Improving Patch Quality by Enhancing Key Components of Automatic Program Repair

Mauricio Soto

CMU-ISR-21-101
February 2021

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

Thesis Committee:
Claire Le Goues (Chair)
Christian Kästner (Institute for Software Research)
William Klieber (Software Engineering Institute)
David C. Shepherd (Virginia Commonwealth University)

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 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. 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. 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 increase the quality of the best performing patch in a patch population, and an evaluation of patch consolidation as a mechanism to increase patch quality.

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 diversity in their generated patches. We propose and evaluate a set of diversity-driven techniques that lead to an increase in semantic diversity of the patch pool and an increase in the best performing patch of the patch population. Finally, we analyze how patch consolidation can be used to increase patch quality.

Acknowledgments

I would like to 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.

Dave Shepherd for being the only interviewer in my life to ask me about Charlie Parker and Cannonball in a job interview. It just got better from there. Noice!

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 guys, like how do QBF solvers work or how to juggle five clubs while riding a monocycle.

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

Mis papás por guiarme y amarme tanto. Todo lo bueno que hay en mí viene de uds.

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 everyday. For them it's more like a nightmare with all my snoring and bear-like movement, but their happiness is my happiness.

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.

Manrique for being my first and best friend, we grew and discovered life together.

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 super-villains are Doctors 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.

Finishing a Ph.D. while working full time, raising a new born, and dealing with a world-wide pandemic has been challenging. But I have good stories to tell, how many people can say they defended their Ph.D. thesis wearing sweatpants and crocs?

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[1]

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

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

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Developers spend on average half of their time at work locating and repairing bugs in software systems; and given the pervasiveness of software, the cost and impact of these errors continues to grow every year [72, 194]. Errors in software 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 [33].

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 [117]. These bugs can also cause millions of dollars to the economy, such as the Millennium 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 [104].

Repairing bugs like these is one of the most resource-intensive tasks in software development, requiring substantial manual effort [72, 194, 208]. 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, 112, 138, 138, 152, 184, 188, 192, 206].

One family of approaches known as *heuristics-based program repair* [116] focuses on using heuristics to modify source code generating patch candidates, which are later validated by a program specification (e.g., test suite). The terms *validation* and *verification* are overloaded with different meanings. Although traditionally checking program behavior against a (partial) specification is called 'verification', in this work, in line with common terminology on generate-and-validate repair approaches, we use the term *validation* to refer to the process of checking whether the patch candidate passes the test suite. Overall, the intent of these approaches is to produce a modified version of source code that behaves accordingly to the behavior described in the provided program specification. We call this variant of source code a *plausible patch*.

The provided program specification can be a *full* specification, which entails a way to describe all possible program inputs and their matching outputs. However, it is often the case that a full specification is excessively large or infinite, and thus, a more common description of desired behavior is provided as a *partial* specification, which is a subset of the full specification. This implies that the specified intended program behavior that guides the automated repair process is commonly a *partial*, and therefore incomplete, description of the desired program behavior.

This leads to a fundamental problem with automatic program repair approaches, which is, their proneness towards generating low-quality patches. This occurs when these approaches generate a plausible patch that behaves correctly when evaluated using a provided partial specification, but not on a broader, more complete specification of the desired program [121, 149, 187] (e.g., more tests or a knowledgeable developer). This problem is typically referred to as "overfitting" to the initial (partial) specification in automated repair.

The work described in this dissertation is a 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 in quality, therefore more often behaving correctly in a wider range of scenarios than the ones described in their limited partial program specification.

1.1 Examples of Automatic Program Repair

To further understand the heuristics-based program repair 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 is then 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(**max(1,2)**,2)	2	✓
assertEquals(**max(2,1)**,2)	2	✓
assertEquals(**max(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.

The full specification of this program, as mentioned before, is infinite given that combinations of any two possible integers are infinite. If there were a full specification, it would be the combinations of all possible integers and their corresponding correct program output (the greater of the two integers). Given that the full specification is infinite, APR works with a partial specification (a subset of the full specification). In this case, the partial specification is described at the bottom of Figure 1.1 with three test cases that 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. This test case demonstrates that the program contains an error that needs to be fixed given that it does not behave properly in at least one provided case that describes desired program behavior.

Heuristics-based program repair 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 program statement to a particular location. Repair approaches typically reason about software by using an abstract syntax tree (AST) representation of the program therefore avoiding syntax errors like miss matched parentheses or indentation. 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, 112, 206].

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

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

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 less-than sign ("<") to a less-than-or-equals sign

("<=") which then modifies the behavior of the program to include cases where both inputs contain the same value and therefore patches the buggy behavior of the program. Similarly, fine-grain mutation operators as such are used by several APR approaches [99, 125, 128].

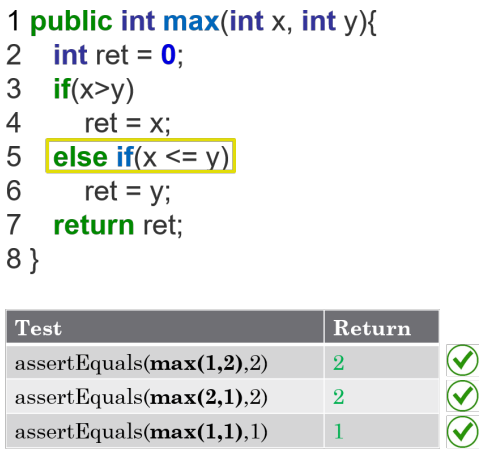


Figure 1.3: Example of a program patched by an automatic program repair technique. In this example, the "Change Condition" mutation operator is used to patch the source code. The "<" symbol is modified to a "<=" symbol creating a correct path for the case where both parameters contain the same value.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

1.2 Generation of Low Quality Plausible Patches

There are several key challenges that heuristic-based techniques must overcome to find patches [206]. First, reasoning about what is the correct portion of code that contains the error. This 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 [19, 67] 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 ways of editing in this thesis as *mutation operators* and they define the *repair strategy* of the APR technique. In this dissertation, we divide our mutation operator set in two groups. *Coarse-grain* mutation operators include *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 [112]. *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 (typically a partial specification), a plausible patch is generated. These plausible patches might fully fix the bug, making the program behave correctly in all possible cases (the unseen full specification). We call these patches *correct patches*. Creating correct patches is the ultimate goal of automated repair and in our quality evaluation, correct patches would obtain the maximum quality.

However, if the plausible patches found are not correct patches (they do not behave correctly in all possible cases), then we define the quality of each plausible patch based on how much the patch *generalizes* to further cases other than the ones described in its initial partial specification. Having a plausible patch that behaves correctly only in the provided partial specification and incorrectly in all cases outside of its partial specification obtains the minimum possible quality. Accordingly, the more cases outside of its initial partial specification the patch generalizes to, the higher its quality.

The possibility of creating low-quality plausible patches is a fundamental problem automatic program repair approaches face [121, 170, 187]. This phenomenon occurs when the approach finds a variant that satisfies the provided partial specification by the guiding test suite, but it fails to *generalize* to the intended program behavior. Guiding test suites are commonly incomplete, since they describe a portion of the full specification 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, further test cases 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 heuristics-based program repair process is a versatile set of steps that several approaches have instantiated [99, 115, 167, 206], and some of its components can be enhanced to increase the probability of the plausible patches generated by these tools to be higher-quality patches or even *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 program description presented by the provided test suite (partial specification), but it overfits to this partial specification. The patch overfits given that it will not behave correctly in further not-provided cases where the intent is to obtain the larger of two values (e.g., where both inputs x and y contain the same value and that value is different from 1).

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 *heuristics-based program repair* process. I have identified three segments in this APR methodology that can benefit from specialized improvements leading to higher quality patches. Previous studies [129,187,189] show the potential impact these components have on the quality of generated plausible patches and how enhancing the techniques used within these components can significantly improve patch quality. This document will not focus on fault localization or test suite evaluation given that these components have either

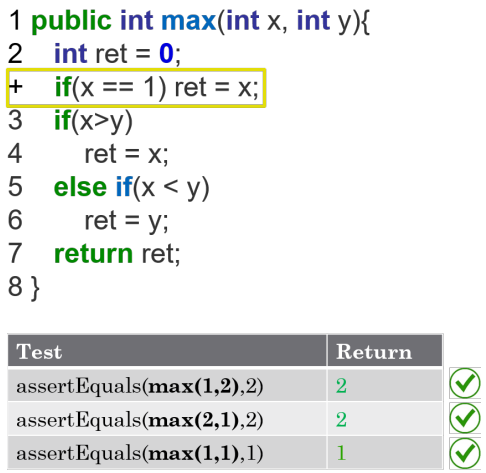


Figure 1.4: Example of a low quality patch created by an automatic program repair technique. The patch satisfies the test cases presented but does not generalize to further cases of the same bug (where both parameters contain the same value).

been extensively analyzed in the past (e.g., fault localization techniques [2,3,91,106,163]), or they focus on improving patch finding speed, not quality (e.g., test prioritization [206] and reduction of test redundancy [102]). The three main topics this dissertation will focus on are therefore summarized below:

- Guiding test suite:
  Heuristic-based program 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 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 heuristics-based program repair approaches rely on stochastic processes [115, 167, 206], therefore, it is common to obtain several similar plausible patches for a single defect. I analyze how we can incentivize software diversity to increase the quality of the best patch within a population, and how we can use patch consolidation to create higher quality patches. This study is composed of a group of diversity-driven techniques such as slicing the program specification and using multi-objective search to increase diversity. Finally, I use two consolidation techniques within a corpus of plausible patches to improve patch quality.

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

> 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 How often do heuristics-based program repair techniques produce patches for real-world Java defects?

RQ2 How often and how much do the patches produced by heuristics-based program repair techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the expected program behavior?

RQ3 How do 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 How frequently do real-world developers edit each statement kind 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 other APR approaches?

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

RQ10 How do diversity-driven techniques affect the quality of the best patch found for a given bug, and the diversity in the patch population?

RQ11 How does the quality of patches consolidated through n-version voting compare to the quality of single patches?

RQ12 How do code-merging consolidated plausible patches behave in relation to their non-consolidated counterparts in terms of patch quality?

1.3.1 Contributions

The work described in this thesis explores in-depth a set of techniques to increase patch quality in the APR process of Heuristic-based repair approaches building on prior studies on smaller programs and other target languages [29, 111, 186]. We create JaRFly, a framework for Java heuristics-based program repair techniques where we create Java versions of GenProg [112] and TrpAutoRepair [167] 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 thesis are detailed below:

- **Patch quality analysis from test suite characteristics (Chapter 4):** 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 (Chapter 4):** 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 (Chapter 3):** 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 (Chapter 5):** This dissertation provides a deeper understanding of human developer edits when fixing errors in source code, and how the analyzed APR mutation operators relate to the human edits.
- **Creation of developer-informed repair approach (Chapter 5):** 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.

- **Creation of diversity-driven patches by slicing different APR components (Chapter 6):** We propose, implement and analyze a set of slicing techniques to improve diversity in the context of APR. These diversity-driven techniques segregate different components of the repair process (fault locations, guiding test cases, and mutation operators) with the goal of increasing diversity in the patch population. This is beneficial, given that a more diverse set of patches tends to increase the quality of the best-performing patch in the population.
- **Creation of multi-objective repair approach (Chapter 6):** The implementation and analysis of a multi-objective repair approach which can optimize for several goals in the repair process. In this dissertation, this approach was used in search space traversal to optimize for correctness and diversity by using a proposed program diversity measurement with the intent of finding a more diverse pool of patches.
- **Patch Consolidation as a means to increase patch quality (Chapter 6):** The analysis of two consolidation techniques (n-version voting and code-merging) in the context of automated repair. These experiments serve as a validation of patch consolidation as a way to increase patch quality in the APR process. The study analyses which APR techniques benefit more from consolidation and the APR tool characteristics that correlate with an increase in patch quality through consolidation.
- **JaRFly, Java Repair Framework (Chapter 3):** A publicly released, open-source framework for building Java heuristics-based program 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 dissertation to run our experiments is made publicly available to support reproducibility and extension[1].

1.3.2 Potential Applications

The research implemented in this paper 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 [132]. The majority of patches generated by current approaches [99, 115, 167, 188, 216] are still low-quality patches therefore making this further research in this direction essential for further adoption of APR tools in industry.

This research 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 paper 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 experiments and studies that comprise this document. Chapter 4 analyzes the role of test suites

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

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 in the best patch of the pool; and patch consolidation as a way to increase patch quality.

Chapter 2

Background and Related Work

Automatic program repair techniques can be classified broadly into three classes based on the way they generate patches [116]: (1) *Heuristic-based* techniques create candidate patches (often via search-based software engineering [76]) and then validate these candidates, typically through testing (e.g., [8, 42, 50, 52, 96, 99, 122, 138, 152, 170, 184, 192, 206, 207].
(2) *Constraint-based* techniques use constraints to build patches via code synthesis, inferred or programmer-provided contracts, or partial specifications (e.g., [89, 164, 205]).
(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., [18, 28, 75, 118, 124, 129, 199]). In this family, commonly tests or an oracle is used to validate the patches (e.g., DeepFix [75] trains a neural network to fix compilation errors and uses a compiler to validate if its patches compile).

This thesis focuses on heuristic-based techniques given that these are not bounded by program oracles nor the underlying power and solutions provided by an SMT solver, they can fix any statement kind and they do not require a large corpus of patches. Test-driven heuristic-based techniques are a particularly interesting subject of exploration, as they have been shown to repair defects in large, real-world software [121, 149] (e.g., Clearview [165], GenProg [112], PAR [99], and Debroy and Wong [50]).

2.1 Heuristic-Based Automatic Program Repair

The heuristics-based program repair process is described in Figure 2.1. This repair approach takes as input a program with one or more bugs and a partial 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.

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

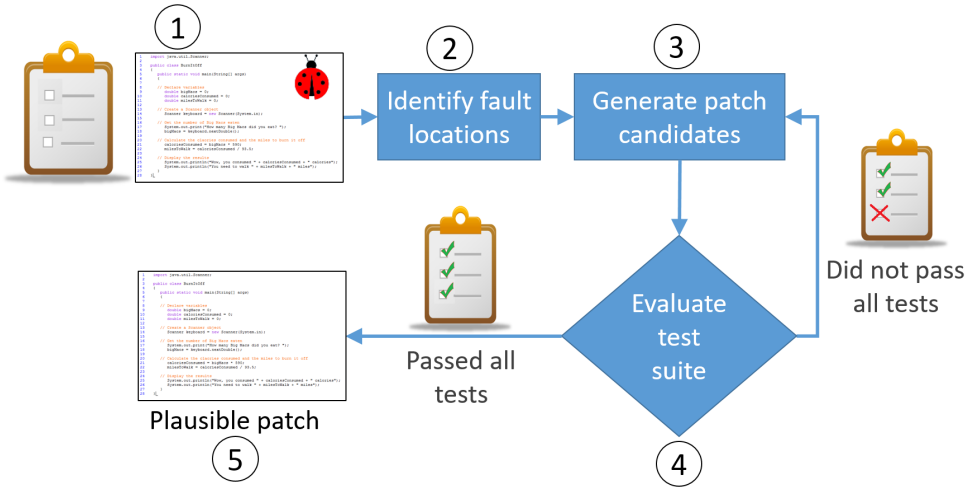


Figure 2.1: The automatic program repair heuristics-based *family to produce plausible patches starting from source code containing a bug and a test suite that describes the desired program behavior with passing and failing test cases.*

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 [115, 167], applying templates [99], transformation schemas [126, 129], or semantically-inferred code [138, 139, 215].

Finally, the approaches *validate* the patch candidates' compliance to the provided program specification (typically, a partial specification). In practice, the most common form of validation is testing (Phase 4), however, other methods of validation can be used (e.g., invariants, oracle program, human in the loop, etc.). If a variant is found that satisfies the behavior described by the provided partial specification, this variant is considered a *plausible patch* (Step 5), where "plausible" indicates that the variant passes all test cases provided [170]. Given that these plausible patches behave correctly when evaluated in a partial specification, it is possible that they fully generalize to the expected program behavior, as opposed to the variants that get discarded because they did not behave as described by the partial specification. These approaches have been successful in patching bugs for real-world software systems [18, 99, 115, 132, 188], as well as simpler software created for educational purposes [187]. Researchers have proposed several instances of this family of approaches as successful exponents of APR achievements [34, 89, 99, 112, 138, 152, 165, 206].

The concept of *plausible* patch is directly related to the core theme of this thesis, 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 is commonly a partial specification, and we refer to as the "guiding test suite" in this thesis. It is worth clarifying that there exist other ways to provide program specification and it does not necessarily need to be a test suite). Therefore, a plausible patch is guaranteed to behave correctly

in the provided partial specification, but there is no guarantee that it will behave correctly in cases from the full specification that are not included in the provided partial specification.

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. Even though patch correctness can be seen as a binary measurement where either a patch behaves correctly in all possible cases or it does not; we find the measurement of patch quality to be best described as a spectrum that depicts how close the patch is to fully generalizing to the expected behavior. This distinction is useful in real world systems, where you can reduce broken functionality of a system by having patches that fix portions of the broken functionality, even when they do not fix all possible cases the system may encounter. Patch quality also helps us reason about how to make APR approach build better patches even when they are not perfect patches.

We perform this quality evaluation by using held-out test suites as described in Section 3.3 where the main goal is to generate a partial specification of the program behavior which is independent of the guiding test suite and evaluates how much of that independent partial specification is the patch able to generalize to. In this thesis, we refer to this independent specification (which also does not necessarily need to be a test suite) as a "held-out test suite". Another complementary technique to evaluate patch quality is by using knowledgeable developers who understand the expected behavior of the program to manually evaluate the patches [133]. This quality mechanism is unrealistic to use in the experiments described in this document given the number of patches found and the knowledge required by the evaluators in these domain specific real-world systems.

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* in further cases, is commonly referred to as the patch "generalizing" to a broader spectrum of cases. This has a positive connotation and relates to high-quality patches.

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 document, target the Java programming language. Search-based approaches can be categorized within two major families: template-based or statement-edit [188] 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 [123] 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 [220] is a Java-focused repair technique that implements multi-objective search and uses NSGA-II to look for simpler repairs. Genesis [124] 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 [70] 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 [110] modifies the fitness function based on fix history to assess patch suitability. The fitness of patch candidates is determined by how similar they are to similar patches from a corpus of analyzed fixes using a graph-based representation. Prophet [129] 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 [115], PAR [99], and TrpAutoRepair [167] 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 thesis 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 [115] is an APR approach that leverages genetic programming while modifying software syntax using coarse-grained mutation operators such as delete, append, and replace. TrpAutoRepair [167, 169] traverses the search space using random search and restricts its pool of possible patches by applying a single edit 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 [187].

