Effective Automatic Program Repair Based on State Abstraction

Liushan Chen

PhD

The Hong Kong Polytechnic University

2021

The Hong Kong Polytechnic University

Department of Computing

Effective Automatic Program Repair Based on State Abstraction

Liushan Chen

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

August 2020

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

__________________________(Signed)

Liushan CHEN (Name of student)

Abstract

Automatic program repair (APR) aspires to automate the otherwise expensive and laborious process of patching bugs. Most existing APR techniques use tests to drive the repair process, where a buggy program with passing and failing tests is used as the input to generate a number of candidate fixes that can make all the tests pass as the output. The test driven APR techniques can be categorized into two groups: synthesis-based techniques and search-based techniques.

While exciting progress has been made in search-based automated program repair and some of such techniques have been successfully applied in industry in the past few years, several limitations are preventing those techniques from being more effective and efficient in fixing more bugs: The quality of generated fixes may not be satisfactory from a programmer's point of view since test cases are just weak oracles for program correctness; Existing fault localization techniques deliver only subpar efficiency that cripples the performance of APR techniques because the localization process is not tightly integrated with the repair process; The generic information about fix patterns learned from human-written patches in the past may not be enough to guarantee an effective and efficient fixing process.

We have developed novel techniques in this thesis that address these limitations and advance the state-of-the-art in search-based automated repair of Java programs. Particularly, we devised the JAID technique that abstracts program states based on a rich set of predicates derived from regular Java code and mitigates the overfitting problem by grounding the repair generation and validation processes on the abstraction; We pioneered the RESTORE technique that reuses the outcome of failed patch validation to yield accurate information about fault locations without incurring onerous computational costs; We introduced the PRIDE technique that enables program repair to apply valuable knowledge repeatedly learned from both completed and on-going fixing processes to better navigate the space of candidate fixes. Supporting tools have also been implemented to enable programmers to use these techniques in their everyday development to generate high quality fixes to errors in Java programs in an automated fashion.

To evaluate the effectiveness and efficiency of these techniques, we applied the tools to automatically repair bugs from benchmarks like DEFECTS4J, INTROCLASS-JAVA, QUIXBUGS, and BEARS. Experimental results show that, compared with other Java APR tools of the same time, ours can produce correct fixes to more bugs with a comparable amount of time.

Keywords: Automated program repair; Fault localization; Predicate abstraction; Mutation analysis; Learning to rank.

Publications Arising from the Thesis

Journal Papers

- **Liushan Chen**, Yu Pei, Carlo A. Furia, Contract-Based Program Repair without The Contracts: An Extended Study. IEEE Transactions on Software Engineering (TSE).
- Tongtong Xu, **Liushan Chen**, Yu Pei, Tian Zhang, Minxue Pan, Carlo A. Furia, RESTORE: Retrospective Fault Localization Enhancing Automated Program Repair. IEEE Transactions on Software Engineering (TSE).

Conference Paper

- **Liushan Chen**, Yu Pei, Carlo A. Furia, Contract-based Program Repair without the Contracts. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17), pp. 637-647, 2017.

Paper Submitted

- **Liushan Chen**, Yu Pei, Minxue Pan, Tian Zhang, Carlo A. Furia, Qixin Wang, Program Repair with Repeated Learning.

Acknowledgements

First and foremost, I feel profound gratitude towards my supervisor, Dr. Yu Pei, for all his meticulous effort through every stage of my study. Learning and working under Dr. Pei's supervision is a precious experience of a lifetime. He is always optimistic, energetic, and inspiring, which profoundly influenced me. I have benefited a lot from our countless discussions; Dr. Pei is always full of fascinating ideas that can turn a difficult problem I encountered into our chance, and he always welcomes and respects different ideas. When I look back, I found that it was one of my best decision to choose him as my supervisor.

I would like to thank my co-supervisor, Dr. Qixin Wang. His systematic method and rigorous attitude to research provide me a great scholar model. I also learned many things from him, including writing skills and research strategy.

