar behavior and similar quality. The fact that we found a negligible quality difference between individual and hypothetical n-version patches could be interpreted as a negative result, but further than that what we learned is that patches generated for a single bug tend to be very similar and this hinders any quality improvement that patch consolidation could bring. Therefore, broader diversity in patch generation is needed to be able to benefit from diverse patch behavior on the potential generation of an n-version patch.

Answer to Research Question 10: The patches exhibit insufficient diversity to improve quality through some method of combining multiple patches.

Our finding is consistent with the prior study for relatively high-quality patches [189]. However, the earlier study found that when patch quality was low (e.g., because of a low-quality test suite being used to repair the defect) the patch diversity may have been sufficient to improve quality [189]. This study does not explore that part of the question because the patches we observe for the Defects4J defects tend to be of relatively-high quality.

technique	patch quality minimum	mean	median	maximum	100%-quality patches
GenProg	78.7%	96.7%	100.0%	100.0%	54.9%
GenProg (n-version)	75.8%	95.7%	99.9%	100.0%	50.0%
PAR	82.4%	97.7%	100.0%	100.0%	76.5%
PAR (n-version)	82.4%	97.6%	100.0%	100.0%	66.7%
TrpAutoRepair	80.1%	97.7%	100.0%	100.0%	59.3%
TrpAutoRepair (n-version)	75.8%	96.3%	100.0%	100.0%	56.0%

Figure 6.5: Quality of the individual and n-version programs made up of those patches.

6.2 Improve Diversity of Generated Patches

As suggested by our previous research question and previous studies [148, 190] an APR approach that incentivizes patch diversity would be beneficial for n-version software solutions and patch consolidation. Therefore, our next two research questions examine this claim by analyzing the benefits of incentivizing diversity in APR (Section 6.2) and consolidation of individual patches into real (non-hypothetical) n-version patches (Section 6.3).

The opposite of diverse patches are redundant or semantically identical patches. Patches as such can be identified when one program can be replaced with another without affecting the observable results of executing the program [167] (programs have the same logical content [143]). Figure 6.7 shows three plausible patches GenProg found (Closure 13). All of them pass all test cases in the guiding test suite.

The first patch performs the minimum change necessary for the failing test case to pass: removing line 49. The second plausible patch deletes line 49, but also adds an irrelevant line of code (line 50). Finally, the third patch performs similar changes to the previous patch, instead of adding a repeated line of code, it adds an if statement that is not executed by the guiding test cases. In examples as such where there is little or no semantic diversity, consolidating these patches would not lead to an improvement in quality given that functionality would only be repeated or overwritten.

We implement an APR technique to enhance the methodology of traversing the search space of patches candidates so that the plausible patches found are more likely to be semantically distinct, therefore amplifying the potential of increasing quality through n-version software or patch consolidation. To this end we propose the following research question:

Research Question 11: How does software diversity impact patch quality in APR?

Methodology:

In this section we describe and evaluate a set of techniques to incentivize patch diversity with the intent to find a way that allows us to explore the search space more broadly and expand the kinds of patches we found therefore finding higher quality patches. We start by slicing the

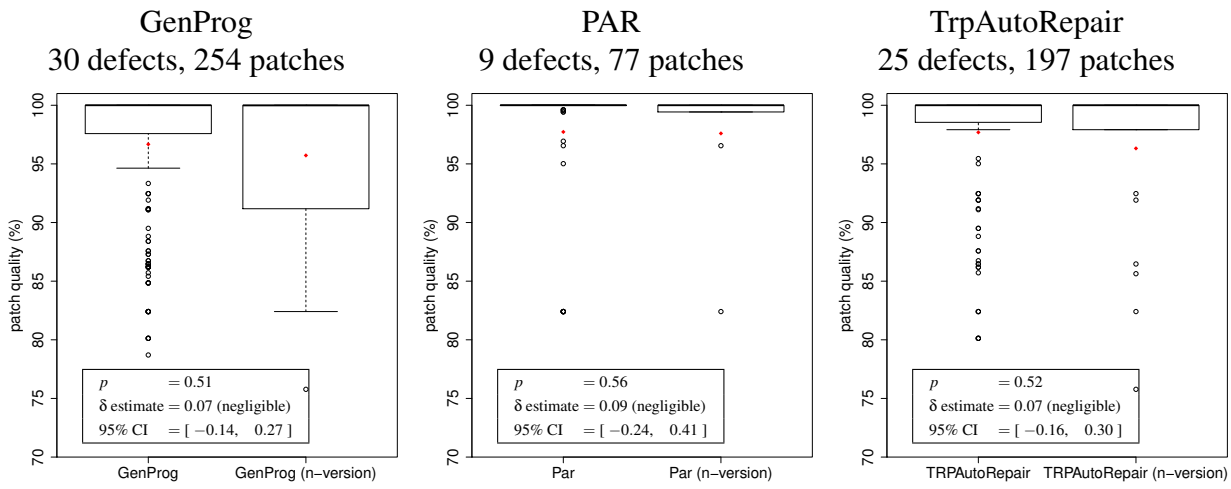


Figure 6.6: The box-and-whisker plots compare the quality of the individual and n-version programs made up of those patches, with the mean as a red diamond. The p values (Mann-Whitney U test) suggest that there is no statistically significant difference in the quality of n-version and individual programs. We measure the effect size using Cliff's Delta test. For the given dataset, n-version programs perform negligibly worse (indicated by the δ estimate) than individual versions for all the three techniques however, the 95% confidence interval spans 0 for all techniques suggesting that, with 95% probability, the quality of n-version program is likely to be same as individual program.

problem space in three different ways (guiding test cases, fault locations and mutation operators), and later we propose an APR approach based on semantic diversity.

We slice the guiding test cases that define the program specification because this changes the description of how the program should behave and therefore it changes how the APR tool interprets a patch and this affects how it navigates the search space. We slice fault locations because this varies the area in which the APR approach looks for places to modify when searching for a patch. Thus, if it finds patches in different locations, these patches have a higher chance of being more diverse.

We slice mutation operators because this provides the APR approaches with different tools to generate new patches. This forces the APR to use only a set of mutation operators per execution and going in depth into what patches can be found with the limited mutation operators. Finally we generate a way to modify the fitness function based on the idea that APR techniques commonly generate semantically similar patches. Our technique introduces a diversity component to the objective function which modifies how the APR approach navigates the search space not focusing only on correctness, but incentivizing diversity as well. These diversity driven techniques and their corresponding experiments are described below:

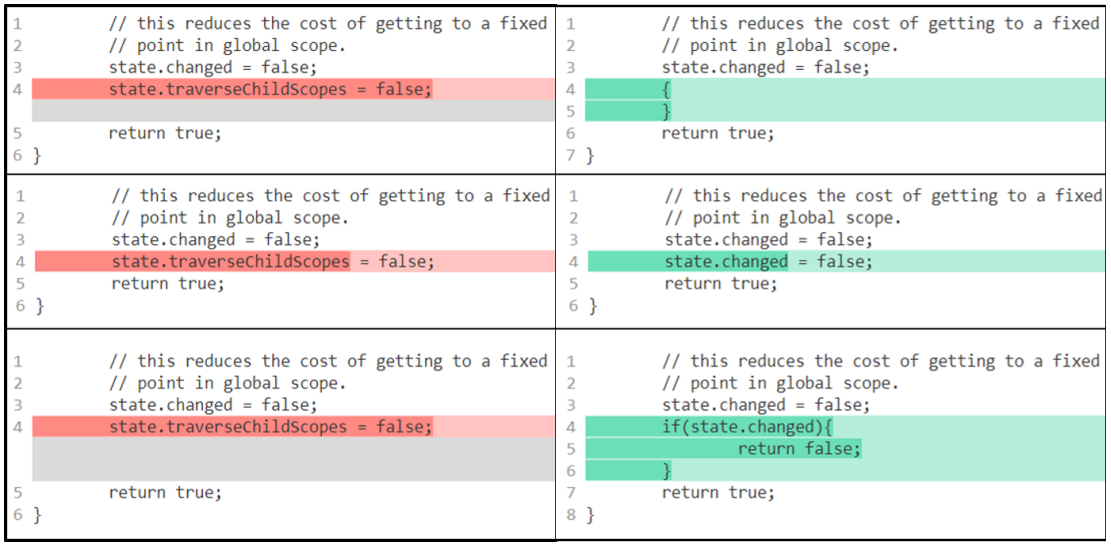


Figure 6.7: Three semantically identical patches showing the lack of diversity in plausible patches generated by automatic program repair approaches.

6.2.1 Modify Fitness Function and Implement Multi-Objective Search to Incentivize Patch Diversity

Several current approaches use genetic programming to find plausible patch candidates. Genetic programming relies on a fitness function used to compute the likelihood of patch candidates to be plausible patches. Most current approaches set their fitness function to look mainly for patch correctness (e.g., [116, 192, 210]) by assigning a fitness score based on the number of passing test cases from the guiding test suite. This fitness score determines the candidate's likelihood to be selected in future generations.

One way to incentivize diversity when traversing the search space is by modifying the fitness function to optimize for both patch correctness and semantic diversity. We created multi-objective search as opposed to the current one-dimensional approach. To implement this multi-objective search, we require a quantitative measurement of diversity to encode it into the search criteria. Since program equivalence is undecidable [175], we have implemented a semantic measurement to approximate software equivalence and its opposite, software diversity:

Test-Suite-Based Semantic Difference Measurement:

Optimizing for patch diversity requires a quantitative measurement of *how different* a program is with respect to another. Figure 6.8 diagrams the implemented process to approximate the semantic difference between two programs, Program A and Program B. The first step (i.e., ① in Figure 6.8) is to create a specification from each of the programs in the form of a test suite describing the behavior of the program as a set of inputs and expected outputs. This can be achieved using test suite generation tools (e.g., Evosuite [65], Randoop [159]).