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 [143] 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 [126] 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 [118].

Several state-of-the-art techniques also resemble or extend GenProg, for example, AE [206] 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 [11, 13] 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 [170] 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 [69] 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. Schulte et al. [179] 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 [197] 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 [38] 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 [75] and ELIXIR [176] use learned models to predict erroneous program locations along with patches. ssFix [211] uses existing code that is syntactically related to the context of a bug to produce patches. CapGen [209] 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 [132] and Getafix [18], two tools deployed on production code at Facebook, efficiently produce correct repairs for large real-world programs. SapFix [132] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in real-time. Getafix [18] 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 [83] 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 [144, 203], finding safety violations [7], refactoring [180], and project management and effort estimation [20]. 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

To provide a full understanding of automated repair and the different families that compose it, we also summarize the main idea and techniques used in Constraint-based and Learning-based APR techniques. Different from the work performed in this thesis, constraint-based repair 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, 164]. 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 [152], DirectFix [138], QLOSE [46], Angelix [139], S3 [109], ACS [213], and Nopol [216] which use SMT or SAT constraints to encode test-based partial specifications. In this dissertation, we compare our proposed improvement mechanisms to other recent APR techniques including Nopol and SimFix.

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 [152] generates repair constraints using symbolic execution and the guiding test suite. It then solves the constraints using an SMT solver. DirectFix [138] uses partial maximum SMT constraint solving and component-based program synthesis building simpler and safer patches than SemFix. Angelix [139] 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 [138,152]. Recently a Java version was proposed called JFix [108]. QLOSE [46] is an approach that finds plausible patches by minimizing an objective function based on semantic and syntactic distances from the buggy version.

S3 [109] focuses on a programming-by-examples methodology which uses code synthesis to find plausible patches. ACS [213] targets `if` conditions, using dependency-based ordering and predicate mining. Nopol [216] 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. The original publication [188] 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 [137] infers partial specifications by symbolically analyzing a correct reference implementation instead of using test cases. Genesis [124], Refazer [174], NoFAQ [47], Sarfgen [204], and Clara [74] process correct patches to automatically infer code transformations to generate patches. SearchRepair [97] blurs the line between heuristics-based and constraint-based techniques by using constraint-based encoding of the desired behavior to replace suspicious code with semantically-similar human-written code from elsewhere.

Similarly, Learning-based APR techniques are approaches that leverage the usage of artificial intelligence over a large corpus of patches to learn and suggest future patches. An example of these techniques include learning correct code from a corpus of compilable software, these approaches then suggest a possible patch based on how similar it is to examples of other learned correct code [128]. Another family of approaches within this category include learning transformation templates from successful patches [28, 124], where the techniques learn how to modify an AST based on a corpus of transformations of previous patches.

Some of these automated repair techniques focus on a particular defect class, such as buffer overruns [183, 185], unsafe integer use in C programs [42], single-variable atomicity violations [89], deadlock and livelock defects [120], concurrency errors [122], and data input errors [8]. Our evaluation has focused on tools that fix generic bugs and do not require large datasets of bugs, correct code or patch transformations.

2.4 Empirical Studies On Automatic Program Repair

All the experiments performed in this dissertation are empirical studies of automated repair and how to improve the quality of the generated patches from our empirical experiments. We, therefore, provide background knowledge on previous empirical studies in automated repair for the reader to understand what has been achieved in this field, what is the state-of-the-art knowledge and the direction research has taken in this area. Prior work has argued the importance of evaluating the types of defects automated repair techniques can repair [148], and evaluating the generated patches for understandability, correctness, and completeness [145]. 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., [34, 45, 55, 89, 115, 134, 206, 207]), how quickly they produce patches (e.g., [112, 206]), how maintainable the patches are (e.g., [66]), 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 [164]. 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 (also called RSRepair), and AE on a 105-defect subset of ManyBugs [170], and by GenProg, Nopol, and Kali on a 224-defect subset of Defects4J showed that patch quality is often lacking in automatically produced patches [134]. 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 partial specification and fail to generalize to an independent partial specification [29, 186]. This work has led to new techniques that improve the quality of the patches [97, 125, 128, 211, 212, 219].

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 [211]. Using heuristics to approximate oracles can generate more tests to filter out 56.3% of the overfitting patches [212]. 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 [219]. Automated test generation techniques that generate test inputs along with oracles [25, 71, 147, 191] or use behavioral domain constraints [9, 68, 90, 196], data constraints [60, 150, 151], or temporal constraints [21, 22, 23, 56, 154] 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 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 us with a 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 less than one fifth of the incorrect patches, underestimating the overfitting problem [107]. 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%) [217].

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 [162]. 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 [149, 170].

Studying the improvement of patch quality of the patches generated by Angelix on the IntroClass [114] and Codeflaws [193] benchmarks of defects in small programs finds results consistent with the work presented in this thesis. By contrast, this dissertation focuses on real-world defects in real-world projects and heuristics-based program repair. Further, prior work has shown that the selection of test subjects (defects) can introduce evaluation bias [24, 166]. The evaluation technique presented in this document 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.

Overall, in the last decade automated repair has moved from a new promising topic of research to a well-established area in the software engineering community. A substantial number of new APR approaches emerge every year and new empirical studies with new benchmarks continue to arise. Empirical studies in this topic show the applicability of automated repair in different contexts demonstrating its far-reaching impact, which now even includes tech giants deploying APR techniques as part of their everyday workflow [18, 132]. However, open questions still need to be addressed to improve APR, such as how to increase patch quality of the generated patches, how to make patches more maintainable by making them more human-like, or how to speed up the patch finding process.

2.5 Defect Benchmarks

Several benchmarks of defects have evolved recently. Throughout this document 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 [114] consists of 185 C defects in real-world software. The IntroClass benchmark [114] consists of 998 C defects in very small, student-written programs, although not all 998 are unique.

The Codeflaws benchmark [193] consists of 3,902 defects from 7,436 C programs mined from programming contests and automatically classified across 39 defect classes. The DBGBench benchmark [27] (based on the CoREBench benchmark [26]) contains a collection of 70 real regression errors in four open-source C projects. The QuixBugs benchmark [119] 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 [55, 134, 148, 215]. 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 heuristics-based program repair techniques demonstrate by construction that the technique is feasible and reasonably efficient in practice [42, 97, 115, 122, 138, 152, 164, 165, 192, 205, 207]. Some show that the resulting patches withstand red team attacks [165], some illustrate with a small number of examples that heuristics-based-generated patches for security vulnerabilities protect against exploits and fuzzed variants of those exploits on typical user workloads [115], and some consider the fraction of a set of bugs their technique can repair [55, 97, 99, 112, 152].

These evaluations have demonstrated that heuristics-based program repair techniques can repair a moderate number of bugs in medium-sized programs, as well as evaluated the monetary and time costs of automatic repair [112], the relationship between operator choices and test execution parameters and success [113, 206], and human-rated patch acceptability [1, 99] and maintainability [66]. However, these evaluations have generally not used an objective metric of correctness independent of patch construction. The evaluation used in this thesis 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 [170, 192]. Evaluating the degree to which *relifix* and GenProg introduce regression errors [192] 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 [149, 170]. 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 [55, 99] and maintainability [66]. 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 [162].

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, 222]. Understanding how automated repair handles particular classes of errors, such as security vulnerabilities [115, 165] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [183, 185], unsafe integer use in C programs [42], single-variable atomicity violations [89], deadlock and livelock defects [120], concurrency errors [122], and data input errors [8]. Other techniques tackle generic bugs. For example, the ARMOR tool replaces buggy library calls with different calls that achieve the same behavior [34], 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 [164, 205] (for Eiffel code) and SemFix [152] (for C). DirectFix [138] 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 [41, 73], rather than address existing bugs. Our work has focused on heuristics-based program repair 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., [13, 50, 99, 138, 152, 157, 165, 206]).

Automated repair has been evaluated in a large set of benchmarks, showing evidence that APR techniques can generalize to different domains, programming languages, expertise level, etc. This supports the long record of success of APR techniques to create patches for difference scenarios. In this thesis, we evaluate our experiments in a benchmark that implements the generalizability these studies show, by providing a wide variety of domain-specific systems and using approaches that have been previously used for other programming languages and other expertise levels (GenProg and TrpAutoRepair).

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 [17]. It was first introduced in 1977 as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification [15]. 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 [15] and Chen [37], where they implemented NVS systems using 27 and 16 independently written versions; Ram [171] and Vog [201] 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 [37]). 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 [16]. In this thesis, 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 [16]. Because of this, identical errors were found in various versions. In this dissertation, our partial 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 [36, 130]. 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 [101]. In this thesis, 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. [160] use multi-objective genetic algorithms (MOGA) to for test case selection as a means to reduce the cost of regression testing. Szubert et al. [190] describe how increasing diversity in genetic algorithms might lead to antagonism between behavioral diversity and fitness in the context of symbolic regression. In addition, previous work [30] 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 [146, 161]. 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 [163, 198] have suggested that MBFL techniques do not significantly distinguish between faulty and non-faulty locations. Smith et al. [187] compare the performance of single patches to N-version patches, where the behavior of the N-version patches is described by a voting system. This document introduces the usage of a held-out independently created test suite as a means for measuring software quality. The work introduced in this thesis leverages these previous ideas to create real N-version patches with actual compilable code to be executed by the test cases.

2.7 State of the Practice

There are several cases where automated repair and similar techniques are currently being used in industrial practices. Probably the most prominent example is currently being implemented at Facebook with SapFix [132] where researchers describe their effort to integrate automatic program repair in a continuous integration tool. The continuous integration tool monitors test failures and automatically looks for patches that can possibly fix the errors present in the code. Once patches are found, these are presented to developers working in the project, and developers finally select which patch best fits the searched solution. This effort currently focuses on automatically repairing crashes in Android apps, however, as with other APR approaches, these repair techniques can be further extended to other contexts within industrial applications.

Similarly, other big companies have started to show industrial projects that will cause an immediate impact in automated repair. For example, Microsoft recently announced Microsoft

Security Risk Detection[1] (formerly "Project Springfield") which is Microsoft's cloud fuzz testing service for finding security critical bugs in software. These testing services based in fuzzing and the failing tests they generate can be seen as the immediate step before applying an APR approach. One can easily conceptualize the idea of using this fuzzing technique to generate failing test cases, and using the passing test cases and source code, execute APR to propose possible patches for the failing behavior found.

Google similarly implemented a similar fuzz testing service called OSS-Fuzz[2] which uses fuzz testing to uncover programming errors in software and has been used to uncover thousands of security vulnerabilities and stability bugs in Chrome components. It is worth noticing that existing APR techniques can easily produce one-line patches that deal with integer overflow and similar vulnerabilities found by OSS-Fuzz and have even successfully repaired prominent errors such as the Heartbleed vulnerability [139].

DeepCode[3] is a real-time semantic code analysis tool that suggests fixes based on large corpus of bugs and patches by using artificial intelligence and leveraging how other community members have fixed similar bugs. This industrial semantic analysis tool can be purchased and ran as an extension for IDE's such as Visual Studio or through public repositories. This APR tool currently supports Java, Python, JavaScript, TypeScript, C/C++ (beta), C# (beta), and PHP (beta).

The Repairnator project [200] also calls for attention in the area of applied automated repair. Researchers have created a bot that monitors for software errors, and automatically find fixes using repair tools. Even though the Repairnator was not created for a particular company in mind, this project takes open source projects (which can be industrial in nature and in practice) and monitors their builds with the intent of using automated repair tools to find patches when an error is presented. Different from most academic environments where APR techniques are constantly evaluated in the same or similar defects, this project regularly receives new errors from new projects, therefore making it more similar to an industrial environment where bugs are analyzed as the project is being built, and not post-mortem once the APR technique already know the solution provided by the developer.

[1] https://www.microsoft.com/en-us/research/project/project-springfield/

[2] https://google.github.io/oss-fuzz/

[3] https://www.deepcode.ai/

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 document 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 document (Section 3.1). We chose Defects4J, a dataset and extensible framework containing 357 real bugs built to support software testing research. We chose this benchmark due to its domain diversity, the fact that the projects included are real-world complex systems, and given that each bug in this framework contains its corresponding human patch, which we use to evaluate the quality of our generated patches. 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 [115] is an APR approach that uses coarse-grained mutation operators and genetic programming to generate patches; TrpAutoRepair [167, 169] uses

[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` (`https://www.dwheeler.com/sloccount/`) to measure the lines of code, reported in thousands of lines of code (KLoC). The **tests** and **test KLoC** columns refer to the developer-written tests.*

single-edits and traverses the search space using random search to create candidate patches; and 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 a 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 document to evaluate error repair. To increase patch quality we require defects on which we can test our hypothesis. For the experiments described in this thesis, 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 [54], what types of defects the techniques are able to patch [148], and the quality of the produced patches [107, 134, 212, 214]. These existing evaluations that measure patch quality use manual inspection [107, 134, 212] or automatically-generated evaluation test suites [107, 211, 214].

3.2 JaRFly: The Java Repair Framework

The second component of this experimental approach overview is the automatic program repair tools used throughout this document. 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 [112] and TrpAutoRepair [167] 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.

Figure 3.3: JaRFly is an extensible automatic program repair framework for Java programs available at `https://github.com/squaresLab/genprog4java/` [149]

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 [77]. JaRFly provides an extensible set of interchangeable pieces for each of these elements. This differentiates our framework from prior work in this area [136].

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 [136] 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.

3.2.1 Problem Representation

The way an APR approach represents the problem space affects its success and efficiency [113]. 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 a plausible patch as a sequence of edits to the original program [112, 113] minimizing the patch representation without losing syntactic information. Previous approaches [63, 207] represented patches as an abstract syntax tree (AST) making the patch representation much larger and therefore losing efficiency in the repair process. Similar problem representations have been used in the past to allow the creation of patches in other programming languages such as Python [4] and C [155, 156].

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 [112, 155, 156] 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 [48, 53, 62, 110] 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 [167, 206] have used to improve APR efficiency by prioritizing the execution of tests that are more likely to fail.

The 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 [112] and TrpAutoRepair [167]:

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 choose statements from in JaRFly is by default a segmentation of the original program being modified. Previous studies show that programming languages are repetitive [80] and fix code is more likely found within the same module and project than from foreign modules and projects. When using information from the code bank, JaRFly checks if the variables and fields used are within scope of the target location, with the intent of avoiding the creation of non-compilable patch candidates. 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. Not all *EditOperation*s can be applied in 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 cannot 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 [188].

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 choose from, including random search, weighted brute force search, oracle search, genetic programming, and NGSA-II [49], 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 [207], and crossback crossover [207]. It also includes a default tournament selection strategy. In Chapter 6 of this dissertation, 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 [39], Ochiai [2], Ample [221], Tarantula [91], Wong [210]).

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 [59].

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 [121, 187] 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 [121, 187].

Since then, research has measured the degree to which heuristics-based program repair patches overfit and what factors affect that overfitting on small C programs [187], how often these patches disagree with developer-written patches [170], how often overfitting happens in Java repair [55, 134], and what is the concentration of correct patches [127].

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 [125], learning patch generation patterns from human-written code [128], and automated test case generation [211]. Further, research has found that overfitting occurs in APR tools targeting different programming languages and repair families [111, 121] 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 [164].

The main goal of this research 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 [29, 187], and manual inspection [134, 170].

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 [66] and how likely developers are to accept them [99]). However, a recent study found that manual-inspection-based quality evaluation can still be imprecise [107].

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 [107] in potential human evaluators.

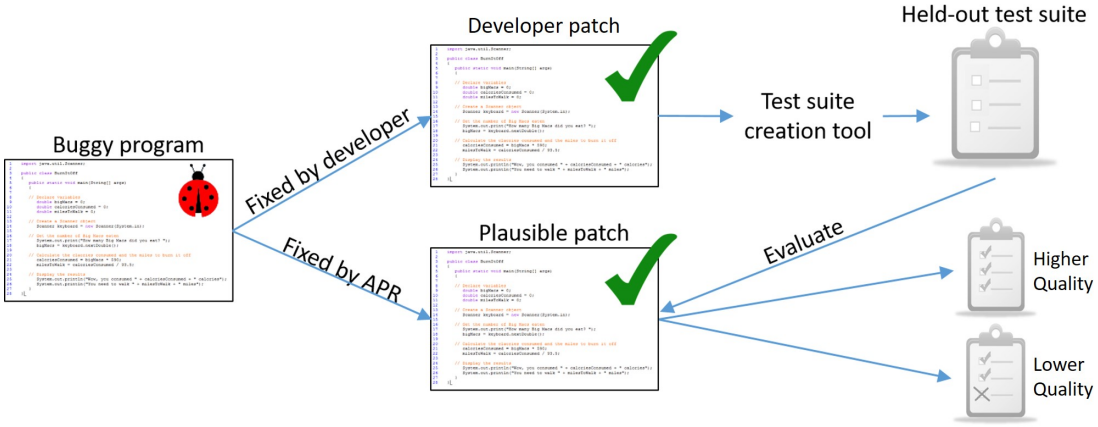


Figure 3.4: Evaluating the quality of generated plausible patches based on a held-out test suite generated from a developer patch. For each buggy program we create plausible patches using APR techniques. To evaluate their quality we generate a held-out test suite from the human-generated patch (i.e., 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 thesis, 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. The quality of a patch is proportional to the number of held-out test cases passed. The quality of a patch is defined by $\frac{T_{pass}}{T_{total}}$, as defined by prior work [186]; where T_{total} represents the total number of tests in the evaluation test suite and T_{pass} represents the number of tests passed by the patched code. A patch that passes all the tests in the evaluation test suite has 100% patch 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 [64]) 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 [85]. 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 [65]. 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 [159] and EvoSuite [65], 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 [181]. 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 34 out of the 106 defects, this algorithm generated test suites that satisfied the coverage criterion. This process is described in Algorithm 1 where $coverage(T_{eval}, covm, covc) == true$ *iff* $methodCoverage \geq covm \wedge classCoverage \geq covc$. As detailed in our description for our evaluation test suites we used the values $covm := 100\%$ and $covc := 80\%$.