I am hugely appreciative of Dr. Carlo A. Furia, who offered much valuable advice on our work. Without his generous help, it would be much more difficult for us to achieve what we have achieved. Also, I would like to thank Tongtong Xu, who provided selfless support during our collaboration.

I also want to thank my family and all my friends, especially my parents, for supporting me with their love. Special mention goes to my closest friends Zhu Gong, Suyu Dai, and Fei Yin. All the chicken nights and conversations are releasing; our friendships are gifts of life, which I cherish. Finally, I want to express my sincerest gratitude to Zejin Zhang, who always encourages me to breakthrough my limitations

and stay with me in my darkest time in all means.

Table of Contents

Certificate of Originality iii

Abstract iv

Acknowledgements vii

List of Figures xii

List of Tables xiii

1 Introduction 1

1.1 Main Results 3

1.2 Terminology 5

1.3 Structure 5

2 Related Work 7

2.1 Automated Program Repair 7

2.1.1 Search-Based Automated Program Repair 9

2.1.2 Synthesis-Based Automated Program Repair 14

2.2 Fault Localization 15

3 Contract Based Program Repair without the Contracts - Jaid 18

3.1 An Example of Jaid in Action 20

3.2 How Jaid Works 22

3.2.1 Overview 22

3.2.2 Program State Abstraction 23

3.2.3 Fault Localization 27

3.2.4 Fix Generation: Fix Actions 28

3.2.5 Fix Generation: Candidate Fixes 31

3.2.6 Fix Validation 32

3.2.7 Fix Ranking 33

3.3 Experimental Evaluation 34

3.3.1 Experimental Design 36

3.3.2 Experimental Results 44

3.4 Summary 68

4 Enhancing Program Repair with Retrospective Fault Localization-Restore 69

4.1 An Example of Restore in Action 71

4.2 How Restore Works 72

4.2.1 Overview 73

4.2.2 Retrospective Fault Localization 75

4.2.3 Initial fix generation 76

4.2.4 Partial fix validation 77

4.2.5 Mutation-based fault localization 77

4.2.6 Retrospective loop iteration 80

4.2.7 Final fix generation 81

4.2.8 (Full) fix validation 81

4.3 Experimental Evaluation 82

4.3.1 Experimental Design 82

4.3.2 Experimental Results 86

4.4 Summary 94

5 Directing Patch Search with Repeated Learning - Pride 96

5.1 An Example of Pride in Action 98

5.2 How Pride Works 100

5.2.1 Overview 100

5.2.2 Fault Localization 102

5.2.3 Fix Generation 102

5.2.4 Static and Dynamic Fix Analysis 103

5.2.5 Fix Ranking 105

5.2.6 Implementation 111

5.3 Experimental Evaluation 112

5.3.1 Experimental Design 112

5.3.2 Experimental Results 117

5.4 Summary 124

6 Conclusion and Future Work 125

6.1 Main Contributions 125

6.2 Future Work 126

Bibliography 128

List of Figures

3.1 An overview of how Jaid works 24

3.2 Schemas used by Jaid to build candidate fixes 32

3.3 Distribution of the top rank of correct fixes of bugs. 50

3.4 Distribution of Jaid's running time per bug 53

3.5 Distributions of Jaid's running time per bug 54

3.6 Distribution of Jaid's running time for the first valid fix 55

3.7 Distribution of Jaid's running time for the first correct fix 56

3.8 Partitioning of Defects4J bugs according to which tool can fix them 61

3.9 How effectiveness and efficiency of Jaid depend on the number of snapshots used for fixing 64

3.10 Violin plots of the top rank of correct fixes for all bugs 65

3.11 Violin plots of the overall running time 66

4.1 An overview of how Restore works 73

4.2 Visual explanation of linear regression lines 86

4.3 Comparison of Jaid and Restore on various measures 89

5.1 Two sharpening fixes generated by Pride (to be inserted before line 3 in Listing 5.1). 100

5.2 Overview of Pride. 101

5.3 Comparison of Pride and Restore on various measures. 120

5.4 Comparison of SimFix∗ and SimFix on T2C. 123

List of Tables

3.1 Defects4J benchmark 37

3.2 IntroClassJava benchmark 37

3.3 QuixBugs benchmark 37

3.4 Automated program repair tools for Java used in the comparison with Jaid 42

3.5 Spectrum-based fault localization algorithms used to replace Jaid's 42

3.6 Effectiveness of Jaid 45

3.7 Bugs in Defects4J and QuixBugs for which Jaid failed to produce any valid fix 46

3.8 Statistics on Jaid's overall running time per bug 52

3.9 Statistics on Jaid's running time 57

3.10 A quantitative comparison of Jaid with 16 other tools 58