The second step (i.e., ② in Figure 6.8) is to combine both specifications into a single Specification AB. This consolidated specification is used to evaluate each individual program (i.e., ③ in Figure 6.8) . We then compare the reports from the evaluations using Hamming distance

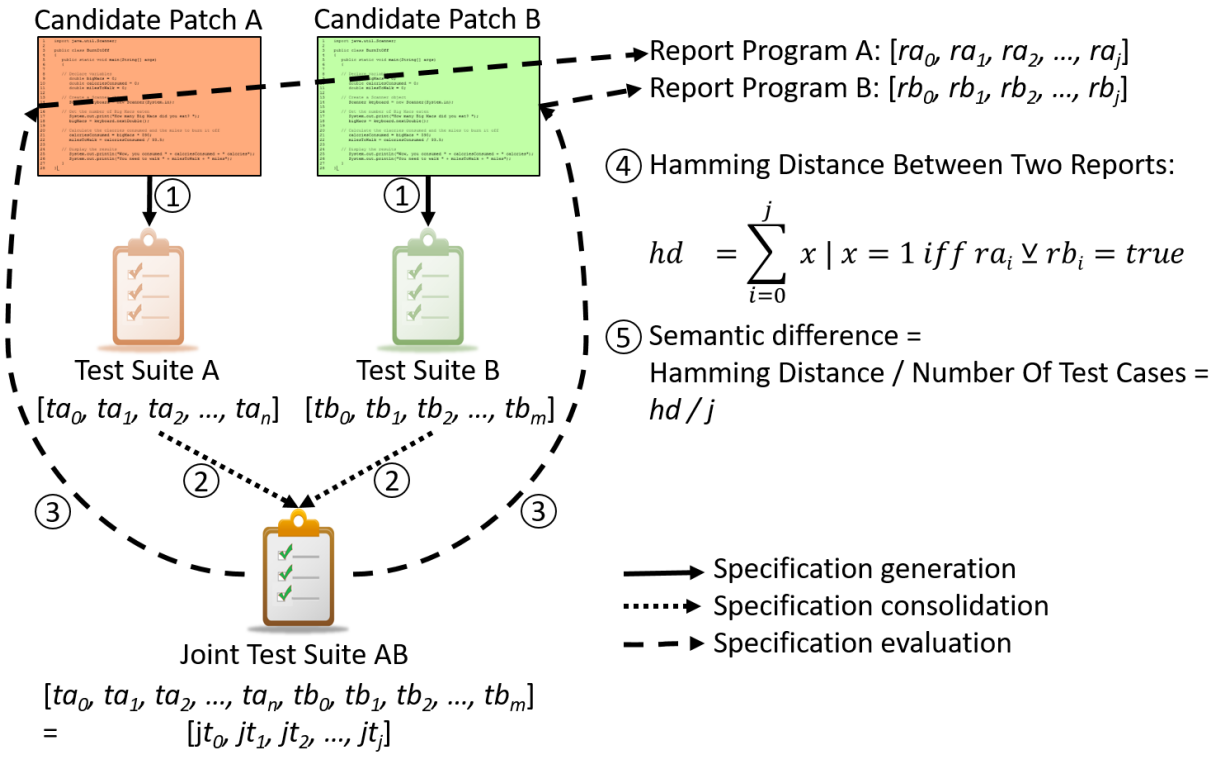


Figure 6.8: Diagram describing the implemented semantic difference measurement. Given two programs, we create a specification from each program (1), then unify both specifications into a single one (2). The unified specification is then evaluated in each of the programs (3). Finally we calculate the semantic difference by computing the Hamming distance between the two executions (4) and represent it as a percentage of the total number of test cases in the unified specification (5).

(i.e., ④ in Figure 6.8) to identify which cases behave differently. Hamming distance is fast and assumes equal distance strings, making it appropriate for our analysis. Finally, we compute the semantic difference based in the Hamming distance as a percentage of the the number of test cases (i.e., ⑤ in Figure 6.8). As a result we obtain a metric describing how semantically different is program A from program B.

It is possible that the joint test suite is not able to compile due to the lack of functionality in one of the programs (e.g., if program A does not have a method evaluated by the test suite generated from program B, the joint test suite will not be able to compile to evaluate program A). In this scenario, this test case behaves differently in both programs and we annotate it accordingly when computing the Hamming distance value. By definition, absent functionality from a program cannot behave equally to present functionality in another program. This behavior is unlikely to happen in modern APR approaches.

This methodology is inspired in literature of semantic code clones [88, 185, 200, 206, 222] defined as program components with similar behavior, but different syntactic representation. This proposed approach uses a similar approach to code clone detection with the major difference that

instead of evaluating code similarity, our approach evaluates code diversity. Semantic code clone detection typically finds pairs of software that might be clones called candidate code clones and then validate clone functional similarity [200] using a variety of techniques.

Previous approaches have used methodologies to find code similarity in the code clone process. Such methodologies include information retrieval [133], program dependence graphs [104], static analysis to extract memory states in function exit points [142], method calls at bytecode level [94] tree-based convolution [222], concolic analysis [106], and randomized testing [52, 88] In our approach, we deal with programs that are almost entirely identical except for portions that are potentially semantically different (patched code). We then evaluate *how functionally different* are the two programs by evaluating their diversity in program behavior.

6.2.2 Slicing Mutation Operators, Fault Locations, or Test Cases

Similarly, we implemented a second set of techniques to increase patch diversity in the automated program repair process based on the intuition of distributed search algorithms (e.g., Multi-Island Genetic Algorithm (MIGA) [83]) where we partition the problem space to find local optima and given the segregation the possibilities of finding diverse solutions increase. In previous studies [36], they report that MIGA significantly increases diversity of the population and also has a better performance in finding the global optimal solution. In our study we are mostly leveraging the former benefit from using a segregated search algorithms.

In this Section we restrict the search space to different clusters in three different ways:

- Slicing mutation operators
- Slicing fault locations
- Slicing test cases

Slicing Mutation Operators

In this experiment we segregate the search space to a single mutation operator per execution. These non-overlapping sets then force the approach to look for a patch candidate that uses the mutation operator in each given set only.

We therefore designed and implemented full executions extending JaRFly to restrict the mutation operators used to be one mutation operator per execution. These single-mutation-operator executions use the following operators:

- Append
- Sequence Exchange
- Delete
- Param Replacer
- Param Add/Rem
- Expression Repl
- Method Replacer
- Null Check

- Castee Mutator
- Replacement
- Expression Add/Rem
- Cast Check
- Size Check
- Range Check
- Object Initializer
- Caster Mutator
- Lower Bound Set
- Upper Bound Set
- Off by One

Slicing Fault Locations

To slice fault locations we force the automated repair approach to look for candidate patches considering only a set of fault locations. For this we follow the procedure described in Algorithm 3 where we traverse the analyzed program recursively assigning node numbers to each `program statement` starting at the program head (the entire program) and stopping recursion when a statement cannot be subdivided into further statements.

Algorithm 3 to create fault location subsets

```
1:  procedure LocationSubSets(s, P)
    ▷ Produce s subsets of statements from program P to use as fault locations
2:      nodeNum ← 0
3:      codeBank = {}
4:      VisitNode(P, codeBank, nodeNum)
    ▷ Once statements are segregated recursively, create sets of locations
5:      setNum ← 0
6:      elemsPerSet ← sizeOf(codeBank)/s
7:      locationSets ← 0
8:      while setNum < s do
9:          subSet ← codeBank.getNextSet(elemsPerSet)
10:         locationSets ← locationSets ∪ {subset : setNum}
11:         setNum ← setNum + 1
        return locationSets

12: procedure VisitNode(N, codeBank, nodeNum)
13:     for all Statement s : N do
14:         codeBank ← codeBank ∪ {s : nodeNum}
15:         nodeNum ← nodeNum + 1
16:         VisitNode(s, codeBank, nodeNum)
```

Finally we segregate the search space into five equally-sized subsets and restrict the search for plausible patches to these locations only: $\frac{1}{5}codeBank$, $\frac{2}{5}codeBank$, $\frac{3}{5}codeBank$, $\frac{4}{5}codeBank$, and $\frac{5}{5}codeBank$ as described in Algorithm 3.

Slicing Test Cases

When slicing test cases, we restrict APR approaches to consider only a set of the failing test cases for the buggy programs that include more than one negative test case. This creates patches that are able to pass the analyzed set of negative test cases.

As described in Section 4.2.2 test cases may contain functional and path redundancy, therefore we create subsets of test cases based on their code coverage following the methodology described in Algorithm 2.

Results: In this section we analyze the results from the experiments executed based on our diversity driven techniques to find higher quality patched in the APR process. We start by describing the quality of the patches generated when running the APR approaches with no modifications (we call this the *standard* approach throughout this section). We later compare these executions with the diversity driven approaches described afterwards.

Standard APR Approaches:

To compare the results of the patches found by the diversity seeking approaches we first run the APR tools in their standard form without any changes to how they traverse the search space. This allows to later to compare the patches found by the diversity seeking approaches against the patches found by the canonical form of the same approaches, therefore drawing a direct comparison between the techniques and analyzing the effect the diversity-driven approaches have in patch quality. Given the computational intensity of running the diversity driven executions, we restrict our attention to a subset of the Defects4J bugs with single-line human patches following previous studies [192]. The number of bugs attempted per project varies based on how many such bugs are available in the dataset.

Table 6.1 summarizes the patches found per each APR technique. The second and third column describe the total number of patches found (including repeated) and number of unique patches found correspondingly. The next three columns depict a description of the quality (minimum, average and maximum) of the patches found in each approach. The last column describes the confidence intervals, given a $p = 0.05$, of the distribution of quality for the patches in each approach. Similar to previous Chapters in this dissertation, the quality is defined as the percentage of test cases the patch passes from the held-out test suite.

Following, we present the patches found following the proposed diversity-driven mechanisms and analyze their respective patch quality.

Slicing Mutation Operators: In this experiment we slice the mutation operators into sets of one to run our experiments therefore restricting the APR approach to look for patches using a single mutation operator.