Algorithm 1 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 ← {}                                          ▷ initialization an empty set
4:      runs ← 10                                            ▷ number of times EvoSuite is run
5:      timebudget ← 30                                      ▷ time budget for each run (mins)
6:      criterion ← branch/line                              ▷ criterion to optimize for in each run
7:      while runs > 0 do
8:          runs ← runs − 1
9:          T ← genTestEvosuite(timebudget)                  ▷ generate tests by running EvoSuite for 30 mins
10:         T_eval ← Distinct(T_eval ∪ 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 ← 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
```

defect set	# of defects	statement coverage of patch-modified	mean	median
at least one patch	106	methods	90.8%	100.0%
		classes	87.2%	96.3%
adequate test suite	71	methods	100.0%	100.0%
		classes	96.7%	98.7%

Table 3.1: Statement coverage of the EvoSuite-generated test suites for the 106 Defects4J defects patched by at least one repair technique in our study, and for the 71-defect subset for which our generated test suites covered 100% of all developer-modified methods and at least 80% of all developer-modified classes [149].

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 process resulted in test suites that satisfied the coverage criterion for a total of 62 defects (11 Chart, 6 Closure, 11 Lang, 30 Math, and 4 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 9 more defects (1 Chart, 3 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 71 of the 106 defects (12 Chart, 9 Closure, 14 Lang, 32 Math, and 4 Time defects). The test suites varied in size from 59 to 7,164 tests, with the mean test suite containing 1,194 tests and the median test suite 648 tests.

We restrict our study to these 71 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 71 defects is 96.7% and the median is 98.7% (with means and medians for the modified methods both 100%, as required by the coverage criterion). Table 3.1 summarizes these statistics for the 71 defects used in our study and the 106 defects patched by at least one repair technique.

Overall, in this section we presented our experimental approach overview composed of three components. First, throughout this document we use a corpus of defects called 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. Second, we created JaRFly, an open-source framework for implementing techniques for automatic repair of Java programs. The implementation includes reimplementations of GenProg [112] and TrpAutoRepair [167] for Java, and the first public implementation of PAR [99]. Finally, we present the patch quality evaluation technique we use throughout this document, which consists of creating high-quality held-out test suites using the developer patch provided by Defects4J and evaluating the automatically generated patches based on the percentage of held-out test the generated patch is able to generalize to. This experimental approach allows us to evaluate different techniques and analyze how APR approaches can be improved to increase the quality of their generated patches.

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 [187], which generate correct outputs for all test cases but where the repair does not fully address the underlying error needed to be fixed [121, 129]. This raises concerns about the usability of automated repair approaches, and outlines possible paths toward building techniques that produce higher-quality patches [96, 125, 128, 140, 188].

Prior work [187] 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 heuristics-based program repair approaches, did not control for confounding factors, and used test suite size as a proxy for coverage). Building up on previous work, in this section we answer five research questions:

RQ1 How often do heuristics-based program repair techniques produce patches for real-world Java defects?

Answer: The heuristics-based program repair approaches executed in our experiment were able to generate patches for 106 out of 357 real-world defects.

RQ2 How often and how much do the patches produced by heuristics-based program repair techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the expected program behavior?

Answer: Tool-generated patches on real-world Java defects overfit frequently to the test suite used in constructing the patch, regularly breaking more functionality than they repair.

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

Answer: For the corpus of patches analyzed, the correlation between test suite size and patch quality is statistically significant with a small effect size. Similarly, test suite coverage has a statistically significant correlation with patch quality in all but one APR approach.

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

Answer: 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: 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 heuristics-based program 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: How often do heuristics-based program repair techniques produce patches for real-world Java defects?

The first step in this study is to get a general sense of how effective automated repair approaches are 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. For SimFix, which is deterministic, we attempt the repair once for each defect using the default timeout of 5 hours, for a total of 357 attempted repairs. This culminates in a grand total of $7{,}140 \times 3 + 357 = 21{,}777$ repair attempts.

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.

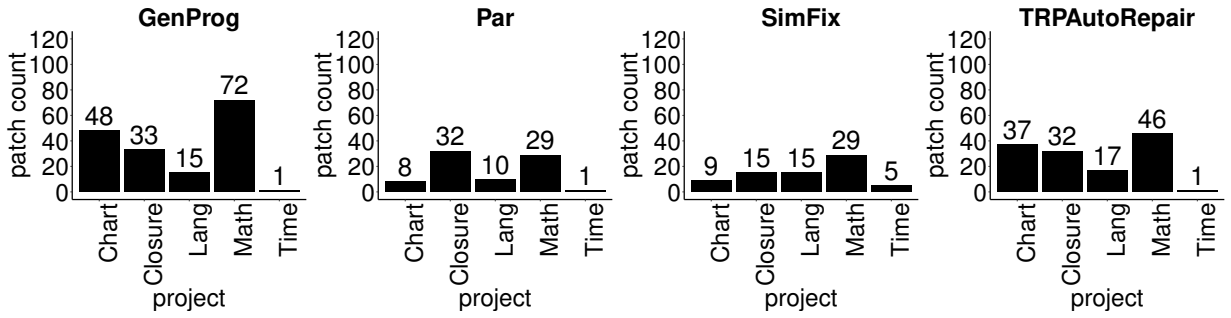
The repair techniques' parameters affect how they attempt to repair defects. For reproducibility purposes, we now describe a series of these parameters and the values used in our experiments. For GenProg, PAR, and TrpAutoRepair, we used the parameters from prior work that evaluates these techniques on C programs [99, 112, 167]. 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, 167]. For GenProg, we set `negativePathWeight` to 0.35 and `positivePathWeight` to 0.65 [113]. For all remaining parameters, we use their default values from prior work [99, 112, 167]. For SimFix, we use its open-source implementation with its default configuration [86]. We describe the complete set of parameters at `https://github.com/LASER-UMASS/JavaRepair-replication-package/wiki/Configuration-parameter-details/`.

(a) Produced patches

technique	patches total	patches unique	defects patched
GenProg	585 (8.2%)	255	49 (13.7%)
PAR	288 (4.0%)	107	38 (10.6%)
SimFix	76 (21.3%)	73	68 (19.0%)
TrpAutoRepair	513 (7.2%)	199	44 (12.3%)
total	1,462 (6.7%)	634	106 (29.7%)

Table 4.1: GenProg, PAR, SimFix, and TrpAutoRepair produce patches 1,462 times (6.7%) out of the 21,777 attempts. At least one technique can produce a patch for 106 (29.7%) of the 357 real-world defects [149].



(a) Unique patch distributions, per technique

Figure 4.1: The distributions of unique patches produced by the four techniques are similarly shaped [149].

Results: Table 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 a stochastic approach to find repairs, some of the patches found can be repeated among the different seeds, therefore we 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. SimFix patches 68 out of 357 defects (8 Chart, 15 Closure, 13 Lang, 27 Math, and 5 Time) and produces a total of 76 patches, out of which 73 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 106 out of the 357 defects. All techniques produced at least one patch for 12 defects. SimFix most often produced patches (21.3% of the attempts) and produced patches for the most defects (19.0%).

Figure 4.1 shows the distributions of unique patches, per project, generated by each of the four techniques.

Compared to prior studies on C defects [186], [114, 167], 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. Several factors affect the differences in APR behavior between the previous studies and the current one. Java compilers are much more restrictive than C compilers, and 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 heuristics-based program repair to Java defects, which found techniques to produce patches for 9.8%–15.6% of the defects [134]. 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]. Even though SimFix patches more defects (19.0%) than other techniques, the fraction

of defects patched by SimFix is still much lower (19.0% vs. 47%) than that those obtained using repair techniques for C defects.

Answer to Research Question 1: The heuristics-based program repair approaches executed in our experiment were able to generate patches for 106 out of 357 real-world defects.

4.2 Analyzing Plausible Patch Quality

Section 4.1 showed that heuristics-based program repair techniques are able to patch 29.7% 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 71 defects for which we are able to generate high-quality evaluation test suites (recall Section 3.3). These 71 defects are a subset of the 106 defects for which at least one repair technique produced at least one patch (recall Table 3.1).

4.2.1 Patch Overfitting

Research Question 2: How often and how much do the patches produced by heuristics-based program repair techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the expected program behavior?

	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%
SimFix	65.0%	96.3%	99.9%	100.0%	46.1%
TrpAutoRepair	64.8%	96.4%	98.4%	100.0%	19.5%

Table 4.2: The quality of the patches the repair techniques generated when using the developer-written test suite varied from 64.8% to 100.0%. The last column describes the percentage of 100% quality patches [149].

Methodology: After confirming the possibility of APR approaches to generate patches for real-world systems, we now analyze the quality of the patches generated by APR to understand how often do these patches overfit to the provided partial specification (guiding test suite). This will provide further understanding to the main motivation of this research: APR approaches generate patches that overfit to the provided partial specification and therefore can benefit from specialized improvement of its components to increase patch quality.

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. The quality of the patch is

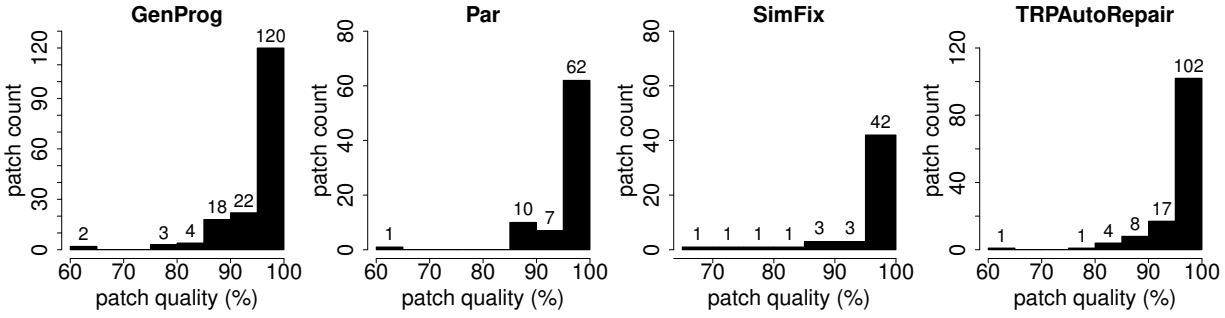


Figure 4.2: The distributions of patch quality is skewed towards the high end. Each bar represents the number of patches found in each quality bucket. The quality buckets are created with a 5% difference between them. On average, 74.1% (GenProg: 75.7%, PAR: 86.2%, SimFix: 53.9% and TrpAutoRepair: 80.5%) of the patches failed at least one test [149].

proportional to the number of held-out test cases passed [186]. A patch that passes all the tests in the held-out test suite is said to have 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: Table 4.2 and Figure 4.2 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, 74.1% of the patches (GenProg: 75.7%, Par: 86.2%, SimFix: 53.9%, and TrpAutoRepair: 80.5%), on average, failed at least one test, thus overfitting to the partial specification and failing to fully repair the defect. The mean quality of the patches varied from 95.7% to 96.4%. The relatively high fraction is not necessarily a proportional indication of the quality of repair: Defective code versions already pass 98.3% of the tests, on average, so 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 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 4.3, 4.4, 4.5, and 4.6 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 Figure 4.7 and Table 4.3, 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 SimFix, 45.8% improved, 35.5% showed no improvement, and 16.7% 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

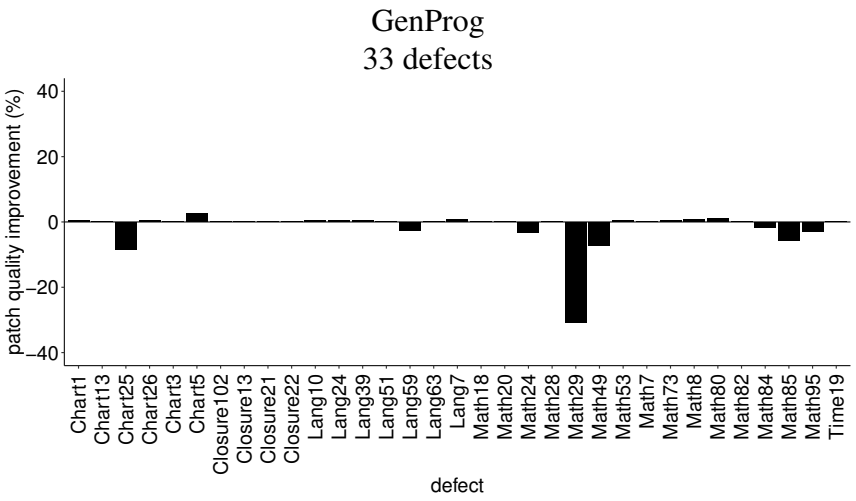


Figure 4.3: Change in quality between the defective version and the patched version of the code per each defect. GenProg created repairs for 33 defects [149].

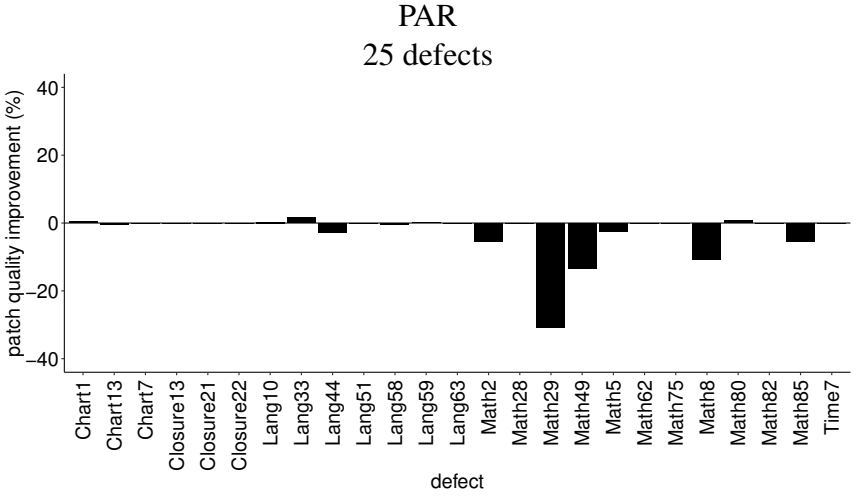


Figure 4.4: Change in quality between the defective version and the patched version of the code per each defect. PAR created repairs for 25 defects [149].

average, larger than the improvement. For all the techniques, the majority (89 out of 137, 65.0%) of the patches decrease or fail to improve quality, and more than a quarter (39 out of 137, 28.5%) of the patches break even more tests than they fix.

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%) [186].

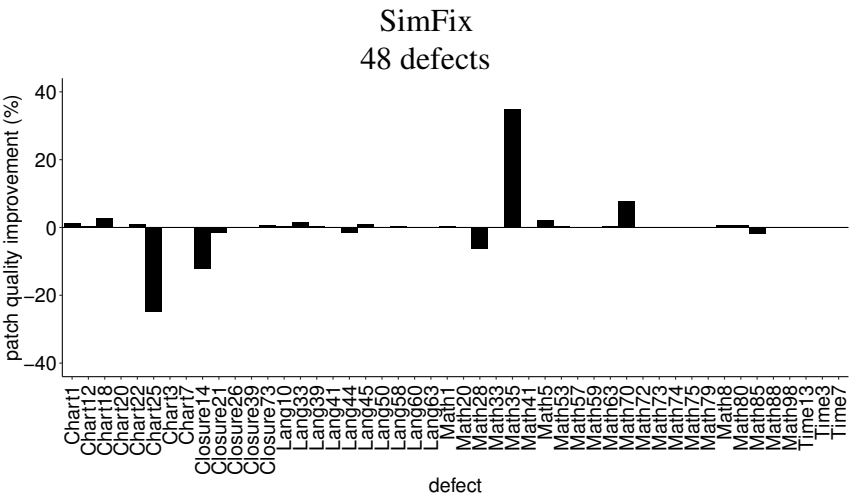


Figure 4.5: Change in quality between the defective version and the patched version of the code per each defect. SixFix created repairs for 48 defects [149].

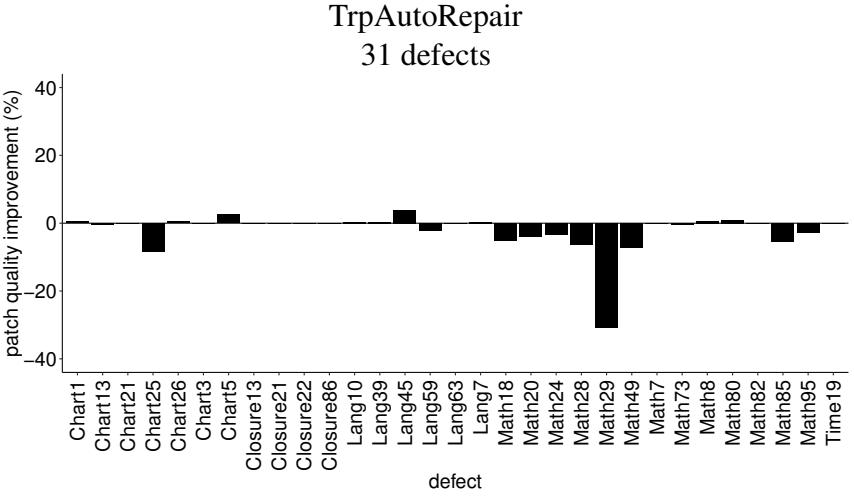


Figure 4.6: Change in quality between the defective version and the patched version of the code per each defect. TrpAutoRepair created repairs for 31 defects [149].

Answer to Research Question 2: Tool-generated patches on real-world Java defects overfit frequently to the test suite used in constructing the patch, regularly breaking more functionality than they repair.

4.2.2 Test Suite Coverage and Size

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

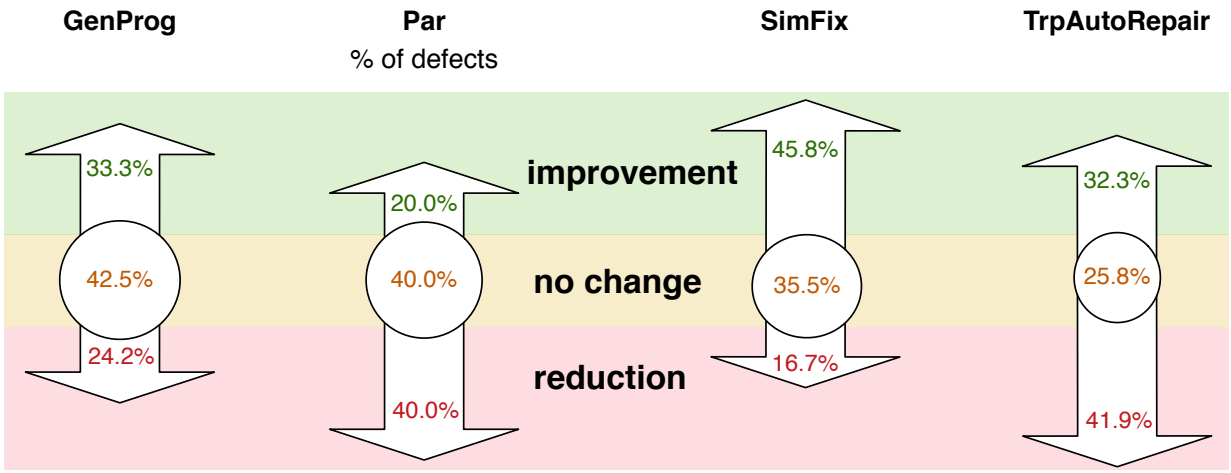


Figure 4.7: Patch overfitting. Aggregated change in quality between the defective version and the patched version of the code. The median patch neither improves nor decreases quality. While more GenProg patches improve the quality than decrease it, the opposite is true for PAR and TrpAutoRepair patches, and, on average, patches break more functionality than they repair [149].

technique	minimum	mean	median	maximum
GenProg	−30.9%	−1.7%	0.0%	2.6%
PAR	−30.9%	−2.8%	0.0%	1.5%
SimFix	−24.9%	0.2%	0.0%	35.0%
TrpAutoRepair	−30.9%	−2.1%	0.0%	3.8%

Table 4.3: 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. The data presented are for the 45 defects with high-quality evaluation test suites, of which GenProg produced patches for 33, PAR for 25, and TrpAutoRepair for 31.
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 [149].

Previous work [186] 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 71 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 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 15 defects for this reason (2 Chart, 1 Closure, 1 Lang and 11 Math) out of the 71 defects that had at least one repair technique produce at least one patch and had a high-quality evaluation test suite (recall Section 3.3).

For each of the 56 remaining defects, we choose 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.

Algorithm 2 Produce a test suite subset given a target coverage and a test suite