3.11 Templates used more commonly by Jaid to produce *correct* fixes . . 63

3.12 How Jaid's effectiveness changes using different fault localization techniques 65

4.1 A quantitative comparison of Restore with 13 other tools 87

4.2 How retrospective fault localization achieves progress 94

4.3 How many times retrospective fault localization iterates 94

5.1 Format of the training data. 110

5.2 Experimental results of Pride and Restore in repairing faults from the 4 benchmarks. All times are in minutes. 117

5.3 A quantitative comparison of Pride with 17 other tools in repairing bugs from the 4 benchmarks. 119

5.4 Usefulness of individual features. 122

Chapter 1

Introduction

Software programs have been widely integrated into almost every aspect of modern societies. Even though people are already investing a huge amount of time and effort on debugging those programs, bugs that escape programmers' attention and find their way into the field still cost roughly $60 billion dollars annually in just the USA.

Studies on bugs fixed by programmers in the past show that a significant percentage of bugs could be corrected with simple changes, i.e., changes that affect only a few lines of code. In view of that, considerable research work has been devoted to automatically generating such corrections, with the hope that reducing the cost of fixing simple faults could help programmers and companies better allocate their resources and focus on more challenging faults.

Most existing automated program repair (APR) techniques are test-driven. More precisely, they require as input a faulty program as well as a list of tests as the oracle for the correctness of the program: The failing tests reproduce the failures caused by the fault to fix, while the passing ones represent correct behaviors the program should preserve after fixing. The goal of automated program repair is then to generate fixes that, when applied, could make all the input tests pass.

Test-driven APR techniques can be grouped into two categories: synthesis-based techniques and search-based techniques. Synthesis-based techniques first construct constraints that encode the expected semantics of the program, and then synthesize candidate fixes based on solutions to the constraints. Synthesis-based APR techniques have focused mostly on repairing incorrect expressions in programs so far. Search-based (a.k.a. generate-and-validate) APR techniques typically produce fix suggestions in three steps, namely fault localization, fix generation, and fix validation, where fault localization identifies program locations where the fault under fixing might reside, fix generation produces a list of candidate fixes, while fix validation checks the validity of the generated fixes.

Exciting progress has been made in search-based automated program repair and we have seen some initial, but successful, applications of such APR techniques in industry in the past few years. There, however, are still limitations that prevent existing search-based APR techniques from being more effective and efficient in fixing more bugs:

- Since tests are only weak oracles for program correctness, fixes that make all the tests pass may still be incorrect—this is referred to as the overfitting problem in APR [73]. Code annotations like contracts have been exploited in APR to significantly help increase the fraction of correct fixes that can be generated [61]. Support for contracts, however, is limited or non-existent in most widely-used programming languages.

- Results produced by fault localization techniques employed in existing APR are not accurate enough to support efficient program repair. Particularly, existing spectrum-based fault localization techniques often used by APR tools are not tightly integrated with the repair process and deliver only subpar efficiency.

- While the generic information about fix patterns that latest approaches leverage to guide the search for correct fixes is clearly critical for the success of program repair, it alone may not be enough to guarantee an effective and efficient fixing process. After all, how a bug should be fixed is ultimately determined by the nature of the bug and the code context where it occurs.