Table 6.2 describes the patches found using each mutation operator. Column one indicates the mutation operator used. Column two, the number of bugs patched by the executions using each mutation operator. Column three and four, the total (including repeated) and unique number of patches found. Column five describes the average quality of the unique patches found, and column

Table 6.1: Patches found using the standard form of APR techniques. The second column describes the total number of patches found (including repeated patches), the third column describes the number of unique patches found correspondingly. The next three columns depict a description of patch quality (minimum, average and maximum). The last column describes the confidence intervals of the patch quality distribution of each approach ($p = 0.05$) where quality is defined as the percentage of test cases the patch passes from the held-out test suite.

	Patches		Quality			
APR Approach	Total	Unique	Minimum	Average	Maximum	Confidence Interval
GenProg	73	38	79.29	94.02	100.00	[96.01, 92.04]
PAR	54	24	85.86	94.42	100.00	[91.93, 96.92]
TrpAutoRepair	61	27	85.86	95.80	100.00	[93.88, 97.72]

six the quality of the single patch with the highest quality in the group. The following columns (seven, eight, nine and ten) represent the number of bugs patched by the standard APR techniques and a description of the patches found by the standard technique when using the mutation operator described in each row. This Table allows us to draw a direct comparison between the standard APR approach, and the diversity driven approach.

Table 6.2: Patches when slicing mutation operators against patches from standards APR approaches. Column one indicates the mutation operator used. Column two, the number of bugs patched. Column three and four, the total (including repeated) and unique number of patches found. Column five describes the average quality of the unique patches found, and column six the quality of the single patch with the highest quality in the group. The following columns (seven, eight, nine and ten) represent a similar description of the patches found by the standard APR techniques.

	Bugs	Patches		Average	Highest	Bugs	Unique	Average	Highest
Operator	Patched	Total	Unique	Quality	Quality	Patched	Patches	Quality	Quality
Append	5	36	19	97.94	100.00	5	8	98.95	100.00
Delete	3	60	12	95.29	99.07	3	10	94.70	99.07
Replace	6	71	52	93.84	100.00	5	20	92.07	99.07
Exp. Remove	5	100	5	99.30	100.0	3	3	99.98	100.00
Param. Repl.	1	3	1	99.93	99.93	2	2	99.08	99.93
Seq. Exch.	3	27	10	94.65	100.00	0	0	–	–
Exp. Replace	6	116	21	94.68	100.00	4	9	91.30	99.07
Exp. Add	8	129	25	95.49	100.00	4	7	91.12	100.00
Func. Repl.	4	64	4	98.63	100.00	2	2	97.26	100.00
Null Check	0	0	0	–	–	1	1	100.00	100.00

Table 6.2 shows how the diversity approach (left side of the bar) surpasses the performance of the standard APR technique (right side of the bar) by several measures. First, we compare the number of bugs patched by the diversity approach against the standard approach. If we compare the number of bugs each approach was able to patch (column two for the single mutation operator, and column seven for the standard APR approach), the single mutation operator approach found

a greater number of bugs to patch than the standard approach in six out of the ten groups of executions (green). It performed worse in two out of the ten groups (orange), and remained the same in two out of the ten groups. This demonstrates that incentivizing diversity by slicing mutation operators leads to a higher number of bugs patched in the majority of executions.

Secondly, we analyze number of patches generated for these bugs. Given that these experiments are run with several different seeds, there is a possibility that the same patch is found in different executions. Therefore, we provide both quantities, the number of "unique" patches which are different patches, and "total" patches which is mostly informational to understand in how many of the executions did the APR approach found a patch, but it includes repeated patches. The quality analysis is performed on unique patches only.

If we compare the number of unique patches from the diversity driven approach (column four) against unique patches from the standard approach (column eight), Table 6.2 shows that for eight out of the ten groups of executions the diversity approach outperformed the standards approach by creating a greater number of patches; in two out of ten, then standard approach generated one unique patch more than the diversity approach.

Third, we analyze quality by comparing average patch quality of the diversity approach (column five) against average patch quality of the standard approach (column nine). In this case, six out of ten groups of executions performed better under the diversity approach than the standard approach and two out of ten groups performed worse.

Finally, we compare highest quality patch from both groups (column six from the diversity approach against column ten from the standard approach). Analyzing the highest quality patch allows to reason about not only how a large number of patches behave when aggregated, but if we were able to find the single patch that behaves the best out of a group of patches, how would our approach perform in that case. In this scenario, the diversity approach outperformed the standard approach in two out of the ten groups of executions. The standard approach was not able to perform better than the diversity driven approach in any case for the highest quality patch.

Through this analysis, we are able to find that the slicing mutation operators approach allows for a number of advantages over the standard approach. Slicing mutation operators allows the APR tool to search in depth for patches well within each particular mutation operator slice, thus finding more and higher quality patches. As seen in Table 6.2, our approach outperformed the standard approach in number of bugs patched, number of unique patches found, average patch quality, and highest quality patch for the majority of the groups of executions analyzed.

Slicing Test Cases:

For this experiment we restrict our APR approaches to consider subsets of failing test cases to guide the traversal of the solution space. Overall, the results found for all patches are described in Table 6.3, which describes the total and unique number of patches found in columns two and three. Column four, five and six depict quality attributes of these patches, and column seven details the confidence interval of the quality distribution in each APR approach with $p = 0.05$.

In Table 6.4 we analyze the quality of patches found per test case slice. In this experiment the slices of test cases are created based on the coverage of subsets of test cases in the code to be patched. Column two of Table 6.4, describes the coverage target for each slice. The number in this column represents a percentage of the delta between the highest possible coverage for each bug (the entire guiding test suite) and the lowest possible coverage for that bug (only the failing

Table 6.3: Patches when slicing test cases. Columns two and three show the total and unique number of patches found. Column four, five and six describe quality attributes of these patches, and column seven details the confidence interval of the quality distribution in each APR approach with $p = 0.05$.

	Patches		Quality			
APR Approach	Total	Unique	Minimum	Average	Maximum	Confidence Interval
GenProg	1,977	791	78.28	94.46	100.00	[94.23, 94.68]
PAR	967	104	51.82	93.52	100.00	[93.08, 93.95]
TrpAutoRepair	1,308	496	79.29	96.03	100.00	[95.77, 96.29]

test cases from the guiding test suite).

For example, if for a particular bug the entire guiding test suite has a coverage of 80%, and the coverage of the failing test cases of the guiding test suite is 30%, then the delta between the highest and the lowest possible coverages is a 50% coverage. Therefore, in this example, the test case slice marked as "20" in column two of Table 6.4, would represent a the the slice containing the patches generated using the minimum coverage plus a 20% of the delta. Thus, the patches generated using a guiding test suite consisting of test cases with 30% coverage. Similarly, the rows with "100" would represent the patches generated with a test suite targeting the minimum coverage (30%) plus 100% of the delta (50%); thus 80% coverage.

Table 6.4: Patch quality analysis based on test case slices. Column one show the APR approach used, column two represents the percentage of coverage delta in each test case slice. Column three, four and five show the average quality of patches within that slice, the standard deviation of quality within that slice and the confidence interval with $p = 0.05$ within that slice.

APR Approach	Percentage of Delta (%)	Average Quality	Std Dev	Confidence Interval
GenProg	20	94.90	4.98	[94.44, 95.35]
GenProg	40	94.54	4.82	[94.07, 95.01]
GenProg	60	94.47	4.87	[93.99, 94.95]
GenProg	80	94.48	4.98	[94.00, 94.96]
GenProg	100	93.65	5.49	[93.03, 94.27]
PAR	20	93.95	5.57	[93.24, 94.67]
PAR	40	92.99	5.32	[92.17, 93.82]
PAR	60	93.93	7.09	[92.98, 94.89]
PAR	80	94.05	7.13	[93.08, 95.01]
PAR	100	92.04	9.16	[90.56, 93.51]
TrpAutoRepair	20	95.49	4.96	[94.96, 96.02]
TrpAutoRepair	40	96.03	4.85	[95.43, 96.62]
TrpAutoRepair	60	96.65	4.16	[96.15, 97.16]
TrpAutoRepair	80	96.41	4.37	[95.87, 96.95]
TrpAutoRepair	100	95.64	5.37	[94.90, 96.39]

Table 6.4 shows that, contrary to expectation, there is little difference in quality between the

slice with the lowest coverage of test cases and the highest. For GenProg and PAR, the average quality of the patches in the slice with the lowest coverage was higher than the slice with the highest coverage. Furthermore, in both approaches the lower bound of the confidence interval of the patches in the slice with the lowest coverage is still higher than the average quality of the patches in the slice with the highest coverage. This provides evidence that guiding test suites with a low coverage create similar patches to guiding test suites with a higher coverage.

As part of this experiment, to further show the advantages of patch diversity, we searched the bug for which the standard approach performed the poorest (the bug with the lowest patch quality across all patches found for that bu), and we analyzed if our diversity driven approach was able to find a better patch for the same bug.

For all three approaches, the patch with the poorest performance was Math 85 where the highest quality patch found for this bug passed 85.86% of the held out test suite. In comparison to how our approach performed in this same bug, the slicing test cases approach was able to find a patch that passes 92.93% of the held out test suite. Often, in the bugs where the standard approach underperforms, creating low quality patches, we can find higher quality patches generated by our diversity driven approach as described in the example before. This provides evidence that the diverse nature of our approach expands the reach of possible solutions found and within those solutions, we often find higher quality patches that outperform the standard approach.

Slicing Fault Locations:

In this experiment we segregate the fault locations into five slices and limit the APR approaches to find patches on each slice per execution. Table 6.5 describes at a high level the patches found and their corresponding quality attributes.

Table 6.5: Summary of patches generated when slicing fault locations. Column one, two and three show the APR approach, and the total and unique number of patches found. Column four, five and six display the quality attributes of said patches, and column seven depicts the confidence intercal for the distribution of patches through each approach.