```
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
```

If we attempted to add a randomly selected test 500 times and failed to reach the target, we stopped, we stopped as detailed in Algorithm 2. For 11 of the 56 defects (2 Chart, 3 Closure, 1 Lang, and 5 Math), the sampling algorithm was unable to generate five distinct test suite subsets for all of the targets, so we discard these 11 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, and one time using SimFix. In total, these 23,625 repair attempts produced 9,144 patches. Figure 4.8 shows the distribution of these patches. GenProg produced at least one patch for 29 out of the 45 defects, PAR 25, SimFix 34, 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; SimFix: 6 Chart, 3 Closure, 8 Lang, 13 Math, and 4 Time; and TrpAutoRepair 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time.)

Table 4.4 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, 75.2% of the patches, on average, failed at least one test in the evaluation test suite.

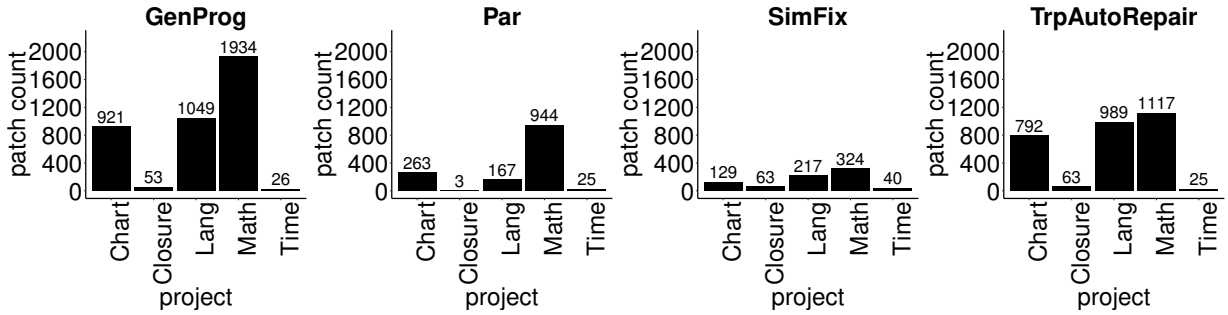


Figure 4.8: Distribution of patches generated using varying-coverage test suites. Distribution of the number of patches produced using developer-written test suite subsets of varying code coverage on the defective code version [149].

technique	minimum	mean	median	maximum	100%-quality patches
GenProg	0.0%	94.8%	98.4%	100.0%	16.2%
PAR	51.8%	91.2%	95.5%	100.0%	13.3%
SimFix	77.3%	98.4%	100.0%	100.0%	50.7%
TrpAutoRepair	62.9%	95.5%	99.0%	100.0%	19.0%

Table 4.4: 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, 75.2% (GenProg: 83.8%, Par: 86.7%, SimFix: 49.3%, and TrpAutoRepair: 81.0%) of the patches failed at least one test [149].

technique	model quality			
	p	R^2	test suite	p
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}
SimFix	4.0×10^{-16}	0.086	size	2.7×10^{-7}
			coverage	1.3×10^{-15}
TrpAutoRepair	6.9×10^{-5}	0.0057	size	1.6×10^{-5}
			coverage	0.96

Table 4.5: 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 ($p < 0.001$) except for coverage for TrpAutoRepair) [149].

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. Table 4.5 shows, for each technique, the results of the regression model. All four 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 four 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 ($p > 0.1$), and was positive for GenProg and TrpAutoRepair, but negative for SimFix and Par. We further detail each technique's regression results next.

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

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

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 SimFix, the quality of the patch is described as:

$$\text{simfix\_patch\_quality} = 98.43 - 0.04(\text{coverage}) + 0.002(\text{size}).$$

Thus, the quality of the patch produced by SimFix decreases by 0.04% for each 1% increase in the test suite coverage and increases by 0.002% for each additional test in the test suite. We observe strongly statistically significant ($p < 0.001$) associations of test suite coverage and test suite size with patch quality however, the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. 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 SimFix.

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: For the corpus of patches analyzed, the correlation between test suite size and patch quality is statistically significant with a small effect size. Similarly, test suite coverage has a statistically significant correlation with patch quality in all but one APR approach.

Overall, our results show that for the corpus of patches analyzed test suite size has a statistically significant correlation with patch quality with a small effect size. Similarly, test suite coverage shows a statistically significant correlation with patch quality for the majority of APR approaches evaluated.

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.

In this section, when we refer to *defect severity* we allude to the number of tests that fail in virtue of the defect. Therefore, if a higher number of developer-written test cases fail due to the analyzed defect, the higher its severity is.

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 71-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 four 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 [186].

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 4.9 shows the frequency distribution of failing tests across the 71 defects for which at least one of the four techniques produced at least one patch, and for which we were able to create a high-quality evaluation test suite. Of these 71 defects, only 5 defects, Chart 22, Chart 26, Closure 26, Closure 86, and Time 3, have at least five failing tests.

Figure 4.10 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 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.

Simfix only produced three patches and did not patch any of the 5 defects when using partial failing tests. Analyzing the execution logs of SimFix revealed that it was not able to localize the bug using partial failing tests. This suggests that fault localization strategy used by repair techniques could be a confounding factor when measuring the effect of the number of failing tests on patch quality. (Recall that SimFix and JaRFly use different fault localization techniques.)

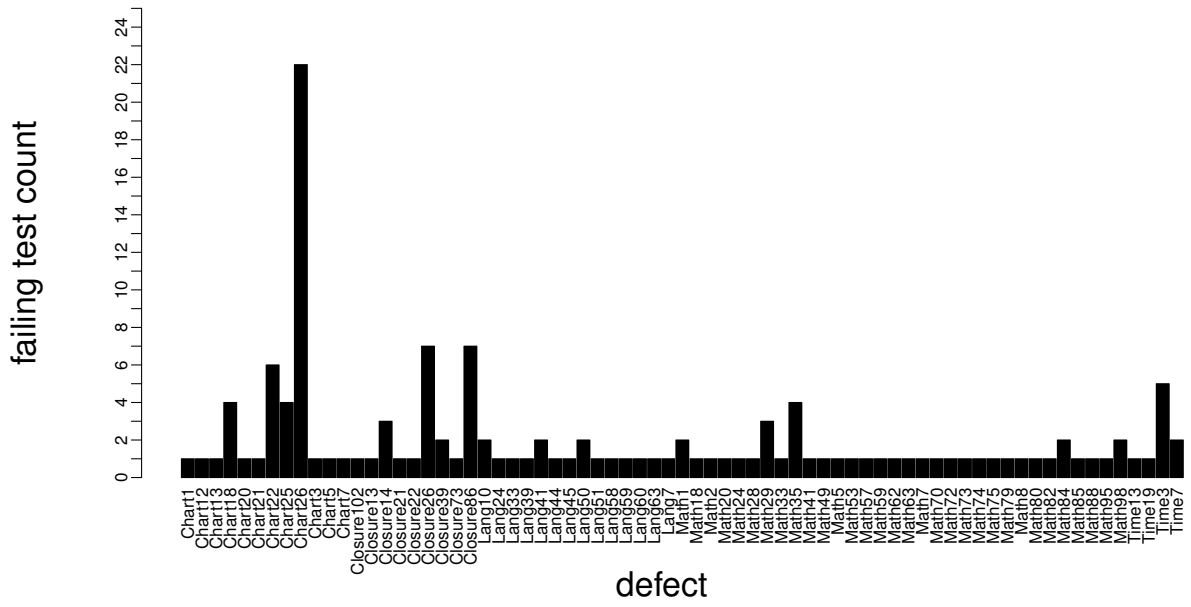


Figure 4.9: Defect severity. The distribution of the number of failing tests in the 71 defects for which at least one repair technique produces at least one patch and has a high-quality evaluation test suite [149].

Answer to Research Question 4: 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 [186, 211, 219]. 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 of the same partial specification [137], or extracted from comments or natural language specifications (e.g., Swami [147], Toradacu [71], Jdoctor [25], or @tComment [191]).

However, a previous study [186] 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

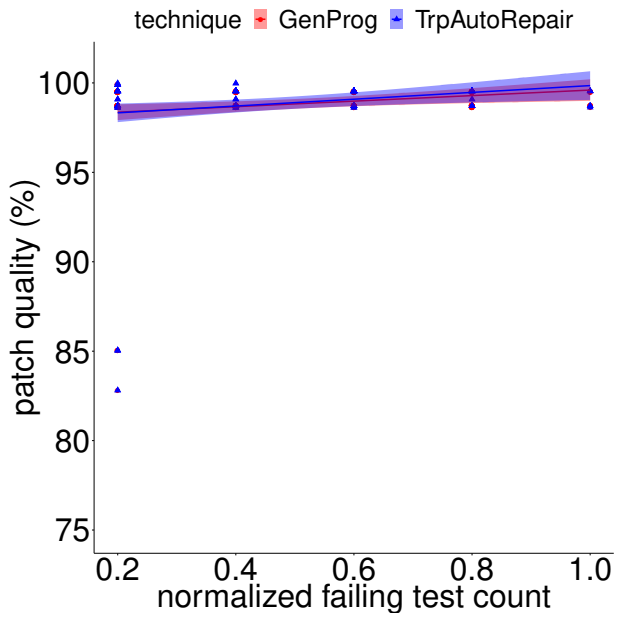


Figure 4.10: Defect severity. Linear regression between patch quality and the number of failing tests and Pearson's correlation show statistically significant positive correlations for GenProg and TrpAutoRepair [149].

described in Section 3.3 to produce patches using heuristics-based program 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 [81] for fault localization and resolves instrumentation conflicts by updating the runtime settings of EvoSuite-generated tests (following official EvoSuite documentation[1]). The EvoSuite-generated tests are compatible with JaCoCo, Cobertura [40], Clover [14], and PIT [43] code coverage tools, but not with GZoltar [32]. Unfortunately, SimFix uses GZoltar, and so could not be included in this experiment. For GenProg, Par, and TrpAutoRepair, as before, we used the developer-written patches 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 [186]), 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

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

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: Table 4.6, and Figures 4.11 and 4.12 summarize our results. Table 4.6 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.

technique	test suite	generated patches	defects patched	patch quality minimum	patch quality mean	patch quality median	patch quality 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%

Table 4.6: 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 [149].

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 4.11 shows

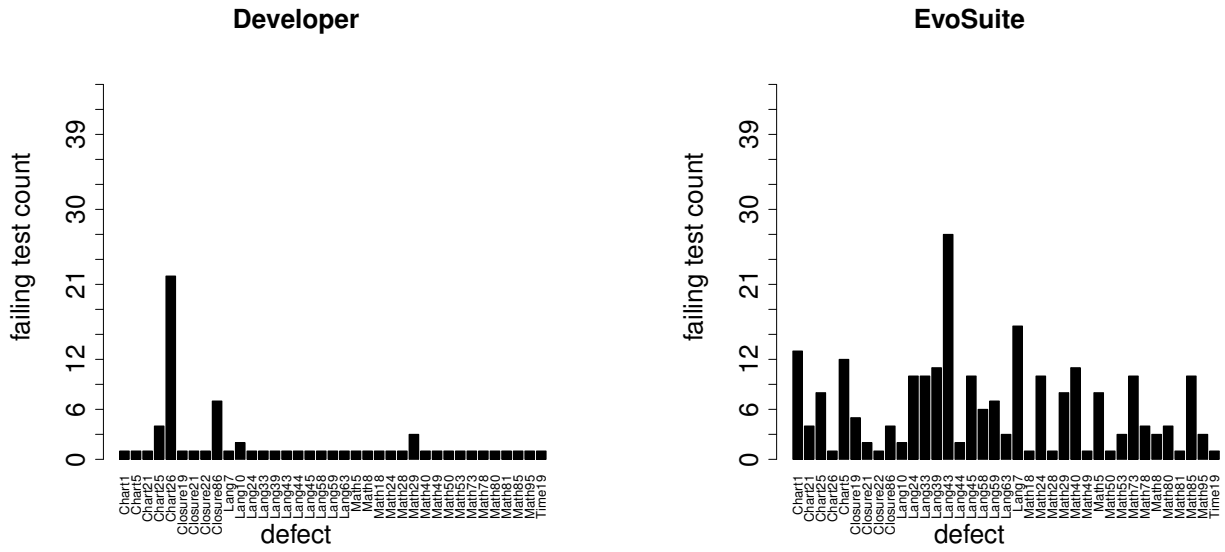


Figure 4.11: Distributions of failing tests in the 37 Defects4J defects' test suites. The EvoSuite-generated test suites typically have more failing tests than the developer-written ones [149].

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, 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 [148, 186].

We compared the quality of the patches produced using the two types of test suites on the in-common populations. Figure 4.12 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 of the delta estimate does not span 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, 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 4.12 shows that the mean and median quality of

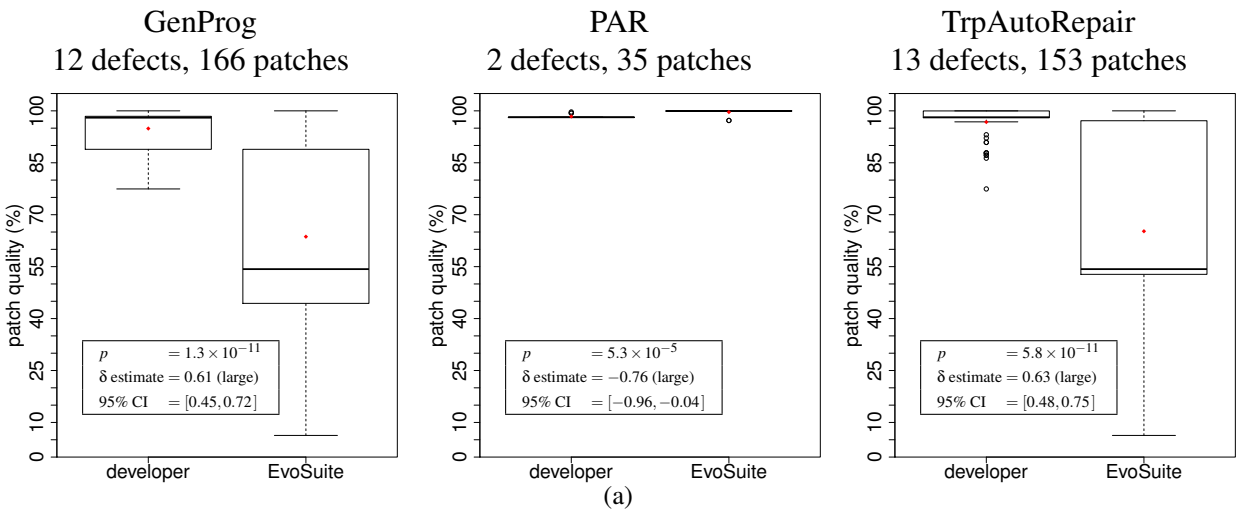


Figure 4.12: Test suite provenance. Patch quality comparison on the in-common (patched using both types of test suites) defect populations. The box-and-whisker plots compare patch quality on the in-common defect populations, showing the maximum, top quartile, median, bottom quartile, and minimum values, 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 [149].

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 [186] 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 [31]) and aimed to achieve 100% code coverage on a reference implementation, but the generated test suites were small.

Answer to Research Question 5: 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 section 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.

Chapter Summary: This chapter helps us understand how APR generated patches often overfit to the provided partial specification (guiding test suite), often breaking more functionality than they repair. We analyzed 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 then 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.

Overall, this chapter helps us further understand one of the main components of APR, the provided partial program specification, and the role it plays in the quality of generated patches. We found that there are quality attributes from these partial specifications that can be optimized to increase the quality of patches. This component is represented as phase 1 from Figure 2.1. Following, we will analyze other two components of the APR process and how their optimization increases the quality of generated patches.

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 [112], applying templates [100], transformation schemas [126, 129], or semantically-inferred code [139, 152, 216]. 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., adding a line of code that increases a variable *var* by one: *"var++;"*) can be repeated infinitely creating a different program version every time. APR approaches can thus generate a variant of source code that appends this line once, incrementing the value of *var* by one. Then generate a second variant that appends the same instruction after the previously added line, thus incrementing the

value of *var* by two, etc. The variable *var* 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 they use 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 section, 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) [129, 215].

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, 112, 129, 206].

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, 129, 215] 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 document, 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 that uses this probabilistic model to select which transformations to apply when creating patch candidates (Section 5.3).

In this section, we answer the following research questions:

RQ6 How frequently do real-world developers edit each statement kind in the bug-fixing process?

Answer: 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 (page 63). The most common mutation operator is "Append" and the least common operator is "Off by One".

RQ8 How does a human-informed automatic program repair tool compare to other APR approaches?

Answer: An APR technique using a mutation operator selection mechanism informed by developer behavior is able to generate fewer 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 (page 70), the most common consequent is "Append".

5.1 Analysis of Developer Changes in Java Projects

> **Research Question 6:** How frequently do real-world developers edit each statement kind 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 [58, 84]. 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 [57] Boa provides the infrastructure to query 4,590,679 bug-fixing commits from a database of 554,864 Java projects. These analyzed bug-fixing commits are not limited to any particular size of statement type.

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 [57]. 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.

For each statement type (as tagged by the Boa infrastructure for Java), we process the results as follows: if number of occurrences of one statement type decreased post-fix and the number of

	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 [189].

another type increased, we say that the first statement type was replaced by the second statement type for that file. Note that this analysis does not distinguish the replacement of the same statement kind, since we are counting the number of appearances of each statement kind.

We follow a similar approach to 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 `For` row (row 6) and `While` column (last column) shows 13.79, indicating that `For` statements were replaced by `While` statements 13.79% of the times. Similarly, we can infer that given a randomly selected replacement of a `For` statement there is a 13.79% chance that it will be replaced by a `While` statement. The sum of all the values in each row adds to 100%.

Given how the values in Table 5.1 were calculated, a more precise interpretation is that given a bug-fixing commit, the statement type described in the row name was removed, and the statement type in the column name was added. This deletion and addition does not necessarily need to take place in the same code location. We acknowledge that this heuristic approximation might contain inaccuracies such as the one previously described. However, the main purpose of this research question is to evaluate how human developers edit different statement kinds in different proportions. In Section 5.2, we perform an in-depth analysis of how humans modify code changes in a corpus of bug fixes, which removes the inaccuracies described in this section.

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.

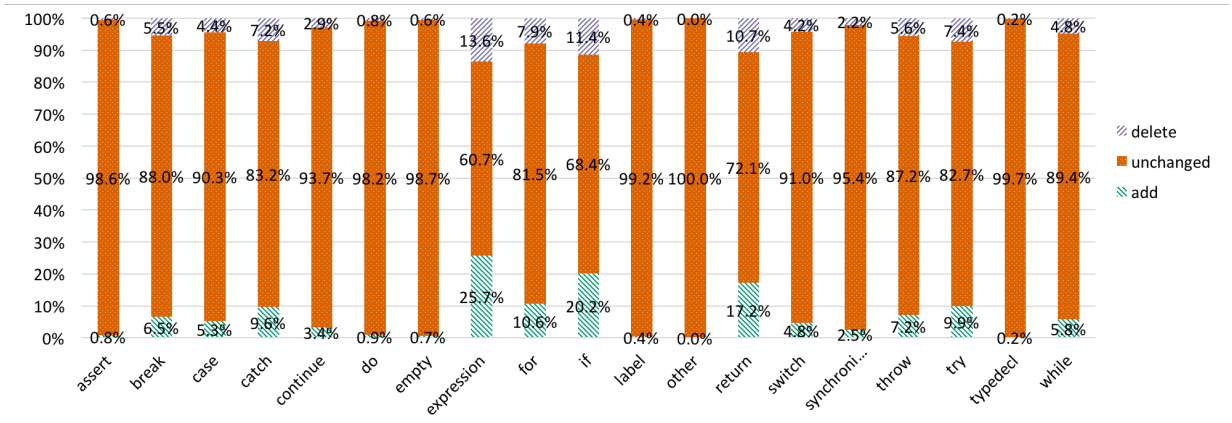


Figure 5.1: This figure shows, per each bug-fixing revision, the percentage of statements that get added, deleted and unmodified per each statement kind. Expression is the statement kind most added in the analyzed bug-fixing revisions and also the most deleted [189].

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 `TypeDecl` (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 [135], which proposes a probability distribution suggesting when to apply which 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 [135] 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: 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 [58, 84] 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 [61, 70] 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 heuristics-based program 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:

Categorizing Mutation Operators: One family of repair approaches [112, 168, 206] 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 [80].

Another family of approaches [100, 126, 129] 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.[1] 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 [188].

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

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 *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 [170] 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 [153], as used in APR [50, 215] 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.

Building Probabilistic Model: Given the categorization of the previous section 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 [24]. We followed previous approaches [44, 99, 178] to identify bug-fixing commits 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 [78, 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 [61], a source code tree-differencing framework to identify deletions and insertions. Similarly, we used components of QACrashFix [70], 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 [188].

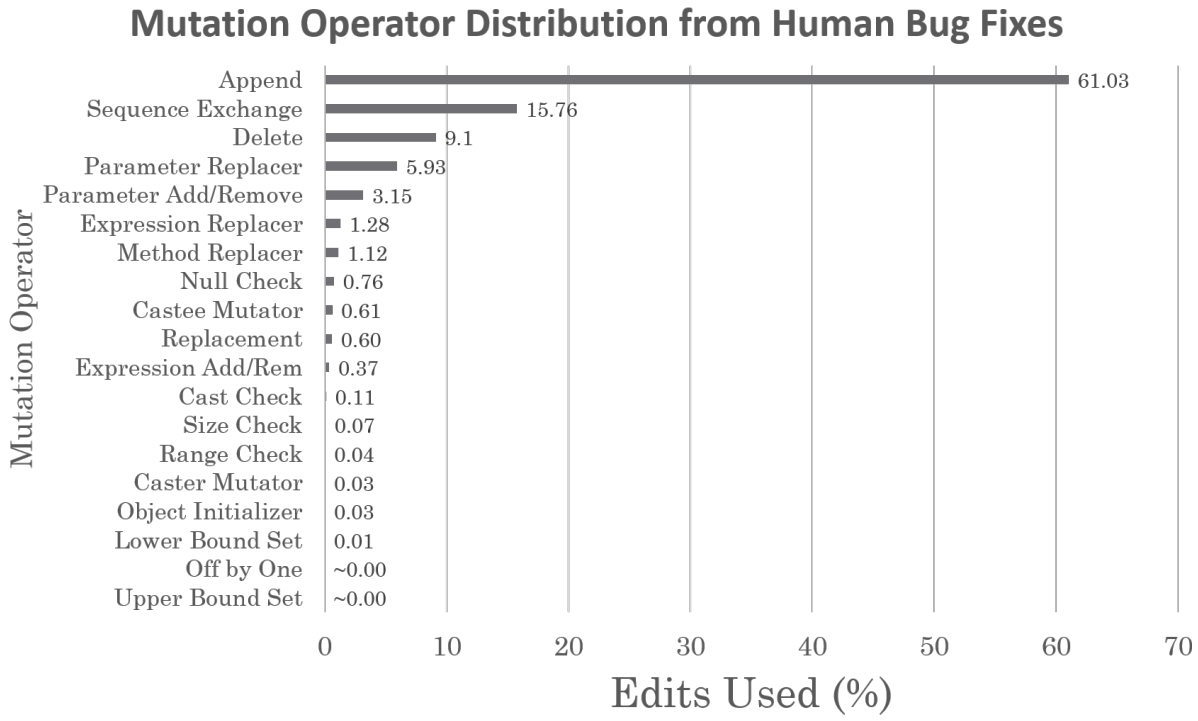


Figure 5.3: Distribution of mutation operators mined from the selected corpus of analysis. The most commonly used mutation operator by human developers is the "Append" operator, and the least used operator is "Upper Bound Set" according to our analysis [188].

Answer to Research Question 7: The distribution of mutation operators is described in Figure 5.3 (page 63). 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.

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, our 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 other APR approaches?

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 four APR approaches: GenProg [115], PAR [99], TrpAutoRepair [167] and Nopol [216]. Nopol is a semantics-guided repair approach that thus uses a fix code identification strategy that is quite different from the operators we consider.

For Nopol, we use patch results released by the Nopol authors on this same dataset, and do not rerun their experiments. The Nopol authors have created patches for the same benchmark used in this study [133] and made their results publicly available.[2] We use results from the "March 2017" release.

We implement a novel syntactic heuristics-based program repair technique that differs from prior work first, in the range of mutation operators considered; 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 and both categories of mutation operators analyzed in Section 5.2.

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. We then apply Laplace smoothing [175] 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

[2] https://github.com/Spirals-Team/defects4j-repair/tree/master/results/2017-march

Section 5.2. As described in Figure 5.3, Template-Based and Statement-Edit mutations contribute 29.26% and 70.74% of the studied edits, respectively. Statement-Edit mutations refers to `Append`, `Delete` and `Replacement`, Template-Based mutations is composed of all the remaining mutation types.

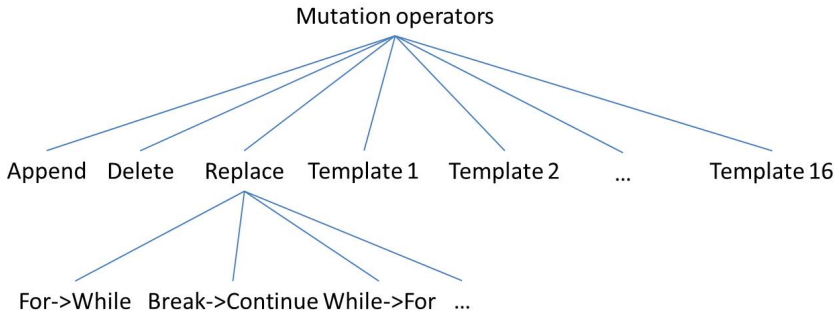


Figure 5.4: This figure describes the two level probabilistic model used in our study. The first level selects among three coarse-grain mutation operators and 16 fine-grain templates. If the "Replace" mutation operator is selected, then a second level of the model is used to select which statements replace 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 [188].

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

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

Table 5.2: Comparison of patches generated using the probabilistic model-based repair and other APR approaches. "–" indicates no patch found. The "Found" column indicate the number of patches found per bug over the multiple random trials. "Gen?" 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.

16. SwitchStatement
17. SynchronizedStatement
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 APR 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%); 6.6% of GenProg's patches generalize; 22.2% of TrpAutoRepair's; 23.1% of PAR's; and 100% of Nopol's five 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.