1.1 Main Results

This thesis has developed search-based APR techniques and supporting tools to (partially) overcome these limitations. The main contributions of this thesis include the following:

- We introduce the JAID technique and tool for APR that is based on detailed, state-based dynamic program analyses - akin those employed by contract-based techniques such as AutoFix [79], but working on regular Java code (without any contracts). A key feature of JAID is that it extracts state abstractions from regular Java code and utilizes the state abstraction to help construct high-quality fixes.

  We also conduct an experimental evaluation to assess the effectiveness and efficiency of JAID, as well as to better understand the trade-offs involved in APR design and discover possible directions for future work. The evaluation results show that JAID perform comparably to or even better than existing automated Java program repair tools in terms of the number of faults correctly fixed (as of May 2017). JAID is the first APR technique that achieves high levels of precision without relying on additional input other than tests and the faulty code; in contrast, other high-precision APR techniques of that time [34], [89] analyze a large number of project repositories to collect additional information that guides fixing.

- We present *retrospective fault localization*, a novel fault localization technique geared to the requirements of automated program repair. Retrospective fault localization leverages mutation-based fault localization [56], [59] to boost localization accuracy. Since mutation-based fault localization is notoriously time-consuming, the key idea of retrospective fault localization is to perform it as a derivative of the usual program repair process. Precisely, retrospective fault localization introduces a feedback loop that reuses the candidate fixes that fail validation to enhance the precision of fault localization. By reusing the outcome of failed patch validation to support mutation-based dynamic analysis, retrospective fault localization is able to provide accurate fault localization information without incurring onerous computational costs.

We implement retrospective fault localization in a tool called RESTORE on top of JAID. Experiments on real-world bugs from the DEFECTS4J curated benchmark [27] produced results indicating that retrospective fault localization significantly improves the overall effectiveness and efficiency of program repair. More specifically, RESTORE generated correct fixes for 41 faults in DEFECTS-4J, 8 more than any other automated repair tool for Java (as of May 2019) and it cut JAID's running time by two thirds or more.

- We devise the PRIDE (Program Repair wIth repeateD lEarning) technique for improving the effectiveness and efficiency of automated program repair. The key novelty of PRIDE is twofold. First, it defines a rich set of features that not only characterize the composition of fixes but also relate the components of fixes to their context code and test executions. Second, it introduces a learning-to-rank model to enable program repair to *repeatedly* learn valuable knowledge from both completed and ongoing fixing processes about the constitution of high quality fixes. The model is trained on data about candidate fixes from past fixing processes and updated when data about candidate fixes processed in new fixing sessions become available.

We implement the Pride technique into a tool with the same name and applied it to repair 983 bugs from 4 popular benchmarks in APR, namely Defects4J, IntroClassJava, QuixBugs, and Bears. Pride proposed valid fixes to 243 bugs and correct fixes to 137 bugs, significantly outperforming Restore and the other existing APR techniques for Java programs. To demonstrate the generality of the Pride technique, we also incorporated it into the SimFix APR technique and produced SimFix∗. Compared with SimFix, SimFix∗ proposed correct fixes to one more fault in repairing Defects4J bugs and achieved 3.3 speedup in terms of repair time needed to find the first correct fix.

One common, key feature to enable the three techniques is program state abstraction based on a rich set of boolean expressions. Such abstraction not only enables us to understand better where the fault causes might be, but also sheds light on how the programs should be changed to correct the faults.

1.2 Terminology

In this thesis, we use the nouns "defect", "bug", "fault", and "error" as synonyms to indicate errors in a program's source code. We also use the nouns "fix", "patch", and "repair" as synonyms to indicate changes we make to the program's source code in the hope of correcting bugs. Fixes that can make all the input test cases pass are called *valid* fixes. Valid fixes, however, may break other test cases not included in the input. Fixes that are semantically equivalent to patches written by programmers are called *correct* fixes.