	Patches		Quality			
APR Approach	Total	Unique	Minimum	Average	Maximum	Confidence Interval
GenProg	98	52	85.86	96.03	100.00	[94.70, 97.36]
PAR	54	34	85.86	94.27	100.00	[92.14, 96.40]
TrpAutoRepair	109	40	85.86	95.90	100.00	[94.29, 97.51]

Table 6.6 describes more in detail the patches found in each of the location slices and compares the quality of the patches found in this approach against the patches located in the same slices by the standard approaches. Column one, two and three describe the APR approach, project and bug number. Column four describes the slice number (one to five) where each slice contains one fifth of the possible program locations to modify. Column five, six, seven and eight show the total number of patches, unique number of patches, average and highest quality of the unique patches found. Finally on the right side of the bar, columns nine, ten and eleven display the unique number of patches, average and highest quality of the patches from the standard approach.

The three columns that best show a comparison in quality between the diversity approach and the standard approach are the columns that display number of unique patches (column six for the

Table 6.6: Detailed description of slicing fault locations. Column one, two and three describe the APR approach, project and bug number. Column four describes the slice number (one to five). Column five, six, seven and eight show the total number of patches, unique number of patches, average and highest quality of the unique patches found. Finally on the right side of the bar, columns nine, ten and eleven display the unique number of patches, average and highest quality of the patches from the standard approach.

				Patches		Average	Highest	Unique	Average	Highest
Approach	Project	Bug	Slice	Total	Unique	Quality	Quality	Patches	Quality	Quality
GenProg	Lang	51	4	8	6	100.00	100.00	2	100.00	100.00
GenProg	Math	85	5	20	3	85.86	85.86	8	85.04	85.86
GenProg	Math	80	1	2	2	92.31	97.04	1	98.42	98.42
GenProg	Math	80	2	10	9	98.07	98.42	3	94.80	98.42
GenProg	Math	80	3	20	17	98.05	98.22	9	98.14	98.42
GenProg	Chart	1	5	20	8	99.06	99.07	6	99.06	99.07
GenProg	Lang	43	1	0	0	–	–	1	100.00	100.00
GenProg	Lang	43	5	18	7	87.06	87.18	5	87.01	87.18
PAR	Lang	51	4	3	3	100.00	100.00	2	100.00	100.00
PAR	Math	85	5	20	12	85.86	85.86	8	85.86	85.86
PAR	Math	80	2	0	0	–	–	1	98.22	98.22
PAR	Math	80	3	8	8	98.05	98.22	0	–	–
PAR	Chart	1	5	13	8	98.90	99.07	2	99.00	99.07
PAR	Math	75	3	10	3	99.75	100.00	1	100.00	100.00
TrpAutoRepair	Lang	51	4	16	9	100.00	100.00	0	–	–
TrpAutoRepair	Math	85	5	20	2	85.86	85.86	3	85.86	85.86
TrpAutoRepair	Math	80	1	2	2	89.45	91.32	0	–	–
TrpAutoRepair	Math	80	2	11	5	98.42	98.42	1	98.42	98.42
TrpAutoRepair	Math	80	3	20	13	98.06	98.22	12	98.03	98.03
TrpAutoRepair	Chart	1	5	20	3	99.07	99.07	4	99.01	99.07
TrpAutoRepair	Lang	43	5	20	6	86.89	87.18	3	86.89	87.18
TrpAutoRepair	Lang	43	1	0	0	–	–	1	100.00	100.00

diversity driven approach and column eight for the standard approach) and average quality of the unique patches found.

A greater number of unique patches is beneficial given that we are interested in diversity, and the more patches we have, the higher the probability that we have more diverse patches. For the bugs where our diversity approach was able to find patches, our approach found a greater number of patches in 16 out of the 22 groups of executions in this experiment (shown in green in column six). There were six groups for which our approach generated a smaller number of patches and there no cases where both generated the same number of patches.

Higher quality is also crucial in our experiment and throughout this thesis. Columns eight and nine describe the average quality and the patch with the highest quality among the population. Our diversity driven approach found a higher average quality in five of the execution groups, and a lower average quality in four execution groups. In seven groups, the average quality remained the same among both our diversity approach and the standard approach, and there are six instances where just one approach(the standard approach or our approach) and therefore the groups can not be compared.

When comparing the patch with the highest quality from each group, our approach generated

a patch with higher quality than the patch with the highest quality than the standard approach in one group of executions, while in two groups the highest quality patch found was worse than the one from the standard approach. Six of the groups can not be compared for the reason described above, and the remaining thirteen groups of executions (the vast majority) had the same highest quality patch both in the standard approach and in our approach.

The standard approach found patches for bugs for which the diversity approach did not find patches and they are omitted from Table 6.6 given that we focus on analyzing the patches for which our diversity approach was able to find patches. The standard approach found patches for three bugs more than the ones our diversity approach did in GenProg, nine more bugs in PAR and three more bugs in TrpAutoRepair.

Overall our approach shows a higher number of unique patches in the vast majority of execution groups and a higher average quality in the majority of execution groups. This implies that even though, the highest quality patch found was in most cases the same as the standard approach our approach is able to find a bigger population of higher quality patches, which benefits diversity of patches.

There were two bugs for which our approach was able to find patches across several location slices (Math 80 and Lang 43). Even though the distribution of patches among different slices is similar our approach and the standard approach, our approach was able to consistently find more patches for each bug.

Semantic-Driven Approach: In this experiment we include the semantic difference measurement described in Section 6.2.1 as part of the objective function and decreases the correctness component of in the objective function.

Table 6.7 shows a summary of the patches generated using this approach. Column one and two describe the bug from Defects4J being patched. Column three and four show the total and unique patches generated. Only unique patches are analyzed, following previous analysis in this dissertation. Columns five and six report the average quality of the patches found for each bug and the highest quality patch found within the group of patches. The right side of the bar shows the number of unique patches, average quality and highest quality patch found by the standard approach to draw a direct comparison between the patches found by our approach and the patches found by the standard technique.

Table 6.7: Column one and two describe the bug being patched. Column three and four show the total and unique patches generated. Columns five and six report the average quality of the patches found for each bug and the highest quality patch found within the group of patches. The right side of the bar shows the number of unique patches, average quality and highest quality patch found by the standard approach.

Project	Bug	Total Patches	Unique Patches	Average Quality	Highest Quality	Unique Patches	Average Quality	Highest Quality
Math	85	20	4	83.96	85.86	8	85.04	85.86
Chart	1	4	4	99.00	99.07	6	99.06	99.07
Math	80	5	4	98.18	98.42	13	97.39	98.42
Lang	43	1	1	86.32	86.32	6	89.17	100

Table 6.7 shows that for every bug patched, the semantics driven approach was able to find a smaller number of unique patches (column four) than the standard approach. Column five describes that for three out of four bugs, the average quality of the patches generated by the semantic diversity approach is lower than the standard approach. In one case (Math 80) the average quality of patches found was higher than the standard approach. Finally, column six describes how the highest quality patch found by the semantic diversity approach was as good as the highest quality patch found by the standard approach for three out of the four bugs patches. This implies that for most cases, the diversity driven approach is still able to find a patch that is as good as the best patch found by the standard approach.

Answer to Research Question 11: Slicing mutation operators was specially favorable for patch quality by finding patches for a greater number of bugs, a higher number of unique patches, higher average patch quality, and highest quality patch for the majority of the groups of executions analyzed.

Some forms of software diversity are beneficial for patch quality in APR. Our experiments show that slicing mutation operators is particularly beneficial for patch quality in the automated program repair process by finding better patches in aggregate and finding better highest quality patches when compared to the patches found by the standard approach. Other forms of diversity found a higher number of patches (slicing test cases) with a slight improvement in patch quality but at the cost of having a much higher overhead given that the creation process of the test suite subsets is complex and time consuming.

The semantic diversity approach was the technique that underperformed the most by finding a smaller number of unique patches and overall a lower average quality for the patches found, also taking into consideration that the approach itself slows down the patching process substantially given the need of creating test suites mid process to compute the semantic score. A good trait of this approach was that the highest quality patch found matches the quality of the highest quality patches found by the standard approach in most cases and for one of the cases the average quality of the patches found was higher than the standard approach.

Finally, when slicing fault locations, our approach created a higher number of unique patches in the vast majority of execution groups, and a higher average quality in the majority of execution groups. Even though the highest quality patch found by our approach displays the same quality as the highest quality patch found by the standard approach in most cases, our approach is able to find a bigger population of higher quality patches, which benefits diversity in the patch population.

6.3 N-Version Patch Quality

In this Section we perform an experiment showing how patch consolidation affects software quality. For this we use the corpus of plausible patches described in Section 4.1 and consolidate the generated plausible patches using the off-the-shelf software merging tool JDime [11] in the patches generated using the three different APR approaches implemented in JaRFly (GenProg, TrpAutoRepair, and PAR).

Research Question 12: How does patch diversity relate to patch quality when plausible patches are consolidated in relation to their non-consolidated counterparts?

Methodology:

Similar to the previous research question, for this Section consider for each bug b_i the set of plausible patches P_i that pass the guiding test suite g_i^k, and the set of patches p_i^j that compose P_i where i describes the bug being fixed, k describes the test number, and j the patch number within the set accordingly.

In this Section we create a set of n-version patches by combining plausible patches p_i^j in P_i. Thus, we create the set of plausible patch combinations C_i composed of patches $c_i^{n,m}$ where n and m describe the patches p_i^n and p_i^m being combined, where $n \neq m$.

To combine plausible patches, we use two merging mechanisms provided by the off-the-shelf code merging tool JDime [11]. These merging techniques provide insight regarding what is the best way to merge patches to get high quality fixes. The two merging techniques are described below:

- Line-based: syntactic- and language-agnostic merge based on line comparison between two files. This approach is similar to what is used by the `diff`[2] command and GitHub merge.
- Structured: language dependent merge based on the abstract syntax tree structure of the program. This approach is much more resource intensive than line-based.