```
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 [188].

APR Technique	Bugs Patched	Patches Generated	Patches Generalize
Probabilistic Model	9	19	10 (52.6%)
GenProg	9	46	5 (10.9%)
TrpAutoRepair	16	30	4 (13.3%)
Nopol	27	27	21 (77.8%)
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.

Regarding how many bugs each APR approach was able to patch, our technique patched 9 of the 63 bugs in our evaluation (shown in Table 5.2). From these 63 bugs, GenProg was able to patch 9; PAR, 16; Nopol, 27; and TrpAutoRepair, 8. At least one approach produced at least one patch for 37 bugs. From these, 19 were patched by only one approach; 10 were patched by two; 3 patched by three; 5 patched by four; and 1 patched by all five. 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%) generalize to the held-out test suites. TrpAutoRepair created 30 patches; 4 of them (13.3%) generalize to the test suites. PAR created 34 patches; 11 of them (32.4%) pass the held-out test suites. Finally, Nopol created 27 patches; 21 of them (77.8%) generalize to the held-out 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 (%)").

Even though Nopol produced a higher percentage of patches that generalize to the held out test suite, it is worth noticing that Nopol targets a specific bug type (bugs in `if statements`)

and does not use a heuristic approach. Our approach therefore presents an important benefit for a broader array of bug types, particularly those to which a semantics-based approach like Nopol does not apply. Overall, these results demonstrate the benefit of applying a probabilistic model for edit selection for those approaches for which such a selection process applies.

Answer to Research Question 8: An APR technique using a mutation operator selection mechanism informed by developer behavior is able to generate fewer 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 [189, 222]. 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 [177]) 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 [6].

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) 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 (sets of 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 (non-overlapping subgroups). For each fold, we build association rules on the remaining nine folds, 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*. *Fully covered* refers to the patches where all of the mutation operators included in that patch can be instantiated by applying the evaluated association rules. Similarly, *Partially covered* refers to the patches where only a subset of mutation operators can be instantiated, and *Not covered* when none of the mutation operators can be instantiated using the association rules under analysis.

*Table 5.4: Association rule example: The top portion (*Patches*) describes three different patches. The bottom section (*Rules*) shows three association rules generated from the corpus of patches described above. [188].*

	Patches
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

To illustrate via example, Table 5.4 shows at the top three different patches, one per row. Each row describes the mutation operators (separated by semicolons) used to create that patch; the order in which the mutation operators were applied is irrelevant for this analysis. The bottom section

shows three association rules. The goal of these coverage metrics is to evaluate what portion of each patch could be created from the association rules.

In this analysis, an APR approach that uses these association rules could potentially build the entire patch, a portion of the patch, or none of the patch (represented in this case by a set of mutation operators). A rule is then considered to be applied if all the mutation operators described in the antecedent and the consequent of said rule are present in a given patch. Therefore, Patch 1 could be constructed by applying rules 1 and 3. The superset of mutation operators in the antecedent and consequent of applied rules 1 and 3 covers all the mutation operators described in Patch 1. We thus classify Patch 1 to be Fully covered by the set of rules.

For Patch 2, Rule 1 can be applied and it would cover the first three mutation operators of Patch 2, but the Replace ("Rep") mutation operator cannot be generated using the listed rules. Replace does show as the consequent of Rule 2, but to obtain that consequent all the antecedents must be present in that patch as well. In this case, ParamRep is not present in Patch 2. This instance is thus classified as Partially covered. For Patch 3, even though Rule 3 contains two of the edits in the rule's antecedent, the instance does not contain the rule's consequent, thus this rule cannot be applied and this instance is classified as Not covered.

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 dissertation):

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.

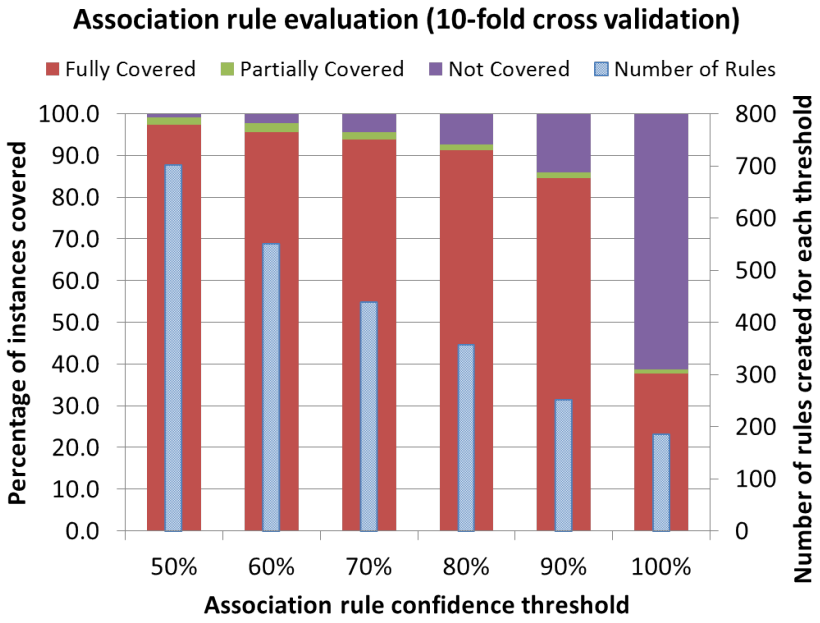


Figure 5.6: The wide bars use the left vertical axis to describe the percentage of patches covered (Fully, Partially or Not) by the association rules. The thin bars use the right vertical axis to describe the number of rules created for each confidence threshold [188].

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, because 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 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 (page 70), the most common consequent is "Append".

Chapter Summary: This section 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 multi-edit patches.

Overall, these experiments show the importance of improving the way APR approaches generate patch candidates and how this component of the APR process can be improved by mimicking the way humans patch code. This component is described as phase 3 in Figure 2.1. The next section will describe how we can improve the final stage of APR (described as phase 5 in Figure 2.1) were we can incentivize patch diversity to improve the quality of generated patches in APR.

Chapter 6

Patch Diversity and Consolidation in Automatic Program Repair

Heuristics-based program repair approaches have the ability to generate several patches for a single bug. Realistically, a single patch is sufficient to properly fix a defect. Therefore, in this chapter, we propose two approaches focused on creating/finding the best patch from a population of patches. The first technique focuses on incentivizing *diversity* in the patch generation process with the goal of creating more diverse patches. Given a more diverse pool of patches, the quality of the patches comprised in that population also tends to show higher variance. Higher quality variance may increase the quality of the highest-quality patch of the population. The second technique describes two types of *patch consolidation* where several lower-quality patches can be combined into a single higher-quality patch. As described in earlier chapters, throughout this document we separate the tasks of building higher quality patches and evaluating the quality of said patches. To identify the highest-quality patch from a population we use held-out test suites (Section 3.3). In production settings, where held-out test suites are not available, this identification can be performed, for example, by human developers.

This section provides an in-depth study of how patch diversity can be used to increase the quality of the best-quality patch found in a population; and how individual patches can be consolidated with the goal of increasing patch quality. We start by evaluating a set of diversity-driven techniques to increase diversity in patch populations. Later we evaluate how patches consolidated through n-version voting compare to the individual patches they are generated from. Finally, we create code-merging consolidated patches and analyze their quality in terms of each APR's ability to generate diverse patches. Therefore, in this section we will answer the following research questions:

RQ10 How do diversity-driven techniques affect the quality of the best patch found for a given bug, and the diversity in the patch population?

Answer: In the majority of cases, the proposed techniques increased (24 out of 57) or maintained (27 out of 57) the diversity in the populations of plausible patches found per bug. Similarly, the quality of the best-performing patch in the diversity-driven population is higher (7 out of 57) or equal (47 out of 57) than its homologous in the non-diversity-driven population.

RQ11 How does the quality of patches consolidated through n-version voting compare to the quality of single patches?

Answer: The quality of n-version voting consolidated patches is negligibly higher than individual patches. Individual plausible patches are very similar; therefore, their corresponding consolidated patches are also very similar.

RQ12 How do code-merging consolidated plausible patches behave in relation to their non-consolidated counterparts in terms of patch quality?

Answer: The impact code-merging consolidation has on patch quality varies depending on the patches being consolidated. In our study, code-merging consolidation was especially beneficial for the patches generated by GenProg, the approach with the most tools to generate diverse patches.

6.1 Improve Diversity of Generated Patches

As described by previous studies [149, 187], a common characteristic of heuristic-based APR approaches is the generation of redundant or semantically identical patches. Patches as such can, in principle, be identified when two programs present an equal output given any possible input (programs have the same logical content [142]).

Our intuition is that in cases where an APR approach generates a population of plausible patches that are low-quality but diverse, we can leverage this patch diversity to increase the quality of the best patch in the population. Thus, we analyze possible ways to increase diversity and use that diversity to increase the quality of the best patch in the population. Given that in heuristic-based APR, plausible patches, by definition, behave equally when evaluated in the guiding test suite; our study focuses on how patches behave in the held-out test suite, which are test cases different and independently generated from the guiding test suite.

Figure 6.1 shows an example of three redundant plausible patches found for the bug Closure 13. All of them behave equally both in the guiding and held-out test suites.

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. Examples as such, are commonly found by APR approaches and present little or no semantic diversity. In this section, we evaluate ways to increase and exploit patch diversity in the APR process with the end goal of benefiting overall patch quality.

We hypothesize that an APR approach that incentivizes patch diversity can be beneficial to automated repair. We evaluate how we might increase patch quality of the best patch in the population by taking advantage of an increase in patch diversity. Our intuition is that by increasing patch diversity the APR approaches will also increase the variance of quality of the generated patches with the intent of increasing the quality of the highest-quality patch in the population.