1.3 Structure

The rest of this thesis is organized as follows. Chapter 2 reviews recent works in automated program repair and related areas. Chapter 3, Chapter 4, and Chapter 5 present the JAID, RESTORE, and PRIDE techniques as well as the experiments we conduct to evaluate them, respectively. Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Related Work

In this chapter, we review related work in areas of automatic program repair and fault localization. For each area, we first categorize the techniques available in that area to help position our work, then focus on reviewing the approaches that have directly influenced our works in this thesis. We also discuss recent APR techniques that are applicable to Java programs, since they are directly comparable to our techniques. More detailed reviews are provided in [86] for fault localization and [55, 18] for automated program repair.

2.1 Automated Program Repair

A program is called defective if its behaviors are inconsistent with its specification. Automatic program repair could refer to a wide range of techniques that utilize various program analysis and synthesis methods to produce modifications to make a buggy program satisfy its corresponding specification.

There are two kinds of systems for correcting program defects automatically under different scenarios, namely repairing and self-healing systems. Repairing systems look for patches to correct bugs at the source code level so that further failures caused by the same faults can be avoided; In contrast, self-healing systems aim to achieve run-time survival: They detect failures in-the-field and try to make modifications at run-time to prevent systems from crashing. Therefore, fixes generated by self-healing systems are usually *temporary* workarounds, while programmers are still expected to permanently fix the underlying bugs by revising the corresponding source code afterwards. Under the hood, self-healing systems often use program analysis techniques similar to those used by repairing systems. ClearView [63], for instance, dynamically infers state invariants (like JAID does) on instrumented binaries to detect and then prevent problems such as buffer overflows. Our work in this thesis focuses on repairing techniques that generate permanent patches at the source code level.

Most existing automated program repair (APR) techniques are test-driven. More concretely, they require as input a faulty program as well as a list of tests as the oracle for the correctness of the program: The failing tests reproduce the failures caused by the fault to fix, while the passing ones represent correct behaviors the program should preserve after fixing. The goal of automated program repair is then to generate fixes that, when applied, could make all the input tests pass.

Test-driven APR techniques can be roughly grouped into two categories: search-based (a.k.a. generate-and-validate) techniques and synthesis-based (a.k.a. correct-by-construction) techniques. A process of search-based program repair typically involves three kinds of activities, namely fault localization, fix generation, and fix validation. In particular, fault localization identifies program locations where the fault under fixing might reside; fix generation produces a list of candidate fixes; and fix validation checks whether the fix makes the program pass all input tests. Fix validation is necessary for such a process since fix generation in search-based APR is usually driven by heuristics, and it provides no guarantee for the correctness of the generated candidate fixes. Our work in this thesis focuses on test-driven techniques that follow the generate-and-validate paradigm to fix general faults.

Synthesis-based techniques [53, 91, 54, 52] replace suspicious expressions in the faulty program with symbolic variables, and employ symbolic analysis to construct constraints on those variables that encode the expected input/output semantics of the program. Then they synthesize candidate fixes based on resolved solutions to the constraints. Since the constraints already reflect the expected behaviors of the input tests, candidate fixes derived from the constraints are guaranteed to make all the input tests pass. Since it is easier to find a formal representation of the repair problem while considering a specific class of faults than producing formalization for general faults, synthesis-based approaches are usually designed for a specific class of faults, such as *condition synthesis*. It, however, is not clear yet how synthesis-based APR can be extended to work on the statement level or to handle expressions with side-effects.

2.1.1 Search-Based Automated Program Repair

Many automated program repair techniques are search-based: Given a faulty program and a group of passing and failing tests, such a technique 1) locates the faulty location usually through spectrum-based fault localization, 2) generates fix candidates by heuristically searching a program space, then 3) applies generated fixes to the faulty program, and checks the validity of the candidates by rerunning all the input tests.