Finally, we compare the quality of the n-version patches $c_i^{n,m}$ to their corresponding individual plausible patches p_i^n and p_i^m by executing the held-out test cases h_i^l from test suite H_i both in the n-version patch $c_i^{n,m}$ and its corresponding individual plausible patches p_i^n and p_i^m.

Results:

Figure 6.9 describes the behavior of the consolidated n-version patches. The light green bar shows the percentage of n-version patches $c_i^{n,m}$ whose quality is *higher* than *at least* one of their corresponding patches p_i^n and p_i^m, while the dark green bar shows the percentage of n-version patches $c_i^{n,m}$ whose quality is *higher* than *both* p_i^n and p_i^m.

Similarly, the light yellow bar describes the percentage of n-version patches $c_i^{n,m}$ whose quality is *lower* than *at least* one of their corresponding patches p_i^n and p_i^m, and the dark yellow bar shows the percentage of consolidated patches $c_i^{n,m}$ whose quality is *lower* than *both* p_i^n and p_i^m.

In Figure 6.9, colored bars follow the left y-axis. Up to 53% of the consolidated patches show higher quality than at least one of their corresponding individual plausible patches, and up to 36% of the consolidated patches show higher quality than both of their corresponding individual plausible patches. This shows that there is a considerable potential in patch consolidation as a means to improve plausible patch quality.

The black horizontal lines in Figure 6.9 follow the description of the right y-axis, and illustrate the number of n-version patches (combinations) generated by each approach and code merging mechanism.

An interesting behavior we notice in this experiment is that the percentage of n-version patches whose quality *increase* by consolidating in comparison to their corresponding single patches is

[2]https://www.unix.com/man-page/all/1/diff/

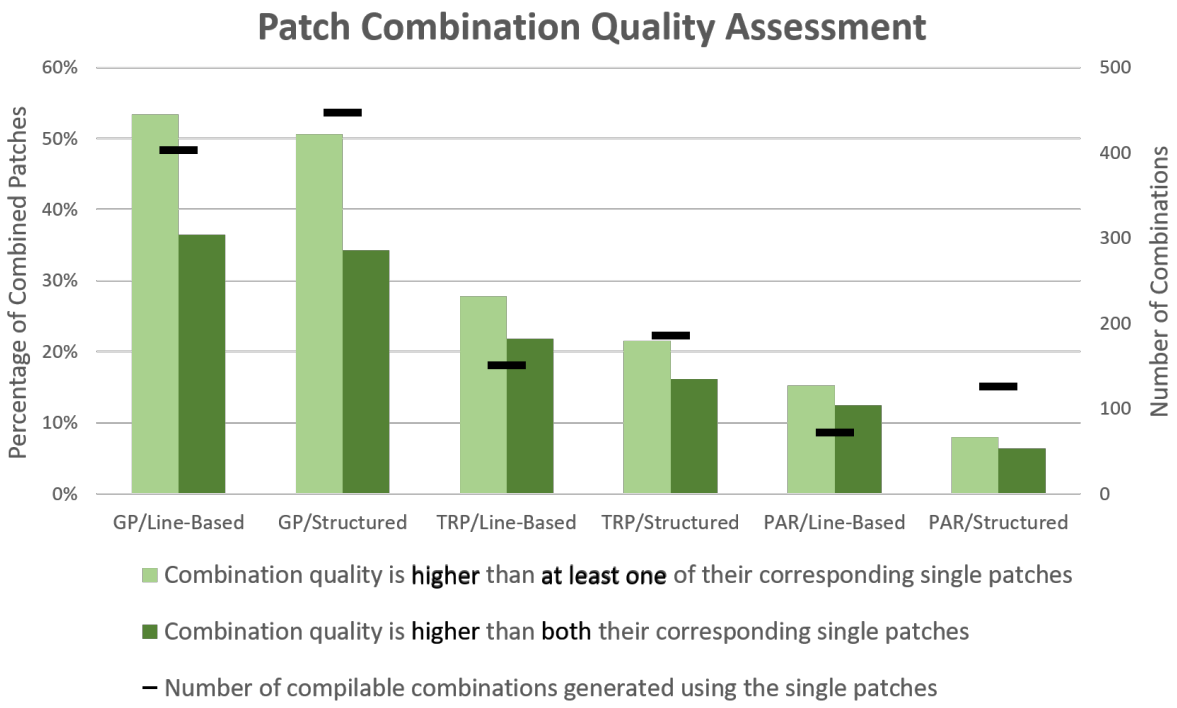


Figure 6.9: Quality assessment of patch combinations and their corresponding individual plausible patches using three APR approaches: GenProg (GP), TrpAutoRepair (TRP) and PAR, and two merging mechanisms: Line-based and Structured. The bars describe the quality of the combinations using the left axis, the dashes describe the number of combinations created using each merging mechanism using the right axis.

higher for GenProg plausible patches. From the three techniques analyzed, GenProg is the tool who can create the most diverse patches given its ability to use crossover over several mutation operators. The percentage of quality increase of GenProg patches is followed by TrpAutoRepair plausible patches. TrpAutoRepair uses the same high-level mutation operators as GenProg therefore providing still a way to create diverse patches, but it restricts the generated plausible patches to a single mutation operator per patch, therefore decreasing the level of diversity allowed.

Finally, PAR is able to only use a set of very specific fine-grain code changes, therefore it's ability to generate diverse patches is reduced even further. Thus, as noticed in Figure 6.9 the probability for patch consolidation to increase the quality of plausible patches is directly proportional to the APR technique's ability to produce diversity in their plausible patch pool. This behavior is consistent with the results from our previous research question (Section 6.1) and with previous work [189].

Answer to Research Question 12: Consolidated patch quality is proportional to the APR technique's ability to generate diverse plausible patches.

This Chapter shows the advantages of software diversity in the APR process and how this diversity affects patch quality. We start by evaluating the idea of a theoretical n-version patch through examining how test cases behave when evaluated in populations of patches for the same bug. We then create a series of techniques to incentivize and exploit diversity in the APR process. We analyze how each of the different techniques affect patch quality, and finally we create a set of real consolidated patches and analyze their behavior and quality.

Overall we found that certain techniques, specially slicing mutation operators and slicing fault locations seem to be beneficial to increase patch quality and number of unique patches, which supports our goal of patch diversity. Similarly, we found that consolidating patches can also be used to increase patch quality and this benefits APR approaches with greater diversity.

Chapter 7

Conclusions and Future Work

7.1 Summary

A fundamental problem current automatic program repair approaches suffer is the generation of low-quality patches that overfit to the guiding test suite and do not generalize to an oracle evaluation. This dissertation describes a set of mechanisms to enhance key components of the automatic program repair process to generate higher quality patches, including an analysis of test suite behavior and how their key characteristics improve the creation of plausible patches in automatic program repair, an analysis of developer code changes to inform the mutation operator selection distribution and a statement kind selection, and a repair technique to increase diversity as a means to generated n-version and consolidated higher quality patches.

The main findings in this thesis are:

- Automatic program repair techniques are able to generate plausible patches when executed in real-world defects. Using our open-source framework JaRFly we were able to generate 68 patches for the 357 analyzed defects. Although the patches generated typically overfit to the guiding test suite (58.4% - 86.2% depending on the technique), and they often brake more functionality than they fix, which implies a very much needed improvement in patch quality by automated program repair tools.
- Fundamental test suite characteristics such as test suite coverage, size, provenance, and number of triggering test cases determine the quality of the resulting plausible patches generated by automated program repair. Specifically, APR techniques using larger test suites produce higher-quality patches. Similarly, the number of failing test cases also correlates with higher quality patches, and test suite provenance has a significant effect in repair quality. Higher-coverage test suites correlate with lower quality, although the effect size is extremely small.
- Human developers use each mutation operator at a different rate. We define how each of these edits maps to APR mutation operators and how often each is used by human developers. An automatic program repair technique making its edit selection based on this human-based distribution increases the quality of the patches generated when compared to other APR techniques.

- Most real-world high-quality patches are composed of several edits. Historically, finding multi-edit fixes has been a challenging task for automated repair techniques. In this thesis we generate an approach to inform multi-edit repair by creating rules of mutation operators that happen commonly together in human repairs, therefore highlighting a path for automated program repair to restrict the vast search space of multi-edit repair.
- Classic search-based automated software repair techniques can typically generate several patches for a single bug. However, these techniques also tend to generate patches that lack diversity, which can be used increase patch quality by create n-version software or using patch consolidation. In this thesis we analyze how current approaches typically lack the diversity necessary to generate n-version software, and we analyze how patch consolidation benefits from higher diversity in plausible patches.

7.2 Limitations

Given the research questions we proposed to answer, we created an experimental setup to be able to address them. This setup included the creation of JaRFly, our Java repair framework which we constantly modified to create new repair approach that allow for different ways to navigate the search space and therefore create higher-quality patches.

Within this framework, we also reimplemented several APR techniques that were either implemented originally targeting the C programming language [113, 169] or were never made publicly available [99]. A limitation of our framework is that it currently implements primarily these three techniques, and there are tens of APR techniques currently available.

Given the speed at which APR techniques are being proposed, it is unrealistic to try to reimplement all/most of them within our framework. Furthermore, some of these techniques do not release their tool's implementation, or only release compiled binaries [213]. Finally, some tools also have environmental changes that prevent us from using them. For example, ACS [218] was designed to work with a particular query style that directly interacts with GitHub, and GitHub has since disable such queries. More generally, a recent empirical study on Java program repair techniques found that 13 out of the 24 (54%) techniques studied could not be used, including ACS and CapGen. The techniques could not be used because they were not publicly available, did not function as expected, required extraordinary manual effort to run (e.g., manual fault localization), or had hard-coded information to work on specific defect benchmarks and could not be modified with reasonable effort to work on others [55]. That said, we made an effort to create our framework to be extensible by design making it easy for other researchers to include their new proposed APR approaches into our framework.