The top subgraph of Figure 6.2 shows a representation of how the quality of patches typically behaves for a population of patches generated given a single bug. In the subgraph at the bottom

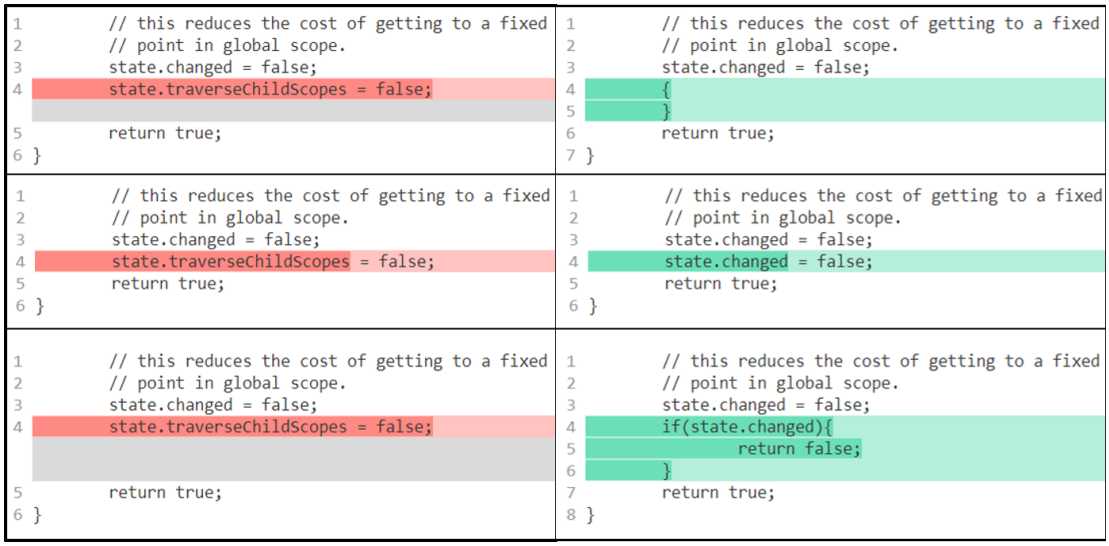


Figure 6.1: Three semantically identical patches showing the lack of diversity in plausible patches generated by automatic program repair approaches.

of Figure 6.2 we show how diversity-driven patches might increase the variance of patch quality. We evaluate how using a diversity-driven approach in APR might then increase the quality of the highest-quality patch in the population.

As shown in this figure, throughout the chapter we describe patch quality as a spectrum of values instead of a binary (correct/incorrect) value. This allows us to describe incremental increases in patch quality, which are valuable, even when the resulting patch is not a perfect patch that behaves correctly in all possible cases. In practical terms, fixing an important portion of a bug can have an enormous impact for a business if the fixed portion is used often. Fixing an important portion of a bug has intrinsic value, even if the patch does not fix the bug for all possible cases.

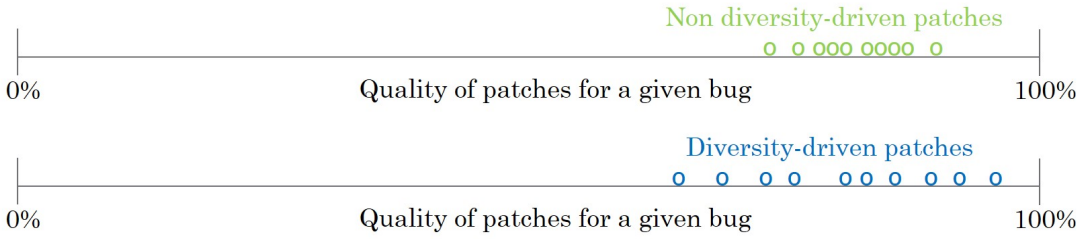


Figure 6.2: The subfigure in the top shows a typical description of patch quality for a population of patches for a given bug. The subfigure in the bottom shows how diversity-driven approaches might increase the variance in quality in the population.

We thus start by performing a set of techniques to increase patch diversity and analyze the variance in quality of the patch pool. A more diverse population of patches amplifies the potential of increasing quality of the best patch of the population. To achieve an increase in quality, we propose, implement, and analyze a set of techniques to enhance diversity in patch candidate search space traversal so that the encountered plausible patches are more likely to be semantically distinct. These techniques are inspired by distributed search algorithms with the goal of finding

local optima by segregating the search space [82]. To this end, we propose the following research question:

> **Research Question 10:** How do diversity-driven techniques affect the quality of the best patch found for a given bug, and the diversity in the patch population?

Methodology: In this section, our goal is to evaluate if we can create techniques to increase the diversity in a patch population, and if this increased diversity leads to an increase in the best-performing patch in the population. To evaluate whether we can create more diverse patches, we perform a controlled experiment on a set of bugs, where we generate patches using four proposed diversity-driven techniques implemented within the repair algorithms. The four diversity-driven proposed techniques are:

- Slicing test cases
- Slicing fault locations
- Slicing mutation operators
- Multi-objective search

As measurement for quality we use held-out test suites (as described in Section 3.3); as measurement for diversity we use behavior bucketing (Figure 6.3).

We start by slicing the problem space in three different ways (guiding test cases, fault locations and mutation operators). In this document, "slicing" refers to the process of creating subgroups of each of the specified components and executing each APR approach with the slice (i.e., subgroup) selected. Our intuition is that by slicing these APR components and executing the APR approaches only within the slice selected, the APR approach will be forced to use the given slice and create patches within that slice exclusively, thus creating different patches per each slice. We also propose a diversity-driven technique for genetic-algorithm based APR approaches where we modify the objective function to guide the search space traversal.

The *test cases* in the guiding test suite (partial program specification) describe the expected behavior of the program. By slicing these test cases, we create subsets of the partial program specification and thus modify the description of how the program should behave. This change allows for more freedom in the APR approach to find different patches and modifies the way that the APR approach navigates the search space. By slicing *fault locations*, the APR approach can consider different locations in the code where the bug might be located. This varies the area of the program where the APR approach looks for a patch. Thus, if patches are found in different locations, these patches have a higher probability of being more diverse.

Mutation operators describe the tools used by the APR approach to edit source code. Slicing mutation operators, thus, forces the APR to use a subset of mutation operators per execution and going in depth into what patches can be found with the selected mutation operators. Given that different mutation operators modify the source code in different ways, creating patches using different mutation operators increases the chances of creating patches that are more diverse.

Finally, we generate a mechanism to modify the fitness function of genetic-algorithm based APR approaches (e.g., GenProg [115], kGenProg [79], jGenProg [136], PAR [99], HDRepair [110]). 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.

We therefore designed and implemented full executions extending JaRFly to restrict the mutation operators, test cases and fault space used (described in detail below). Similarly as in previous experiments, per each bug we run an APR approach using each slice. The APR approaches are run using 20 different seeds and four-hour time slots per seed. Given the computational intensity of these experiments, we executed our diversity-driven approaches in a subset of the Defects4J bugs, thus considering the bugs where human developers used single line changes to patch the bug (63 defects).

We generate plausible patches for our corpus of bugs both with and without the diversity-driven techniques, and we compare these two populations. We evaluate the diversity in the populations of plausible patches generated, and the quality of the best performing patch of each population.

The goal of this technique is to evaluate how diverse a population of plausible patches is by dividing the plausible patches based on their behavior and analyzing the number of plausible patches that behave differently. The higher the number of plausible patches that behave differently within a population, the more diverse the population is. This technique follows the same principle of evaluating the analyzed programs based on how they behave (semantics) independent of how they are written (syntax). We use this technique both for the diversity-driven approach described above and for off-the-shelf standard techniques we compare against.

Given the population of patches, we then group all the patches that fail the same tests into a single bucket as described in Figure 6.3. All patches within a single bucket are guaranteed to behave differently from the patches in other buckets. Therefore, a population of patches that can be grouped into a higher number of buckets implies a higher diversity of behavior within the patches of that population, as opposed to a population of patches that can be grouped into a smaller number of buckets.

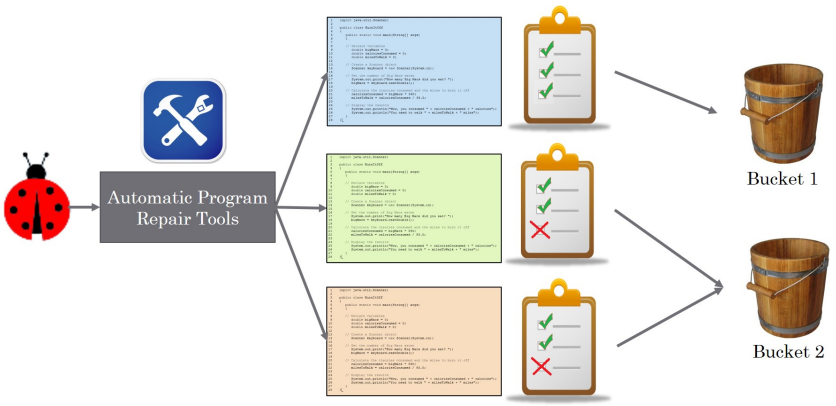


Figure 6.3: Each bug is run through an automated repair technique, which may generated several patches. We group the patches that fail the same tests into buckets. The patches in one bucket are guaranteed to behave differently than the patches in other buckets. The higher number of buckets a population is divided into, the higher the behavioral (semantic) diversity of that population.

6.1.1 Slicing Mutation Operators, Fault Locations, or Test Cases

We implemented a 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) [82]) where we partition the problem space to find local optima and given the segregation, the possibilities of finding diverse solutions increase. Previous studies [35] report that MIGA significantly increases diversity of the population and has a better performance in finding the global optimal solution. In our study, we are mostly leveraging the former benefit from using 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 restrict the APR approach to use a single mutation operator per execution therefore segregating the search space in non-overlapping sets of mutation operators. Slicing single mutation operators allows for maximum usage of each mutation operator in each execution, it forces the APR approach to find plausible patches that are created exclusively using the selected mutation operator.

We acknowledge the increase in number of executions when slicing APR components as proposed in this section, which makes these approaches more computationally intensive when compared to the standard APR approaches. However, since the main goal of this research is to find higher-quality patches, we consider this trade-off worthwhile, especially since the cost of computational resources is rapidly declining, and given that these executions can be performed in parallel. Thus, the 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	· Off by One	· Upper Bound Set	

Slicing Fault Locations

When slicing fault locations, our intent is to segregate the places in the program where the APR will be looking for patches with the intuition that patches found in distinct places have a higher probability of being semantically diverse. Similarly, we restrict the search space of program locations to a single subset therefore forcing the automated repair approach to search for candidate patches considering the specified program slice exclusively.

The procedure we followed to create the fault location subsets is described in Algorithm 3. Our fault localization technique works at program statement level, thus we traverse the analyzed program recursively assigning node numbers to each program statement. We segregate the program statements that describe the location search space (*codeBank* in Algorithm 3) into five equally-sized subsets. Different from slicing mutation operators, we cannot segregate the fault locations into single program statements given our location search space tends to contain

thousands or millions of statements. Thus, we create five subsets of program statements per bug, which provides a balance between number of program statement and a feasible number of executions. Finally, we execute each bug and each APR technique using the selected portion of code exclusively.