GenProg [82] pioneered search-based repair by using genetic programming to mutate a faulty program and generate fix candidates. It implements a genetic algorithm that mutates a faulty C program by deleting, adding, or replacing the code. Here, the new code added or used as the replacement is taken from other parts of the file or project under fixing—following the intuition [51] that ingredients needed to patch a bug often exist already in the fixing context. GenProg was substantially extended in [35] to handle code bases of realistic size and it managed to produce valid fixes for 52% of 105 bugs in an experiment conducted to evaluate the tool.

Encouraged by GenProg's promising results, various approaches tried to make the mutation of candidate fixes more effective or the exploration of the fix space better oriented and thus more efficient. For example, MutRepair [13] works in a similar way to GenProg but only modifies operators appearing in expressions (such as comparison and logical operators), since they tend to be common sources of programming mistakes.

PAR [30] bases the generation of fixes on ten patterns, which are selected based on a manual analysis of programmer-written fixes. These fixing patterns are encoded as sequences of AST rewriting rules. The intuition behind is that patterns learned from human fixes helps to generate fixes that are more readable, and possibly more acceptable from the perspective of developers.

A complementary approach [76] uses *anti*-patterns to characterize fixes that are likely to be incorrect but still pass validation. Anti-patterns help identify fixes that are frequently discarded by developers. By excluding potential false-positive candidates, anti-patterns help prune the search space and enable APR to produce fixes with higher quality.

RSRepair [65] works similarly to GenProg but uses random search instead of genetic programming during fix generation. This approach is based on the assumption that random search could perform better than evolutionary algorithms since progressive evolution might be hard to encode in APR. Moreover, to improve the efficiency of the fixing process, RSRepair runs test cases that are more likely to fail earlier when validating candidate fixes.

AE [81] enumerates variants of the faulty code systematically and uses simple semantic equivalence checking to reduce the number of fix candidates that have to be validated. The equivalence checking considers two fix candidates to be equivalent if they only differ in any of the following elements: 1) duplicated statements or variable names; 2) dead code; 3) instructions that could be reordered without affecting the semantic of the program. With the semantic equivalence checking, AE is able to discard a certain amount of candidates despite different syntax, thus saving expensive evaluation operation time.

Search-based APR tools described above are capable of working on real-world bugs. However, they tend to *overfit* the input tests [73]—thus generating many fixes that pass validation but are not actually correct [67]. Other tools address this problem by exploiting *additional information* about the properties or patterns a correct code should satisfy or follow.

AutoFix [79, 62, 61] relies on contracts (specifications embedded in the program text) to localize the faulty program states and facilitate the generation process to produce fixes steering the program away from the faulty state. By exploiting code contracts that encode partial program specifications, AutoFix was the first general-purpose APR technique to be able to generate correct fixes to a significant number of faults. Our work JAID generalizes AutoFix's state-based analysis to work on Java code without contracts and improves the quality of the generated fixes while without sacrificing applicability.

Another source of additional information that is commonly used to assist program repair is developer-written fixes from software repositories. HDA [34] adopts a stochastic search approach inspired by generic programming to evolve a patch for a given faulty program. In the approach, fix candidates are generated by applying a collection of popular mutation operators to the original program, and mutants that match the heuristics learned from fix history are selected into the next generation.

Genesis [43] learns templates that abstract away program details for code transformations from human patches to capture common change patterns. During the fixing process, Genesis instantiates learned templates to generate new fixes for unseen applications. And it navigates the trade-off of search space coverage and searching efficiency by an integer linear program(ILP). The ILP maximizes the number of training patches covered by the inferred search space while bounding the number of generated candidate patches.

Elixir [70] specializes in repairing buggy method invocations; it involves various templates to handle different variants of method invocations. Elixir also trains a model with developer-written patches to predict the correct probability of a patch from the relevance between the patch and its context. The machine-learned model is then used to prioritize the most effective repairs.

ssFix [88] matches contextual information at the fixing location to a database of developer-written fixes, and uses code chunks structurally similar and conceptually related to the faulty context to drive fix generation. The assumption behind is that correct fixes for a fault are contained in the similar chunks of developer-written fixes.