Another limitation in this direction is that in this dissertation, we focused on comparing our proposed approaches against *only* the three reimplemented techniques that our framework has available. This was made on purpose to maintain consistency across the dissertation. It is worth noting that in several of the publications made by the author, adviser and colleagues, we also compared against other APR approaches such as Nopol [192] and SimFix [148].

From our experimental setup, we also acknowledge limitations regarding our bug dataset. Defects4J is an extensive framework for bugs in the Java language, it contains 357 bugs and test suites from five popular open-source projects. A limitation we found is that, for example, the

majority of the Defects4J defects have a single failing test, which makes it hard to study the association between the number of failing tests and patch quality. Similarly, a lack of variability in the statement coverage of the developer-written tests makes it hard to study the relationships that involve that coverage. These shortcomings in the benchmark may reduce the strength of the results. Nevertheless, this paper has developed a methodology that can be applied to other benchmarks to further study these questions.

Similarly, to evaluate quality for the generated fixes, we created a methodology for creating high-quality evaluation test-suites. This methodology allowed us to generate evaluation test suites for a considerable portion of the bugs we were able to repair. A limitation is that we were not able to generate high-quality test suites for *all* of the bugs, therefore leaving some of the patches outside of the reported results. This limitation comes in part because of limitation within the test suite generation tools we created which cannot always fulfill every possible path in the program, and in part because of our lack of expertise in the domain of the open-source projects chosen in Defects4J. In any case, the methodology we generated can be used to create new benchmarks, and the instances of evaluation test suites we have created for Defects4J can be used for future evaluations on that benchmark in a reproducible manner.

7.3 Threats to Validity

There are several aspects in our experiments that might pose a risk to the validity of the results proposed in this thesis. In this Section we discuss in detail these threats to validity and the steps we have taken to mitigate such threats.

Internal validity:

Regarding possible errors in our implementation and experiments. There is the possibility that our implementation of APR techniques as described in this dissertation and the reimplementation of APR approaches from our framework JaRFly contains errors and inaccuracies in its source code. To mitigate this threat to validity, we have released our code which includes the source files for our proposed approaches to increase patch qualities, and the source code for the three reimplemented APR techniques.

We also make publicly available our independently-generated high-quality test suites, and mined models for scrutiny and extension by other researchers, to mitigate the risk of errors in our implementation or approach. This source code and test suites can be further analyzed and inspected to guarantee its quality and allow for extension. During the process of performing these experiments we also used and shared this code and the scripts to run these experiments among several developers, which also mitigates the risk of anti patterns and code smells associated with low quality.

External validity:

It is possible that our results will not generalize to external datasets and to real bugs. To attenuate this threat we use Defects4J, a well-established benchmark of defects in five real-world, open source Java projects. The diversity, number and real-world nature of Defects4J mitigates the threat that our study will not generalize to other defects. Defects4J is evolving and growing with

new projects, and our methodology can be applied to subsequently added projects, and to other benchmarks, to further demonstrate generalizability.

Similarly, we generated our probabilistic model from a large corpus of well-known open-source programs, covering a diversity of applications, and distinct from the dataset from which the models were evaluated.

Our objective methodology for measuring patch quality requires independently generated test suites and the quality of those test suites affects our quality measurement. We use state-of-the-art automated test generation techniques, EvoSuite [66] and Randoop [160], but even state-of-the-art tools struggle to perform well on real-world programs. To mitigate this threat, we experimented with two test generation tools and their configuration parameters, and developed a methodology for generating and merging multiple test suites.

Our test-suite-based methodology for measuring patch quality inherently overestimates the quality of patches because the evaluation test suites are necessarily partial specifications. If our methodology identifies a test that fails on a patch, the patch is necessarily incorrect; however, if our methodology deems a patch of 100% quality, there could still exist a hypothetical evaluation test the patch would fail. As a result, our conclusions are conservative and potentially a portion of the patches that we label as high-quality given that they generalize to an independent test suite might actually not generalize to an even broader specification.

Construct validity:

Regarding the suitability of our evaluation metrics, we evaluate patch quality by running the generated patches on a held-out test suite created from a human patch, which is to a certain extent a biased measure since we can not guarantee that the human created patch is perfect [190]. Nevertheless, we consider this to be a best known practice, since this way we provide an alternative to subjectively asking a biased human developer whether he/she considered the patch to be correct or not, also taking into account that given the number of patches our approaches create, using human evaluators is infeasible and less scalable. We also rely on Evosuite [66] as our test suite generation mechanism for the held-out test suite used for evaluation, and we acknowledge that the test suites created by this tool may not be perfect, nor provide full coverage for all cases. Nonetheless, this is a state-of-the-art test suite generation tool that mitigates the risk of bias in manually constructing evaluation test suites.

Overall, our methodology follows the guidelines for evaluating randomized algorithms [13] and uses repair techniques' configuration parameters from prior evaluations that explored the effectiveness of those parameter settings [99, 113, 169]. We carefully control for a variety of potential confounding factors in our experiments, and use statistical tests that are appropriate for their context. We make all our code, test suites, and data public to increase researchers being able to replicate our results, explore variations of our experiments, and extend the work to other repair techniques, test suite generation tools, and defect datasets. JaRFly repair framework is available at `https://github.com/squaresLab/genprog4java/` and our generated test suites and experimental results at `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

7.4 Discussion and Future Work

In this thesis we analyzed the problem of the low-quality plausible patches created by generate-and-validate automated program repair techniques and how they might overfit to the guiding test suite and not generalize to a broader specification. We also hypothesize, plan, execute, and validate different mechanisms to increase the quality of plausible patches or at least discarding lower-quality ones. Researchers can use the patch quality evaluation methodology and high-quality test suites we have developed to evaluate their techniques on real-world defects and demonstrate improvements over the state-of-the-art based on the results of their proposed approaches.

We observed that test-suite size correlates with higher-quality patches, and test-suite coverage correlates with lower-quality patches, though both effects are extremely small. These findings, surprisingly, suggest that improving test suites used for repair is unlikely to lead to better patches. Future research should explore if there exists other guidance developers can use to improve their test suites to help program repair produce higher-quality patches.

Controlling for fault localization strategy, the number of tests a buggy program fails is positively correlated with higher-quality patches. This is an outstanding result given that fixing a larger number of failing tests usually requires fixing more behavior (although it is certainly possible for a small bug to cause many tests to fail, and for a large bug to cause only one test to fail). One key observation is that fault localization can be a confounding factor. A larger number of failing tests can help fault localization identify the correct place to repair a defect, improving the chances the technique can produce a patch. A recent study similarly found that fault localization can have a significant effect on repair quality [6].

We found that human-written tests are, usually, better for program repair than automatically-generated ones. This suggests that automatically generating tests to augment the developer-written tests may not help program repair. However, the method of generating the tests likely matters, and future research should study that relationship. One potential approach is exploring whether new approaches that generate tests from natural-language specifications [27, 149] are helpful.

We observed that Java generate-and-validate repair techniques produce patches for more defects than C generate-and-validate repair techniques. This could be, as mentioned previously, due to the difference in compiler rules, and how C compilers are typically more permissive towards allowing the generation of patch candidates without the consideration of program semantics. For example, allowing the compilation of code added after a return statement (i.e., dead code) or appending a super constructor call anywhere in the program (when it should only be allowed in the first line of a constructor of a subclass). Future research could target understanding the differences in the languages that cause this and improving the fix space and repair strategies used by the Java repair techniques.

More broadly, one feature of this and other families of repair approaches is that they are based on a partial specification and the problem this dissertation tackles is based on APR techniques creating low-quality patches, where low-quality refers to the fact that these plausible patches can fully meet the expectations set by the guiding test suite (which is a partial specification) but not to a broader specification. Future work can look into ways to more broadly define program specification (perhaps using state machines or natural language processing) where there is more overhead in terms of programmer tasks to define said specification, but where time spent in maintainability becomes logarithmic instead of linear. Therefore the problem of APR

plausible patches overfitting might be mitigated. We see this as possible future work in this area, and understand that current engineering practices commonly include test cases as a program specification, making the approach discussed in this thesis much more immediately relevant and applicable.

In this thesis we define an initial approach towards building multi-edit repairs. These kinds of patches are more predominant in the real-world and make the biggest portion of high-quality patches. Given that multi-edit patches are generated by combining sets of single edits, the search space for said patches is naturally exponential and because of its size it is much harder to navigate than it is for single-edit patches. Possible future work in this direction might include path analysis for a given program combined with constraint-based APR to be able to create program paths with the desired conditions. Approaches have been proposed in this direction [139, 140, 153, 220] but given that the program specification is usually created from the execution of test cases, these solutions still show overfitting behavior to a similar degree than search-based approaches.

Code diversity can also be improved in several ways yet to be explored beyond this dissertation. A possible proposed approach is to incentivize syntactic difference instead of semantic difference as in this thesis. When evaluating this possibility, our opinion was that even when there is the possibility that syntactic difference might be a proxy for semantic difference between programs, in our experience it is common in APR plausible patches, to find patches that are syntactically different but semantically equal (e.g., Figure 6.7). Because of this reason we decided to go with the task of creating a measurement to approximate semantic diversity that takes longer in the beginning but is closer to measuring program functional diversity which is the goal of our approach.

In this dissertation we look into code consolidation using patch diversity as a possible way to increase patch quality. A challenge when consolidating patches is that the APR technique only has knowledge of a small portion of expected behavior (the guiding test suite) within the breadth of all possible correct executions. Therefore, given the limited knowledge, it can be used to discard consolidated patches that decrease quality even in the guiding test suite (i.e., it would not even be considered a consolidated plausible patch given that it does not pass all the test cases in the guiding test suite), but even when it does, is challenging to know which consolidations might generalize more than others or even which consolidations might decrease quality instead of increasing it.

Given the case of a large corpus of test cases in a project with considerable redundancy (which is becoming everyday more common in industrial practices), one approach could be to segregate the corpus of tests, where a majority of the corpus is used as guiding test suite and the remaining of the corpus is used to guide consolidation efforts and validate quality, similar to how in machine learning literature [174] a common practice with large corpora of data is to segregate the corpus into training data, testing data, and validation data.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, Brighton, UK, 2005.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th Pacific Rim International Symposium on Dependable Computing*, PRDC'06, 2006.