Algorithm 3 Segregates the location search space into s number of 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)
```

Slicing Test Cases

When slicing test cases our intent is to modify the provided partial program specification thus increasing the probability of creating more diverse patches. In this experiment, APR approaches consider exclusively a subset of the passing test cases in the guiding, developer-written, test suite, and all of the tests that evidence the defect (failing 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.

To generate the test suite subsets for each defects we used the methodology described in Section 4.2.2 where we compute the minimum and the maximum code coverage of the developer-written test suite of that defect. We then choose five coverage *targets* evenly spaced between the minimum and the maximum coverages. Finally, we used these coverage targets to create five distinct test suites, one per each coverage target, and we execute our APR approaches using each test suite.

6.1.2 Implementing Multi-Objective Search to Incentivize Patch Diversity

Several current APR approaches use genetic programming to find plausible patch candidates (e.g., GenProg [115], kGenProg [79], jGenProg [136], PAR [99], HDRepair [110]). Genetic programming relies on a fitness function used to compute the likelihood of patch candidates to become plausible patches. Most approaches set their fitness function to look mainly for patch correctness (e.g., [115, 188, 206]) 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 of the genetic algorithm.

One way to incentivize diversity when traversing the search space is by modifying the fitness function to optimize for several parameters of interest making it a multi-objective fitness function, instead of the current single-objective implementation. We created an extensible multi-objective fitness function, which can optimize for several parameters at the same time. This fitness function takes the form of:

$$patchCandidateFitness = \sum_{n=1}^{k} weight_n * score_n$$

where k represents the parameters that compose the multi-objective fitness function, and *patchCandidateFitness* symbolizes the fitness of each patch candidate. It is described as a number between zero (least fit) and one (maximum fit) inclusively. The variables $weight_n$ describe how much each parameter is contributing to the overall fitness score. The variables $score_n$ are defined by each parameter. Both $weight_n$ and $score_n$ are values between 0.0 and 1.0 inclusive.

In this experiment, we instantiate this multi-objective fitness function to account for two parameters: a correctness score, and a semantic diversity score (a quantitative measurement of diversity), so that these two dimensions are encoded into the search criteria. Given that our fitness score calculation is based on a linear combination of fitness scores per each parameter, Pareto frontiers could be computed to maximize the distribution of the weights associated to diversity and fitness scores (i.e., the best percentage to assign to each of these parameters). However, such an analysis is computationally expensive; thus, we set an equally distributed weight assignment of 0.5 for fitness and 0.5 for diversity.

Semantic program equivalence is undecidable [173], making the computation of a score for diversity (and equivalence) an undecidable problem. Therefore, we have proposed and implemented a semantic measurement to approximate software equivalence and its opposite, software diversity, as described in Section 6.1.2.

The fitness computation for this two-dimension multi-objective objective function thus take the form described below:

$$patchCandidateFitness = correctnessWeight * correctnessScore + diversityWeight * diversityScore$$

where a fitness of 0.0 would imply that the patch candidate does not pass any of the test cases and it is semantically equivalent to all other candidate patches. The variable *correctnessScore* is computed as the number of passed test cases divided by the number of all test cases (some approaches use a sample of the test suite to calculate fitness before running the entire test suite to validate a plausible patch) and *diversityScore* is computed as the average of the semantic diversity scores between one patch candidate and all other patch candidates in its population. Diversity score computations are described in Section 6.1.2. Similarly as with the previous approaches, an APR approach is run per each bug using 20 different seeds and four-hour time slots per seed.

Test-Suite-Based Semantic Difference Measurement:

As described in the previous section, optimizing for patch diversity requires a quantitative measurement of "how different" a patch candidate is with respect to others. A quantitative measurement allows us to reason about how different a patch candidate is from another. With this goal, we proposed and implemented an approach to approximate semantic patch diversity, which tells us how different two programs behave independently of their syntax.

Figure 6.4 diagrams the implemented process to approximate the semantic difference between two candidate patches, Candidate Patch A and Candidate Patch B. The first step (i.e., ① in Figure 6.4) is to create a partial 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 [64], Randoop [158]).

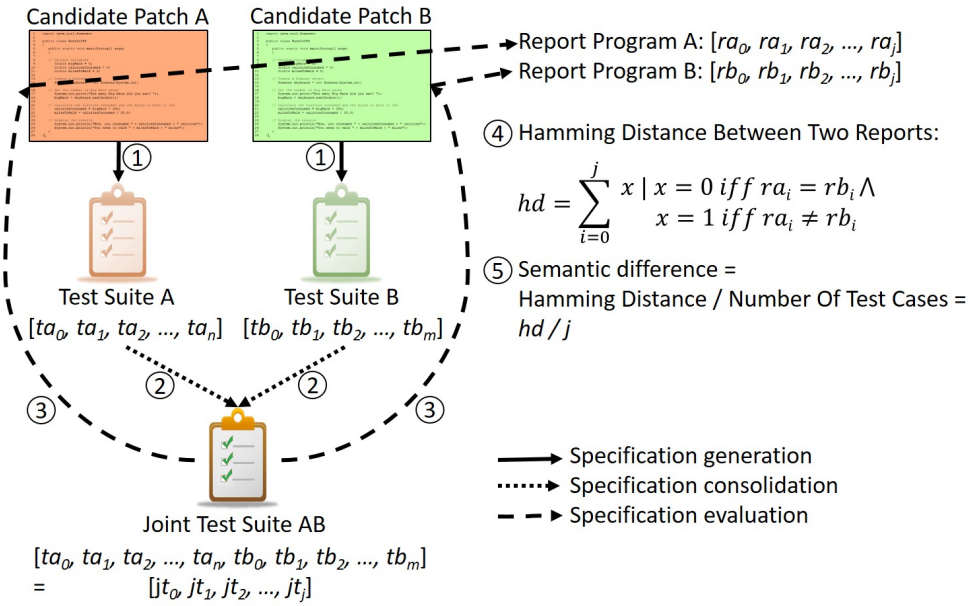


Figure 6.4: Diagram describing the implemented semantic difference measurement. Given two programs, we create a partial specification from each program (1), then unify both specifications into a single one (2). The unified specification is then evaluated in each of the programs (3). Finally, we calculate 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).

We then (i.e., ② in Figure 6.4) combine both partial specifications into a single test suite AB. This consolidated test suite is used to evaluate each individual program (i.e., ③ in Figure 6.4). We compare the reports from the evaluations using Hamming distance where *ra* is true is test jt passes and false otherwise, same for *rb* (i.e., ④ in Figure 6.4) to identify which cases behave differently. Each report consists of a list of passing/failing tests. We thus compare, per each test, when both candidate patches behave the same (both pass the same test, or fail the same test), and where the candidate patches behave differently (one fails and the other passes a test). Hamming distance is fast and assumes equal-length strings, making it appropriate for our analysis. Finally, we compute the semantic difference based in the Hamming distance as a percentage of the number

of test cases (i.e., ⑤ in Figure 6.4). As a result we obtain a metric describing how semantically different is program A from program B.

This methodology is inspired in literature of semantic code clones [88, 182, 195, 202, 218] 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 the opposite, 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 [195] using a variety of techniques. Such methodologies include information retrieval [131], program dependence graphs [103], static analysis to extract memory states in function exit points [141], method calls at bytecode level [94] tree-based convolution [218], concolic analysis [105], and randomized testing [51, 88] In our approach, we deal with programs that are almost entirely identical except for portions that are potentially semantically different (i.e., the patched portion of the program), therefore previous approaches from code clones literature are not practical to use in this scenario. We thus proposed and implemented the functionality described in Figure 6.4 as an alternative approach to find code diversity inspired in previous code clone literature.

Results: In this section we analyze the results from the experiments executed based on our diversity-driven techniques to find higher quality patches and greater diversity in the APR process. In these experiments, we evaluate APR approaches with and without the usage of our proposed diversity-driven techniques as described in Section 6.1, and analyze the diversity of each population of patches per each bug and the quality of the best-performing patch within the patch population. We aim to analyze if the proposed techniques increases diversity in the generated patch populations, and if the best patch found using the diversity-driven techniques shows an increase, decrease or maintains the quality as the best patch found without using the proposed techniques.

Population Diversity: Table 6.1 shows the diversity results when slicing test cases. To directly compare the diversity-driven approach to the off-the-shelf standard APR approach, we analyze the in-common populations of patches found, where both the diversity-driven techniques as well as the non-diversity-driven techniques were able to find patches. The pairs of cells highlighted in green shows the cases where the population of patches generated using the diversity-driven approach was grouped into a higher number of buckets than the population of patches generated without using the diversity-driven technique, thus showing higher diversity in the patch population.

From the 25 in-common patch population pairs found, 15 show an increase in patch diversity when using test case slicing as opposed to not using test case slicing. For this diversity-driven technique, in 10 of 25 in-common patch population pairs found, the number of buckets was the equal for both with and without test case slicing, implying that the diversity remained the same for these cases. Finally, there were no cases where the diversity decreased when using test case slicing as opposed to not using test case slicing. The greatest increase in diversity was found for the bug Math 2 using test case slicing in PAR, where this diversity-driven technique generated a 12-fold increase in diversity as opposed to the population generated without using this diversity-driven technique.

The results described in the previous paragraph and the results for the remaining diversity-driven approaches are summarized in Figure 6.5. Each bar represents a diversity-driven technique.

Table 6.1: Diversity Results Slicing Test Cases: The table shows three pairs of columns, one pair of columns per each APR technique evaluated. Each pair of columns describes on the left, the number of buckets created for the population of patches without using test case slicing; on the right, the number of buckets created using the diversity-driven technique (test case slicing). The pairs highlighted in green shows the cases where the diversity-driven technique created a higher number of buckets (higher diversity) than without the diversity-driven technique.

Bugs Patched	Buckets Without Slicing GenProg	Buckets Slicing Tests GenProg	Buckets Without Slicing PAR	Buckets Slicing Tests PAR	Buckets Without Slicing TrpAutoRepair	Buckets Slicing Tests TrpAutoRepair
Chart 1	2	5	2	2	1	4
Closure 31			1	1		
Closure 62			1	1		
Closure 63			1	1		
Closure 86					1	1
Lang 33			1	1		
Lang 51	1	1	1	2		
Lang 58			2	4		
Lang 59	1	5	1	1	1	3
Math 2			1	12		
Math 5			1	3		
Math 75			1	1		
Math 80	4	10	1	6	2	8
Math 85	2	3	1	6	1	2
Time 19	1	2			1	1

The height of the bar describes the number of in-common patch population pairs found. The top section of each bar shows the number of population pairs where the diversity-driven approach was able to generate a higher number of diversity buckets than without using the diversity-driven technique. The middle portion of the bar shows the cases where the diversity was maintained equal with and without the diversity-driven technique. Finally, the bottom section of each bar shows the cases where using the diversity-driven technique led to a decrease in patch diversity in the population of patches. Overall, the proposed approaches increased diversity in the studied populations of plausible patches, with the exception of the diversity-driven technique based on multi-objective search. At the end of this section, we describe possible reasons for these results and propose ways to improve this approach in the future.

Best-Performing Patch Quality: Figure 6.6 describes the quality comparison between the diversity-driven patch populations of patches against the non-diversity-driven populations where we consider the best patch of each population. In this figure we describe in the top portion of each bar (green) the in-common patch population pairs where the patch with the highest quality in the diversity-driven population surpasses in quality the patch with the highest quality in the non-diversity-driven population. Figure 6.6 shows that in the majority of cases the APR approaches using diversity-driven techniques increase in quality when compared to the non-diversity-driven approaches.

The two remaining portions of each bar represent the in-common patch population pairs where the quality of the best patch was maintained (middle portion of the bar, grey), and finally, the cases where the best patch decreased in quality when using the diversity-driven technique (bottom portion of each bar, orange).

Overall, the cases where quality of the best patch of the population is increased surpass the cases where quality of the best patch decreased in all but two diversity-driven techniques. It is also

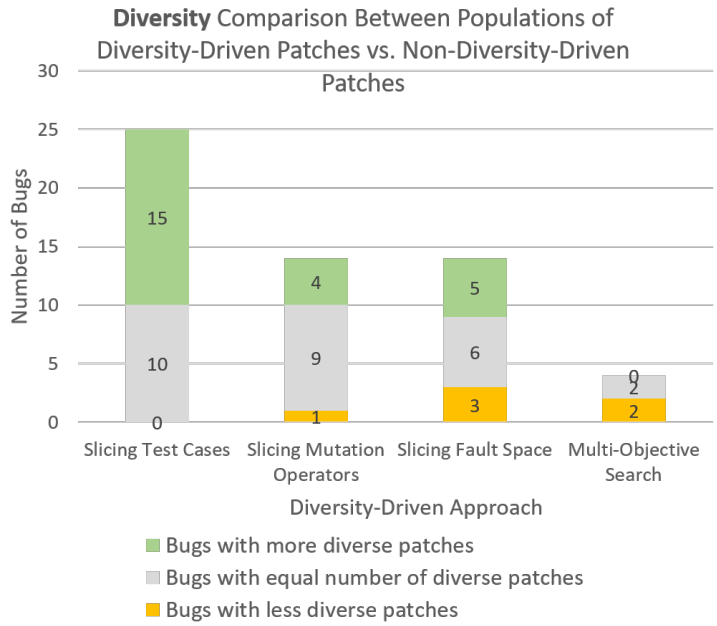


Figure 6.5: Diversity-Driven Results (Diversity): the top section of each bar (green) shows the number of population pairs where the diversity-driven approach was able to generate a higher number of diversity buckets than without using the diversity-driven technique. The middle portion (gray) of the bar shows the cases where the diversity was maintained with and without the diversity-driven technique. The bottom section (orange) shows the cases where diversity decreased using the diversity-driven technique.

important to notice that the technique where diversity decreased the most (Slicing Fault Space) also shows the largest decrease in quality of the best patch in the population; and vice versa, the case where diversity increased the most (Slicing Test Cases) also shows the largest increase in quality of the best patch of the population. This supports our intuition that increasing diversity leads to an increase in quality variance, which may lead to an increase in quality of the best patch in that population.

Answer to Research Question 10: In the majority of cases, the proposed techniques increased (24 out of 57) or maintained (27 out of 57) the diversity in the populations of plausible patches found per bug. Similarly, the quality of the best-performing patch in the diversity-driven population is higher (7 out of 57) or equal (47 out of 57) than its homologous in the non-diversity-driven population.

Software diversity can lead to an increase in quality for the best patch in the patch population of patches generated in APR. Our experiments show that slicing test cases is particularly beneficial for both generating a more diverse pool of patches, and creating a better highest-quality patch when compared to the patches found without using our diversity-driven techniques. It is also important to understand that increased variance in quality (as a product of increased diversity) also implies a risk of lower-quality patches. Increasing quality variance expands the boundaries of quality in both directions, it can generate better higher-quality patches and worse lower-quality

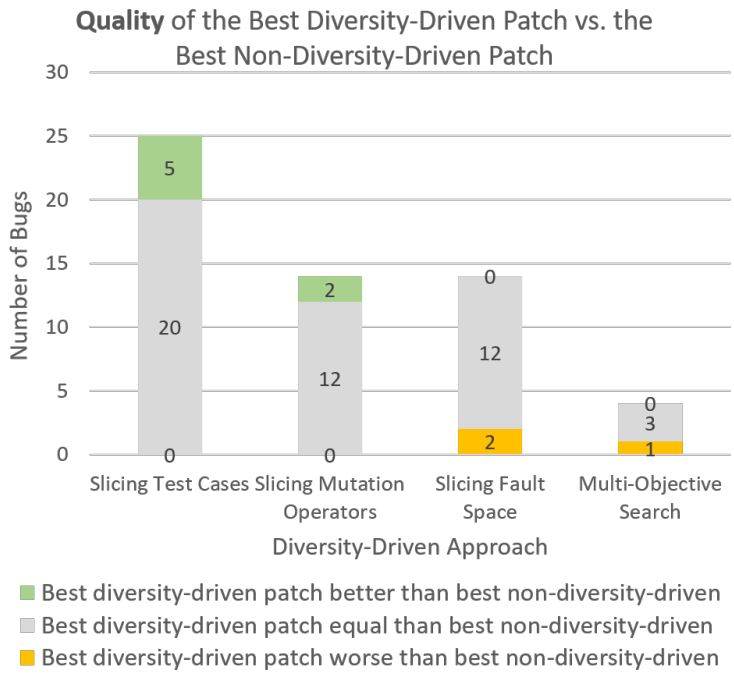


Figure 6.6: Diversity-Driven Results (Quality): the top portion of each bar (green) the in-common patch population pairs where the best patch in the diversity-driven population surpasses in quality the best patch in the non-diversity-driven population. The middle portion (gray) represents the pairs where the quality was maintained, and the bottom portion (orange) represent the cases where the quality decreased.

patches. In our analysis, we focus on the highest-quality patch of the population given that a single patch is sufficient to fix a defect.

Slicing mutation operators and slicing fault space led to an increase in diversity in the patch population. Slicing mutation operators also led to an increase in quality. The multi-objective search approach was the technique that underperformed the most by finding a smaller number of unique patches and overall a lower quality for the patches found. We acknowledge that this technique is particularly time consuming given the process of creating and evaluating test suites mid APR execution. Improvements can be made to attenuate this limiting factor, like reducing the population size of patch candidates. This is a configurable parameter in genetic-algorithm-based APR that describes how many patch candidates are created per generation. Reducing the population size would reduce the time it takes to create the test suites given that the approach generates one test suite per patch candidate, and reduces the time it takes to evaluate the test suites given that the behavior of each patch candidate is compared against the behavior of all other patch candidates (i.e., this comparisons grow exponentially, not linearly).

Another possible improvement is to use proxies for semantic diversity such as the number of test cases failed per each patch candidate. Finally, another improvement could be to modify the weight of the different parameters. In this experiment we performed our executions using a 50-50 weights (i.e., 50% of the fitness score comes from the correctness score and 50% comes from the diversity score). A high diversity weight like this one, can guide the APR approach towards a

portion of the search space that is unlikely to contain correct patches. Therefore, reducing the diversity weight of the objective function can be beneficial to finding more patches.

6.2 Patch Consolidation

A possible use of low-quality patches is to create a single higher-quality consolidated patch based on the combination of several lower quality ones. The intuition behind this idea is that there can be cases where each plausible patch might be a partial solution that covers a subset of the correct execution space but fails to cover all possible correct executions of the expected behavior. Therefore, consolidating several plausible patches might increase the number of correct executions and thus the quality of the overall solution.

In this section, we perform an analysis to understand when patch consolidation could lead to an increase in automated patch quality. We propose and analyze two consolidation techniques: consolidation through *n-version voting*, where individual patches vote on whether a consolidated patch passes or fails each test of a test suite; and consolidation through *code-merging*, where we use an off-the-shelf code-merging tool to generate consolidated patches and later evaluate their corresponding quality.

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) * 5/9.0 + 273.15);
19       ret = Integer.toString(degrees);
20       ret += " K";
21    }
22    return ret;
23 }
```

Figure 6.7: Illustrative example of a program that displays temperature in different systems. The program takes two variables as inputs: the degrees in Fahrenheit, and the system to display. It returns the temperature to be displayed.

Set of test cases that describe intended program behavior (*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 6.8: Corresponding test suite of the program shown in Figure 6.7 which partially describes the intended behavior of the program.

Consolidation Example: Consider Figure 6.7, which depicts a program that displays temperature in different measurement systems. We use this simplistic example for explanatory purposes since real world systems tend to be more complex. This program 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 conversion 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 when converting to Celsius ('c') since the variable `degrees` is converted to the correct number (line 14), but it is never added to the return String `ret` (line 15).

In Figure 6.8, there is a partial specification (test suite) describing the expected behavior of the program in Figure 6.7 using a set of cases. 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.

Figure 6.9 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 6.8 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`. This is a contrived and simplistic fictional example meant to easily explain the basic notion behind patch consolidation. The patches generated for our study of real-world projects are much more complex and, thus, unnecessarily difficult to present as example.

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       ret += " °C"; //bug! "degrees" is never added to "ret"
10  }
```

```
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       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       if(degrees >= 0)
10        ret = Integer.toString(degrees);
11      ret += " °C"; //bug! "degrees" is never added to "ret"
12  }
```

Figure 6.9: Example of three overfitting plausible patches. The first patch creates a fix only for the execution described by Test5 from Figure 6.8. The following two patches describe code changes that satisfy the description of Test5 and a superset of executions (`degrees <= 0` in the second patch and `degrees >= 0` in the third patch correspondingly).

Figure 6.9, therefore, presents a case where three diverse patches can be created by following the same partial specification (the test suite in Figure 6.8). Figure 6.10 exhibits an example of how these low-quality overfitting patches described in Figure 6.9 can be consolidated (through code-merging), 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.

Consolidated higher-quality patch

```
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 6.10: Example of a higher-quality consolidated patch created from the three plausible patches in Figure 6.9 that generalizes to the intended program behavior.

6.2.1 Patch Consolidation Through N-Version Voting

In this section, we analyze the impact software consolidation has on patch quality. We want to understand whether a merge strategy that performs like the average of the patches, if it existed, would be beneficial for patch quality. Thus, the next research question we analyze is the following:

> **Research Question 11:** How does the quality of patches consolidated through n-version voting compare to the quality of single patches?

Previous studies [187] have suggested n-version voting as a possible way to analyze how randomness can be exploited in benefit of automatic program repair. Following this vision, we replicated a previous study [187] with a broader sample of defects and modifying the provenance and complexity of the defects. In the original study, defects were created to teach an introduction to programming to a class of novice developers. These were simple and small examples of buggy software. In our study, we modify the origin and complexity of the defects to real-world and highly complex defects from highly scrutinized open-source projects.

The intuition behind n-version voting patch evaluation is that given a set of patches for a defect and given a set of held-out tests to evaluate the patches, if correct behavior is present in most patches, the consolidated patch will exhibit that correct behavior as well. Similarly, if most patches show overfitting behavior, then the consolidated patch will show overfitting behavior as well. Therefore, per each test, all individual patches vote on how the consolidated patch would behave based on their own behavior (pass or fail), and the most common behavior prevails. This technique allows us to evaluate the quality of a consolidated patch based on how its corresponding patches behave as a validation of how patch consolidation can impact patch quality.

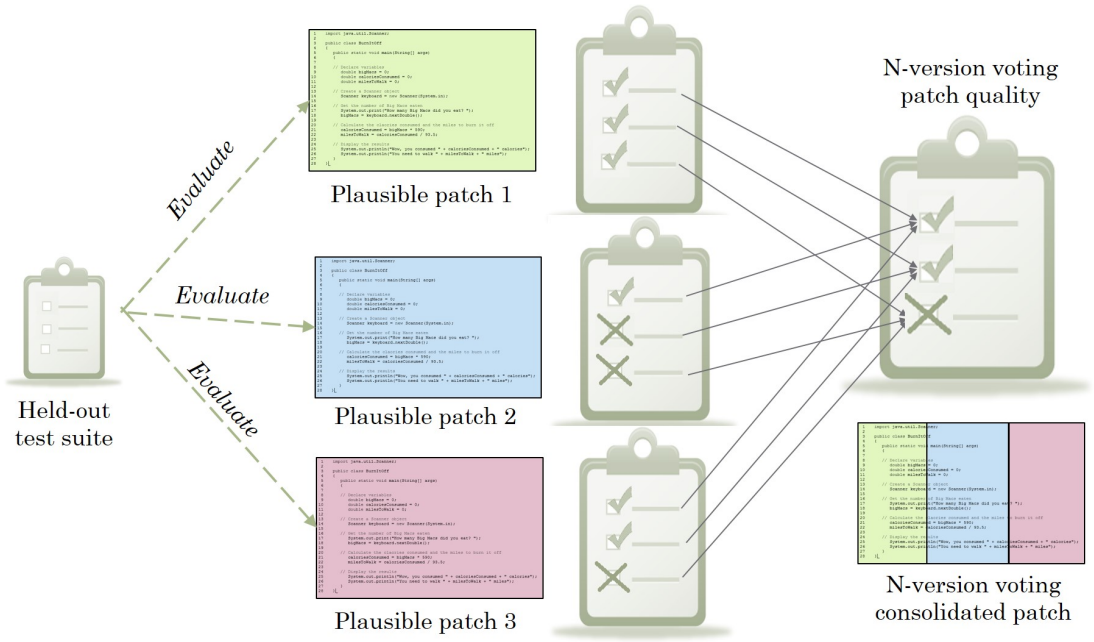


Figure 6.11: Patch Quality N-Version Voting: Each individual patch is evaluated using the held-out test suite. To evaluate the behavior of the consolidated patch when evaluated in the held-out test suite, per each test in the held-out test suite, each individual patch votes on how the n-version patch behaves (pass or fail the test). If a strict majority of individual patches passed the test, then the consolidated n-version patch is said to pass the test, and conversely, if the majority of patches fail the test (or there is a tie), the consolidated patch is said to fail the test.

Methodology: For this research question we used the most comprehensive corpus of patches we have produced (described in Section 4.1) generated by the three repair techniques implemented in JaRFly. These patches were created by running these repair techniques on all 357 Defects4J defects, which produced 561 unique patches (255 by GenProg, 107 by PAR, and 199 by TrpAutoRepair; recall Figure 4.1 (page 38) and Table 4.1 (page 37)).

Following previous studies [187], for each technique, we identified the defects for which that APR approach produced at least 3 distinct patches. For these defects, we then executed the held-out test suite on each patch in the population. Given each test of the held-out test suite, each individual patch then votes on how the n-version consolidated patch behaves based on how the individual patch behaves. If the majority of the patches vote to pass the test, the consolidated patch passes the test. Otherwise, if the majority of patches fail (or tie), then the n-version patch fails as well. In case of ties, the test fails in the n-version patch. For GenProg, 30 defects qualified for this experiment, 9 for PAR, and 25 for TrpAutoRepair.

More formally: for each bug b_i, consider the set of patches P_i that pass its guiding test suite (plausible patches). P_i is composed of patches p_i^j where i denotes 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 in G_i pass when evaluated in each p_i^j in P_i by definition of plausible patches in the APR algorithm. We evaluate the quality of each patch p_i^j by analyzing the percentage of tests passed in H_i.

The n-version voting consolidated patch behaves the same way as the majority of the patches it is generated from. The intuition is that if correct behavior is present in most patches, the consolidated patch will contain that correct behavior as well, and likewise with overfitting behavior. The most common behavior prevails. N-version voting, as a whole, is a process that provides an initial high-level validation of how patch consolidation might impact patch quality. Therefore, in this research question we analyze the quality of each n-version patch y_i for each bug b_i, where y_i is a product of the combination of all plausible patches p_i^j in P_i. P_i may contain different number of patches p_i^j on each bug. This is of minor relevance since what is being evaluated in n-version voting is the predominant behavior in the majority of the individual patches p_i^j. The behavior of y_i in each test case h_i^l is thus defined by the most common behavior in each plausible patch p_i^j in each test case h_i^l.

Finally, we compare the performance of the consolidated patch to the performance of individual patches P_i. The quality of both populations of patches is described by executing the same held-out test suites. Given that these populations are of different sizes (one n-version voting consolidated patch is composed of several individual patches), then we compare the quality of these two populations using 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* (i.e., δ or d) to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95% confidence interval (CI) of the delta estimate.

Results: Figure 6.12 and Table 6.2 compare the quality of the consolidated patches to the individual patches that make up those consolidated patches for all bugs in each APR approach. Table 6.2 describes per each APR approach, the minimum, mean, median and maximum quality of the patches found for both the individual and the n-version (consolidated) patches. It also includes the percentage of patches that pass 100% of the held-out test cases (described as 100%-quality). Figure 6.12 shows the quality distribution of the patches per each APR approach (individual and consolidated), and shows the p-value, delta estimate and 95% confidence interval of the delta estimate. The Mann-Whitney U test indicates the differences between the patch quality of the individual patches and the n-version (consolidated) patches are not statistically significant given

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%

Table 6.2: Quality of the individual patch in comparison to the quality of the n-version patches made up from the test case behavior from the individual patches patches perach repair approach [149].

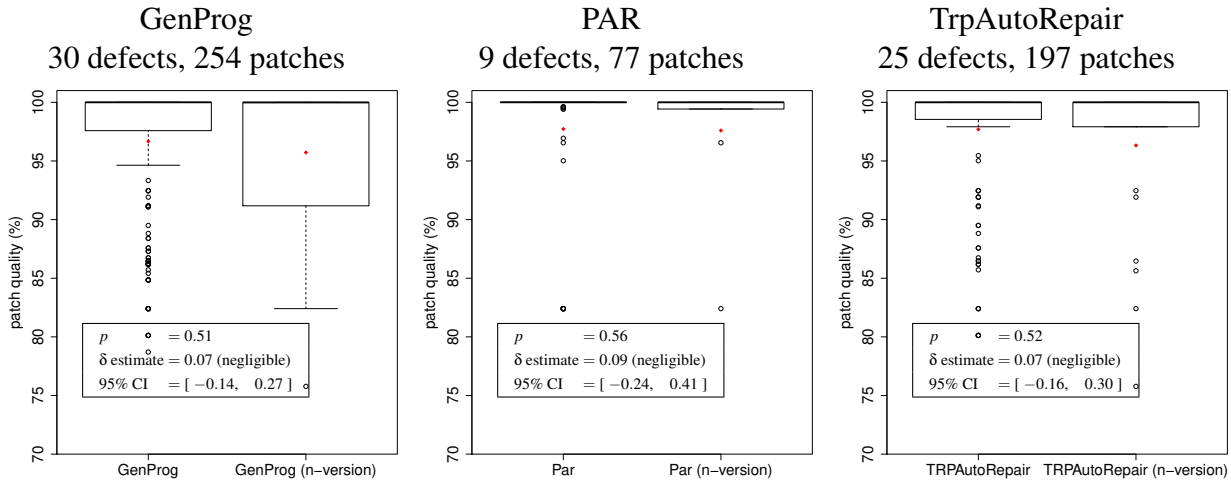


Figure 6.12: The box-and-whisker plots compare the quality of the individual and n-version programs made up of those patches, with the mean as a red diamond. The p values (Mann-Whitney U test) suggest that there is no statistically significant difference in the quality of n-version and individual programs. 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 [149].

that the p-value is greater than 0.05 in every case. The Cliff's δ estimate is positive, indicating that there is an increase in quality in the population of consolidated patches when compared against the individual patches. However, the Cliff's delta is less than 0.1 in all cases, suggesting the magnitude of the difference between populations is negligible. This is also confirmed by the 95% confidence interval.

Our results thus suggest that given the patches generated from our studied APR techniques, the difference in quality performance between an n-version patch and individual patches is positive (consolidated patches have higher quality than individual patches), but this cannot be confirmed with a 95% confidence. This similarity in quality between consolidated and individual patches

happens given that patches generated for a single defect tend to be very similar themselves and behave similarly, thus analyzing their behavior when combined results in similar behavior and similar quality.

Answer to Research Question 11: The quality of n-version voting consolidated patches is negligibly higher than individual patches. Individual plausible patches are very similar; therefore, their corresponding consolidated patches are also very similar.

Our findings are consistent with a prior study using the same consolidation technique [186]. 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 [186]. This study does not explore this aspect given that the patches we observe for the Defects4J defects tend to be of relatively high quality.

6.2.2 Patch Consolidation Through Code-Merging

In the previous section we analyzed how a consolidation mechanism could benefit patch quality by understanding the behavior of an n-version patch using n-version voting. N-version voting is thus, a more theoretical approach towards understanding the impact software consolidation can have in patch quality. In this section, we analyze a more practical approach where we consolidate pairs of patches using code-merging and evaluate the quality of the resulting code-merged patches. This experiment allows us to create a patch that can be analyzed, executed and tested. Therefore, we propose the following research question:

Research Question 12: How do code-merging consolidated plausible patches behave in relation to their non-consolidated counterparts in terms of patch quality?

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. Our goal is to provide evidence of how code-merging can be used to create consolidated patches and analyze the quality of said patches as compared to their non-consolidated couterparts.

In this section we create the set of consolidated plausible patches C_i, where each consolidated patch $c_i^{n,m}$ is composed of patches p_i^n and p_i^m being combined, where $n \neq m$.

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 [10]. We use *Structured merging*, a three-way merging mechanism provided by JDime that considers the common ancestor (the buggy version, in our case) of the programs being merged. Structured merging also considers the abstract syntax tree of the program (e.g., matching brackets), and uses JastAddJ, [1] an extensible Java compiler to reason about conflict detection given the data structure of the program (e.g., whether the order of two nodes makes a difference in the semantics of the program).

[1] http://jastadd.org/web/

Finally, we compare the quality of the consolidated 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 consolidated patch $c_i^{n,m}$ and its corresponding individual plausible patches p_i^n and p_i^m. We then report the percentage of consolidated patches whose quality is higher than one or both its corresponding individual patches. Later we perform an homologous analysis reporting the percentage of consolidated patches whose quality is worse than its corresponding non-consolidated patches.

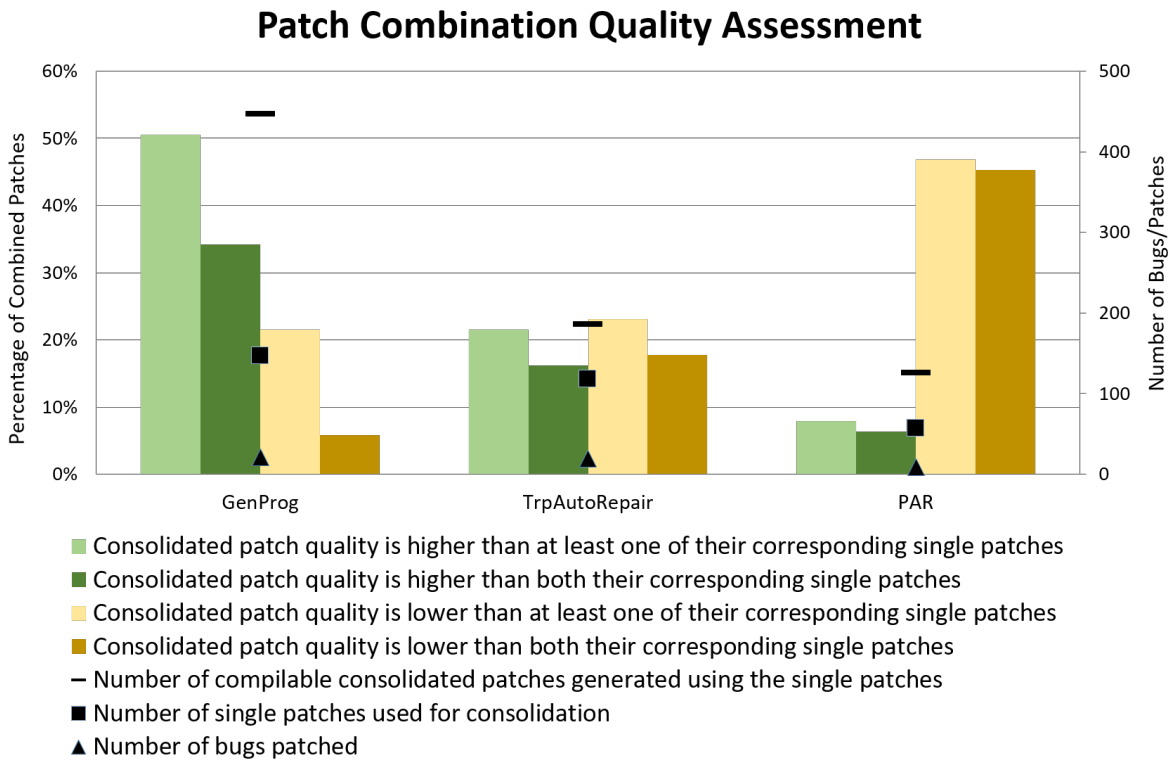


Figure 6.13: Quality assessment of patch combinations and their corresponding individual plausible patches using three APR approaches: GenProg (GP), TrpAutoRepair (TRP) and PAR. The bars describe the quality of the consolidated patches using the left axis, the markers describe the number of consolidated patches created, number of single patches used and number of bugs patched.

Results: In Figure 6.13, colored bars follow the left y-axis. Up to 51% of the consolidated patches show higher quality than at least one of their corresponding individual plausible patches. This implies that code-merging consolidation has the capacity to increase the quality of at least one of its corresponding in over half of the consolidated patches analyzed.

It is worth noticing that one possible explanation for this behavior is that when merging two patches, the functionality of the higher-quality patch is predominant and the behavior of the lower-quality patch becomes redundant behavior that can be eliminated through code reduction. This may be useful to reduce the time it takes to perform human patch inspection. In industrial setting where APR techniques can be applied to generate patches, there are typically no automated oracles used to evaluate the quality of our automatically generated patches. Therefore, human inspection

is the foreseen method required to validate the automatically generated patches. Current APR approaches in industrial settings rely on human inspection of generated patches [132]. The results shown in this experiment imply that we can reduce the time it would take a human developer to inspect these patches if we can consolidate several patches into a single one and remove the redundant or unnecessary functionality.

Figure 6.13 shows that up to 36% of the consolidated patches show higher quality than both of their corresponding individual plausible patches. These results show considerable potential in patch consolidation as a means to automatically improve plausible patch quality. Unlike the case discussed above, in these cases code-merging consolidation was able to successfully increase the quality of the patch as compared to its corresponding non-consolidated counterparts. A possible explanation for these can be cases like the one described in Figure 6.10, where both individual patches fix overlapping but distinct portions of the problem space, and when consolidated the resulting patch covers both fixes creating a solution that covers a superset of both patches.

The percentage of consolidated patches whose quality *increase* by consolidating in comparison to their corresponding single patches is higher for GenProg patches. From the three analyzed techniques, GenProg is the approach with more diversity techniques in its patch creation process, given that it has both, the ability to create multi-edit patches and uses coarse-grain operators in the patch generation process, which are more diverse than fine-grain (template-based) mutation operators. The percentage of quality increase of GenProg patches is followed by TrpAutoRepair plausible patches. TrpAutoRepair uses the same coarse-grain mutation operators as GenProg therefore providing still a way to create diverse patches, however, 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 based on templates; therefore, its ability to generate diverse patches is reduced even further. Thus, we empirically notice in Figure 6.13 a correlation between the probability for patch consolidation to increase the quality of plausible patches and 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 6.2) and with previous work [186].

Answer to Research Question 12: The impact code-merging consolidation has on patch quality varies depending on the patches being consolidated. In our study, code-merging consolidation was especially beneficial for the patches generated by GenProg, the approach with the most tools to generate diverse patches.

Chapter Summary: This chapter shows the advantages of software diversity and patch consolidation in the APR process and how diversity affects patch quality. We start by creating a series of techniques to incentivize and exploit diversity in the APR process as a means to increase the quality of the best patch found in a patch population. Later, we explore the idea of a consolidated patch using n-version voting and examining how test cases behave when evaluated in populations of patches for the same bug. Finally, we create a set of consolidated patches using code-merging and analyze their behavior and quality when compared to the individual plausible patches they are composed of.

Overall, we found that consolidation through n-version voting shows a negligible increase in patch quality when compared to individual patches due to lack of diversity in the patches generated.

N-version programming (independent groups of developers implementing software based on the same program description) has also shown lack of diversity [16], thus, following a similar set of results from the ones described in our experiment. In industrial environments, validation of automatically generated patches is still an open challenge that the techniques described in this chapter have to face. Developer manual inspection of the generated patches is still considered the best option for patch validation.

Certain techniques discussed in this chapter, especially slicing test cases, seem to be beneficial for increasing the quality of the best patch in a population. Similarly, we found that consolidating patches through code-merging 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. It includes 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 using patch diversity and consolidation as a means to increase quality of patches in APR.

The main findings in this thesis are:

- Automatic program repair techniques are able to generate plausible patches when executed on real-world defects. Using our open-source framework JaRFly, we were able to generate 106 patches for the 357 analyzed defects. These patches generated typically overfit to the guiding test suite (e.g., for GenProg 75.7% of the patches overfit, for TrpAutoRepair 80.5%, and for PAR 86.2%), and often break 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. We can create diversity-driven techniques in APR with the aim of increasing the quality of the best patch in a population. Similarly, we analyze how patch consolidation can be used to increase the quality of individual plausible patches and how this approach 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 that allows us to navigate the search space in different ways 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 [112, 167] 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 [209]. Finally, some tools also have environmental changes that prevent us from using them. For example, ACS [214] 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 [54]. 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 JaRFly currently implements *only* three reimplemented techniques. JaRFly is an extensible open-source repair framework for Java. It implements a diverse set of APR approaches that vary in mutation operator usage, search strategy, and number of edits per patch. It is also worth noting that we compare our results against more recent APR approaches such as Nopol [188] and SimFix [149].

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 thesis 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 dissertation. 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 thesis 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 diverse set 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 [65] and Randoop [159], 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 partial 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 cannot guarantee that the human-created patch is perfect [187]. 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 [65] 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 state-of-the-art test suite generation tool mitigates the risk of bias in manually constructing evaluation test suites.

Overall, our methodology follows the guidelines for evaluating randomized algorithms [12] and uses repair techniques' configuration parameters from prior evaluations that explored the effectiveness of those parameter settings [99, 112, 167]. 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 dissertation we analyzed the problem of the low-quality plausible patches created by heuristics-based 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 discard 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 [5].

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 [25, 147] are helpful.

We observed that Java heuristics-based repair techniques produce patches for more defects than C heuristics-based 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. The problem this thesis 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 a partial program specification (perhaps using state machines or natural language processing) where there is more overhead in terms of programmer tasks to define said partial specification, but in the long run require less maintainability given that they need to meet a higher bar to be considered plausible patches. 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 dissertation 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 [138, 139, 152, 216] 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 thesis. A possible proposed approach is to incentivize syntactic difference instead of semantic difference as in this document. 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 6.1). 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 thesis, we look into code consolidation 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 the 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). However, even when it can consolidate them, it 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. A majority of the corpus can be used as guiding test suite and the remaining of the corpus used to guide consolidation efforts and validate quality. This is similar to how in machine learning literature [172] 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An Eclipse plug-in for testing and debugging. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 378–381, Essen, Germany, September 2012.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

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

[68] 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.

[69] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In *International Conference on Software Engineering*, ICSE '15, 2015.

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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, 2017.

[76] 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.

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

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

[79] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kgenprog: A high-performance, high-extensibility and high-portability apr system. In *Asia-Pacific Conference on Software Engineering*, 2018.

[80] 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.

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

[82] 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.

[83] 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.

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

[85] 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.

[86] Jiajun Jiang. SimFix implementation. `https://github.com/xgdsmileboy/SimFix/`, 2017.

[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] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In *International Symposium on Fault-Tolerant Computing*, FTCS'99, page 30, 1999.

[102] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using

dependence analysis. In *International Conference on Software Maintenance*, 2002.

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

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

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

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

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

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

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

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *International Conference on Automated Software Engineering*, 2001.

[132] 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.

[133] 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. *Springer Empirical Software Engineering*, 21, 2016.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] MeCC: memory comparison-based clone detector. H. kim and y. jung and s. kim and k. yi. In *International Conference on Software Engineering*, 2011.

[142] Elliot Mendelson. *Introduction to Mathematical Logic (Second edition)*. 1979.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] 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.

[148] 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.

[149] Manish Motwani, Mauricio Soto, Yuriy Brun, Rene Just, and Claire Le Goues. Quality of automated program repair on real-world defects. page to appear, 2020.

[150] 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.

[151] 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.

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[158] 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.

[159] 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.

[160] 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.

[161] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. In *Software Testing, Verification and Reliability*, pages 605–628, 2015.

[162] 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.

[163] 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.

[164] 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.

[165] 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.

[166] 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.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] Gopinath Rebala, Ajay Ravi, and Sanjay Churiwala. *An Introduction to Machine Learning*. 2019.

[173] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. volume 74 of *American Mathematical Society*, pages 358–366, 1953.

[174] 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.

[175] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. 2010.

[176] 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.

[177] 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.

[178] 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.

[179] 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.

[180] 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.

[181] 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.

[182] 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.

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

[184] 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.

[185] 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.

[186] 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.

[187] 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.

[188] 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.

[189] 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.

[190] 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.

[191] 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.

[192] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[193] 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.

[194] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, 2002.

[195] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. Towards semantic clone detection via probabilistic software modeling. In *International Workshop on Software Clones*, 2020.

[196] 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.

[197] 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.

[198] 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.

[199] 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.

[200] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to design a program repair bot? insights from the repairnator project. In *Proceedings of Intern. Conf. on Software Engineering, In Track Software Engineering in Practice*, 2018.

[201] U. Voges. *Software Diversity in Computerized Control Systems*, volume 2. 1988.

[202] 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.

[203] 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.

[204] 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.

[205] 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.

[206] 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.

[207] 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.

[208] 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.

[209] 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.

[210] 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.

[211] 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.

[212] 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.

[213] 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.

[214] 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.

[215] 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.

[216] 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.

[217] 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.

[218] 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.

[219] 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.

[220] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of Java programs via multi-objective genetic programming. ArXiv, 2017.

[221] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Software Engineering*, 2002.

[222] 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.