[3] 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 (TAICPART-MUTATION)*, pages 89–98, Windsor, UK, September 2007.

[4] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1427–1434, Dublin, Ireland, 2011.

[5] Robert E. Adamson. Functional fixedness as related to problem solving: A repetition of three experiments. *Journal of Experimental Psychology*, 44(4):288–291, 1952.

[6] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering (TSE)*, 2020.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases*, VLDB'94, pages 478–499, 1994.

[8] Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1066–1073, London, England, UK, July 2007.

[9] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 225–236, San Jose, CA, USA, July 2014.

[10] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Themis: Automatically testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 871–875, Lake Buena Vista, FL, USA, November 2018.

[11] S. Apel, O. LeBenich, and C. Lengauer. Structured merge with auto-tuning: balancing

precision and performance. In *International Conference on Software Engineering*, ICSE '12, 2012.

[12] Andrea Arcuri. Evolutionary repair of faulty software. In *Applied Soft Computing*, volume 11, page 3494–3514, 2011.

[13] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, Honolulu, HI, USA, 2011.

[14] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[15] Atlassian. Clover code coverage tool. `https://www.atlassian.com/software/clover`, 2016.

[16] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *International Conference on Computers, Software and Applications*, IEEE COMPSAC 77, pages 149–155, 1977.

[17] A. Avizienis, M. R. Lyu, and W. Schuetz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In *International Symposium on Fault-Tolerant Computing*, FTCS'88, pages 15–22, 1988.

[18] Algirdas Avizienis. *The Methodology of N-version Programming*. 1995.

[19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue*, 3, October 2019.

[20] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *International Symposium on the Foundations of Software Engineering*, FSE'14, pages 306–317, 2014.

[21] Ahilton Barreto, Márcio Barros, and Cláudia Werner. Staffing a software project: A constraint satisfaction approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[22] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 252–261, San Francisco, CA, USA, May 2013.

[23] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering (TSE)*, 41(4):408–428, April 2015.

[24] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 267–277, Szeged, Hungary,

September 2011.

[25] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[26] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, Amsterdam, The Netherlands, August 2009.

[27] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 242–253, Amsterdam, Netherlands, 2018.

[28] Marcel Böhme and Abhik Roychoudhury. CoREBench: Studying complexity of regression errors. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–115, San Jose, CA, CA, July 2014.

[29] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? An experiment with practitioners. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 117–128, Paderborn, Germany, September 2017.

[30] D.B. Brown, M. Vaughn, B. Liblit, and T.W. Reps. The care and feeding of wild-caught mutants. In *Joint Meeting on Foundations of Software Engineering*, pages 511—-522, 2017.

[31] Yuriy Brun, Earl Barr, Ming Xiao, Claire Le Goues, and Prem Devanbu. Evolution vs. intelligent design in program patching. Technical Report `https://escholarship.org/uc/item/3z8926ks`, UC Davis: College of Engineering, 2013.

[32] Armand R. Burks and William F. Punch. An efficient structural diversity technique for genetic programming. In *Genetic and Evolutionary Computation Conference*, GECCO, pages 991–998, 2015.

[33] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[34] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. In *IEEE Security and Privacy*, volume 12, 2014.

[35] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 782–791, San Francisco, CA, USA, 2013.

[36] Hong Chen, Ryozo Oka, and Shinsuke Kato. Study on optimum design method for pleasant outdoor thermal environment using genetic algorithms (ga) and coupled simulation of convection, radiation and conduction. In *Building and Environment*, pages 18–31, 2003.

[37] J. J. Chen. *Software Diversity and Its Implications in the N-Version Software Life Cycle*. PhD thesis, 1990.

[38] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *International Symposium on Fault-Tolerant Computing*, FTCS'78, pages 3–9, 1978.

[39] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 637–647, Urbana, IL, USA, November 2017.

[40] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 32th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 595—-604, Washington, DC, USA, 2002.

[41] Steven Christou. Cobertura code coverage tool. `https://cobertura.github.io/cobertura/`, 2015.

[42] Robert Cochran, Loris D'Antoni, Benjamin Livshits, David Molnar, and Margus Veanes. Program boosting: Program synthesis via crowd-sourcing. In *Symposium on Principles of Programming Languages (POPL)*, pages 677–688, Mumbai, India, January 2015.

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

[44] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for java (demo). In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 449–452, Saarbrücken, Germany, 2016. ACM.

[45] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31:446—465, 2005.

[46] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE) short paper track*, pages 550–554, Auckland, New Zealand, November 2009.

[47] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)*, pages 383–401, Toronto, ON, Canada, July 2016.

[48] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. NoFAQ: Synthesizing command repairs from examples. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages

582–592, Paderborn, Germany, September 2017.

[49] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1443–1450, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation (TEVC)*, 6(2):182–197, April 2002.

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

[52] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the dynamic detection of functionally similar code fragments. In *European Conference on Software Maintenance and Reengineering*, 2012.

[53] Aritra Dhar, Rahul Purandare, Mohan Dhawan, and Suresh Rangaswamy. CLOTHO: Saving programs from malformed strings and incorrect string-handling. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 555—-566, Bergamo, Italy, 2015.

[54] Zhen Yu Ding, Yiwei Lyu, Christopher Timperley, and Claire Le Goues. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *International Workshop on Genetic Improvement (GI)*, Montreal, QC, Canada, 2019.

[55] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 302–313, Tallinn, Estonia, 2019.

[56] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the Defects4J dataset. *CoRR*, abs/1505.07002, 2015.

[57] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1999.

[58] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.

[59] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. *Boa: an Enabling Language and Infrastructure for Ultra-large Scale MSR Studies*. 2015.

[60] EclEmma. JaCoCo Java code coverage library. `https://www.eclemma.org/jacoco/`, 2017.

[61] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, 2001.

[62] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, ASE'14, pages 313–324, 2014.

[63] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 965–972, July 2010.

[64] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Conference on Genetic and Evolutionary Computation (GECCO)*, pages 947–954, Montreal, QC, Canada, 2009.

[65] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering*, FSE'11, pages 416—-419, 2011.

[66] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, February 2013.

[67] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, Minneapolis, MN, USA, July 2012.

[68] Mark Gabel and Zhendong Su. Testing mined specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Cary, NC, USA, 2012.

[69] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: Testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 498–510, Paderborn, Germany, September 2017.

[70] 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 *International Conference on Software Engineering*, ICSE '15, 2015.

[71] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Automated Software Engineering*, ASE'15, pages 307–318, 2015.

[72] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–224, Saarbrücken, Genmany, July 2016.

[73] Pete Goodliffe. *Becoming a Better Programmer: A Handbook for People Who Care About Code*, page 76. O'Reilly Media, 2014.

[74] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. In *Communications of ACM*, CACM 19, pages 56–65, 2019.

[75] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Symposium on Principles of Programming Languages (POPL)*, pages 317–330, Austin, TX, USA, 2011.

[76] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 465–480, Philadelphia, PA, USA, June 2018.

[77] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. DeepFix: Fixing common C language errors by deep learning. In *National Conference on Artificial Intelligence (AAAI)*, pages 1345–1351, San Francisco, CA, USA, February 2017.

[78] Mark Harman. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 342–357, 2007.

[79] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[80] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Working Conference on Mining Software Repositories*, MSR'13, pages 121–130, 2013.

[81] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, ICSE'12, pages 837–847, 2012.

[82] Marc R. Hoffmann, Brock Janiczak, Evgeny Mandrikov, and Mirko Friedenhagen. JaCoCo code coverage tool. `https://www.jacoco.org/jacoco/`, 2009.

[83] Xingzhi Hu, Xiaoqian Chen, Yong Zhao, and Wen Yao. Optimization design of satellite separation systems based on multi-island genetic algorithm. In *Advances in Space Research*, 2013.

[84] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 12–23, Gothenburg, Sweden, June 2018.

[85] Che Shian Hung and Robert Dyer. Boa views: Easy modularization and sharing of msr analyses. In *International Conference on Mining Software Repositories*, 2020.

[86] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at Google. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 955–963, Tallinn, Estonia, August 2019.

[87] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 298–309, Amsterdam, The Netherlands, July 2018.

[88] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *International Symposium on Software Testing and*

Analysis, volume 81, 2009.

[89] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 389–400, San Jose, CA, USA, 2011.

[90] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Causal testing: Understanding defects' root causes. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Seoul, Republic of Korea, May 2020.

[91] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, FL, USA, 2002.

[92] 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 2014.

[93] 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 2014.

[94] T. Kamiya. Agec: an execution-semantic clone detection. In *International Conference on Program Comprehension*, 2013.

[95] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *International Conference on Software Engineering*, ICSE'11, pages 351–360, 2011.

[96] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference On Automated Software Engineering*, ASE'15, pages 295–306, 2015.

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

[98] J.P.J. Kelly and A. Avizienis. A specification oriented multi-version software experiment. In *International Symposium on Fault-Tolerant Computing*, pages 121–126, 1983.

[99] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA, 2013.

[100] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE'13, pages 802–811, 2013.

[101] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering (TSE)*, 12(1):96–109, 1986.

[102] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In

International Symposium on Fault-Tolerant Computing, FTCS'99, page 30, 1999.

[103] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *International Conference on Software Maintenance*, 2002.

[104] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering*, 2001.

[105] MS Krishnan and CK Prahalad. The new meaning of quality in the information age. *Harvard Business Review*, 77:109–118, 1999.

[106] D.E. Krutz and E. Shihab. Cccd: concolic code clone detection. In *Working Conference on Reverse Engineering*, 2013.

[107] S. Kulczynski. Die pflanzenassociationen der pienenen, 1927.

[108] X. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. S. Pasareanu. On reliability of patch correctness assessment. In *ACM/IEEE International Conference on Software Engineering*, ICSE'19, 2019.

[109] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. Jfix: semantics-based repair of Java programs via symbolic pathfinder. In *International Symposium on Software Testing and Analysis*, ISSTA'17, pages 376–379, 2017.

[110] 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 *Foundations of software engineering*, ESEC/FSE 2017, pages 593–604, 2017.

[111] Xuan Bach D. Le, David Lo, and Claire Le Goues. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, March 2016.

[112] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 163–163, 2018.

[113] 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 *AMC/IEEE International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Switzerland, 2012.

[114] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Representations and operators for improving evolutionary software repair. In *Conference on Genetic and Evolutionary Computation (GECCO)*, pages 959–966, Philadelphia, PA, USA, July 2012.

[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*, 38:54–72, 2012.

[117] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38:54–72, 2012.

[118] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. In *IEEE Computer*, volume 26, pages 18–41, 1993.

[119] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *International Conference on Software Engineering*, 2020.

[120] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity Poster Track*, pages 55–56, Vancouver, BC, Canada, October 2017.

[121] Yiyan Lin and Sandeep S. Kulkarni. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–247, San Jose, CA, USA, July 2014.

[122] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *International Conference on Software Engineering*, 2020.

[123] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 318–329, Hong Kong, China, November 2014.

[124] Xuliang Liu and Hao Zhong. Mining StackOverflow for program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 118–129, Campobasso, Italy, March 2018.

[125] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 727–739, Paderborn, Germany, September 2017.

[126] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, Bergamo, Italy, 2015.

[127] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference/International Symposium on the Foundations of Software Engineering*, FSE'15, pages 166–178, 2015.

[128] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 702–713, Buenos Aires, Argentina, 2016.

[129] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*,

pages 298–312, St. Petersburg, FL, USA, 2016.

[130] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, 2016.

[131] Abraham S. Luchins. Mechanization in problem solving: The effect of Einstellung. *Psychological Monographs*, 54(6):i–95, 1942.

[132] M. R. Lyu, Chen J. H., and A. Avizienis. Software diversity metrics and measurements. In *IEEE Computer Society Signature Conference on Computers, Software and Applications*, COMPSAC 1992, pages 69–78, 1992.

[133] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *International Conference on Automated Software Engineering*, 2001.

[134] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated end-to-end repair at scale. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[135] 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 (EMSE)*, 22(4):1936–1964, April 2017.

[136] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. In *Empirical Software Engineering*, ESE'15, pages 176–205, 2015.

[137] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java (Demo). In *International Symposium on Software Testing and Analysis (ISSTA) Demo track*, pages 441–444, Saarbrücken, Germany, 2016.

[138] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 129–139, Gothenburg, Sweden, 2018.

[139] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, pages 448–458, Florence, Italy, May 2015.

[140] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, ICSE'16, pages 691–701, 2016.

[141] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701, Austin, TX, USA, May 2016.

[142] MeCC: memory comparison-based clone detector. H. kim and y. jung and s. kim and k. yi. In *International Conference on Software Engineering*, 2011.

[143] Elliot Mendelson. *Introduction to Mathematical Logic (Second edition)*. 1979.

[144] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering*, ICSE'13, pages 502–511, 2013.

[145] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(12):1085–1110, December 2001.

[146] 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 *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 234–242, Hyderabad, India, June 2014.

[147] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *International Conference on Software Testing, Verification and Validation*, ICST '14, pages 153–162, 2014.

[148] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues. Quality of automated program repair on real-world defects. page to appear, 2020.

[149] Manish Motwani and Yuriy Brun. Automatically generating precise oracles from structured natural language specifications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[150] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering (EMSE)*, 23(5):2901–2947, October 2018.

[151] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) New Ideas Track*, pages 631–634, Saint Petersburg, Russia, August 2013.

[152] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Preventing data errors with continuous testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 373–384, Baltimore, MD, USA, July 2015.

[153] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 772–781, San Francisco, CA, USA, 2013.

[154] A. Jefferson Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *2006 International Workshop on Automation of Software Test*, AST'06, pages 78–84, 2006.

[155] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 19–30, Västerås, Sweden, September 2014.

[156] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved representation and genetic operators for linear genetic programming for

automated program repair. *Empirical Software Engineering (EMSE)*, 23(5):2980–3006, 2018.

[157] Vinicius Paulo L. Oliveira, Eduardo F. D. Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved crossover operators for genetic programming for program repair. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 112–127, 2016.

[158] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[159] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 815–816, Montreal, QC, Canada, 2007.

[160] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 75–84, Minneapolis, MN, USA, May 2007.

[161] Annibale Panichella, Rocco Oliveto, Massimiliano Di Penta, and Andrea De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, TSE 14, pages 358–383, 2014.

[162] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. In *Software Testing, Verification and Reliability*, pages 605–628, 2015.

[163] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209, Toronto, ON, Canada, 2011.

[164] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *International Conference on Software Engineering*, ICSE'17, 2017.

[165] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)*, 40(5):427–449, 2014.

[166] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[167] Andrew M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, International Summer School,*, APPSEM 2000, pages 378–412, 200.

[168] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *International Conference on Automated Software Engineering (ASE)*, pages 362–371, Lawrence, KS, USA, November 2011.

[169] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance*

(ICSM), pages 180–189, Eindhoven, The Netherlands, September 2013.

[170] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance*, ICSM'13, pages 180–189, September 2013.

[171] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering*, ICSE'14, 2014.

[172] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, Baltimore, MD, USA, 2015.

[173] C.V. Ramamoorthy, Y.R. Mok, F.B. Bastani, G.H. Chin, and K. Suzuki. Application of a methodology for the development and validation of reliable process control software. pages 537–555, 1981.

[174] Gopinath Rebala, Ajay Ravi, and Sanjay Churiwala. *An Introduction to Machine Learning*. 2019.

[175] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. volume 74 of *American Mathematical Society*, pages 358–366, 1953.

[176] 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 *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 404–415, Buenos Aires, Argentina, May 2017.

[177] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. 2010.

[178] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. ELIXIR: Effective object oriented program repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659, Urbana, IL, USA, November 2017.

[179] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 13–24, Montreal, QC, Canada, May 2019.

[180] Adrian Schroter, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. If your bug database could talk. . . In *International Conference on Software Engineering*, ICSE'06, pages 18–20, 2006.

[181] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *International conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 317–328, 2013.

[182] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. 15:281–312, 2014.

[183] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on*

Genetic and Evolutionary Computation (GECCO), pages 1909–1916, Seattle, WA, USA, July 2006.

[184] Sina Shamshiri, René Just, José M. Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201–211, Lincoln, NE, USA, November 2015.

[185] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. A detection framework for semantic code clones and obfuscated code. In *Expert Systems with Applications*, volume 97, pages 405–420, 2018.

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

[187] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 43–54, Portland, OR, USA, 2015.

[188] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2005.

[189] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 532–543, Bergamo, Italy, September 2015.

[190] 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 *European Software Engineering Conference/International Symposium on the Foundations of Software Engineering*, ESEC/FSE'15, pages 532–543, 2015.

[191] M. Soto and C. Le Goues. Common statement kind changes to inform automatic program repair. In *Mining Software Repositories*, MSR'18, 2018.

[192] M. Soto and C. Le Goues. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 221–231, March 2018.

[193] Mauricio Soto. Improving patch quality by enhancing key components of automatic program repair. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2019.

[194] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *International Conference on Mining Software Repositories (MSR) Mining Challenge track*, Austin, TX, USA, 2016.

[195] Marcin Szubert, Anuradha Kodali, Sangram Ganguly, Kamalika Das, and Josh C. Bongard. Reducing antagonism between behavioral diversity and fitness in semantic genetic

programming. In *Genetic and Evolutionary Computation Conference*, GECCO, pages 797–804, 2015.

[196] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 260–269, Montreal, QC, Canada, 2012.

[197] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[198] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *IEEE International Conference on Software Engineering Poster Track*, pages 180–182, Buenos Aires, Argentina, May 2017.

[199] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, 2002.

[200] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. Towards semantic clone detection via probabilistic software modeling. In *International Workshop on Software Clones*, 2020.

[201] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, 22 November 2019.

[202] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in C. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 752–762, Paderborn, Germany, September 2017.

[203] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. An investigation into the use of mutation analysis for automated program repair. In *Symposium on Search-Based Software Engineering*, SSBSE'17, 2017.

[204] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of Intern. Conf. on Automated Software Engineering*, 2018.

[205] U. Voges. *Software Diversity in Computerized Control Systems*, volume 2. 1988.

[206] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. How are functionally similar code clones syntactically different? an empirical study and a benchmark. In *PeerJ Computer Science*, 2016.

[207] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12, Portland, ME, USA, July 2006.

[208] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 481–495, Philadelphia, PA, USA, June 2018.

[209] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, Trento, Italy, 2010.

[210] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366, Palo Alto, CA, USA, 2013.

[211] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 364–374, Vancouver, BC, Canada, 2009.

[212] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, 2007.

[213] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–11, Gothenburg, Sweden, June 2018.

[214] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software (JSS)*, 83(2):188–208, 2010.

[215] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 226–236, Santa Barbara, CA, USA, 2017.

[216] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 789–799, Gothenburg, Sweden, June 2018.

[217] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhan. Precise condition synthesis for program repair. In *International Conference on Software Engineering*, ICSE'17, pages 416–426, 2017.

[218] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 416–426, Buenos Aires, Argentina, May 2017.

[219] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering (TSE)*, 2016.

[220] 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*, pages 34–55, 2016.

[221] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study

of automatic program repair on the QuixBugs benchmark. In *IEEE International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–10, Hangzhou, China, February 2019.

[222] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *27th International Conference on Program Comprehension*, 2019.

[223] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering*, 24(1):33–67, February 2019.

[224] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of Java programs via multi-objective genetic programming. ArXiv, 2017.

[225] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Software Engineering*, 2002.

[226] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.