SimFix [25] is similar to ssFix in that they both leverage similar code chunks for curbing the search space. SimFix looks for similar code chunks from the buggy program, rather than from a given codebase; it derives needed modifications from the differences between the buggy code chunk and the found similar code chunk. SimFix also mines a code base for common fix patterns. It matches the derived modifications with minded fix patterns to guide the fix generation. The design of leveraging patterns learned from developer-written fixes together with similar code chunks from the faulty program helps SimFix achieve higher precision with good efficiency.

CapGen [83] improves the effectiveness of expression-level fix generation by leveraging fault context information together with the learned patterns from open source projects to prioritize ingredient expressions and fixing operations (a.k.a., patterns in a finer granularity) respectively. Therefore, fixes more likely to be correct are generated first. The important assumption behind is that the correct fixes share similar context with buggy elements.

Hercules [71] proposes fixes to multi-hunk bugs in two steps. First, it identifies a set of repair locations with similar code that should undergo similar changes according to their context and the git history of the buggy project. Then, it generates candidate fixes that introduce changes to all related locations at the same time.

TBar [41] investigates how fix patterns from the literature contribute to the performance of APR tools. It systematically applies most of the existing fix patterns summarized from the literature to generate fixes during its fixing process.

Existing search-based APR tools are quite effective in generating correct fixes for real bugs, and several of them achieve so by mining developer fixes as *additional information*. Techniques that we develop in this thesis, however, do not need extra source of information in addition to the project being fixed. The idea of mining programmer-written code is therefore also applicable to all the three techniques JAID, RESTORE, and PRIDE, to provide additional information and further mitigate the risk of overfitting.

Most of these approaches employ off-the-shelf spectrum-based fault localization techniques to obtain a list of program entities that might contain the fault, each of which is then used as the target for fix generation in order. Timperley et al. [77] investigated whether mutation-based fault localization can help search-based APR achieve better efficiency. No significant improvement, however, was observed in the experiments conducted on bugs from the BugZoo benchmark[1], supposedly because the single-edit mutations used in the study may be too simple to reveal substantial differences between programs variants. In our retrospective fault localization, we integrate mutation-based fault localization into the RESTORE search-based APR technique where candidate fixes are treated as "higher-order" program mutants. In this way, RESTORE benefits from the additional accuracy of mutation-based fault localization while without suffering from the extra, high overhead typical of mutation-based analysis.

Compared with techniques like HDA, SimFix, and CapGen that mine additional

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

information from manual patches, PRIDE repeatedly learns valuable knowledge from both completed and ongoing fixing processes about the constitution of valid and correct fixes. PRIDE extracts a rich set of features to not only characterize the composition of fixes but also relate the components of a fix to its context, to its impact on test executions, and to the overall desirability of the fix. Then PRIDE trains a ranking model on data about candidate fixes from past fixing processes and update the model with data about candidate fixes produced during a running fixing session. The ranking model enables PRIDE to apply and update valuable knowledge learned from both completed and ongoing fixing processes to assist current repair task.

2.1.2 Synthesis-Based Automated Program Repair

Synthesis-based program repair techniques [53, 54, 58, 91, 33] express the repair problem as a constraint satisfaction problem, and then use a constraint solver to build fixes that satisfy those constraints. Relying on static instead of dynamic analysis makes synthesis-based techniques generally *faster* than search-based ones, and is particularly effective when looking for fixes with a restricted, simple form. Constraint-based approaches often target the synthesis of *conditions* in if-statements or loops, since changing those conditions often affects the control flow in decisive ways.

SemFix [58] is one of the early examples; it relies on symbolic execution to summarize tests and on location-based fault localization, and it synthesizes expressions in conditionals and in assignments that try to avoid triggering failures. DirectFix [53] expresses the repair problem as a MaxSMT constraint, and supports generating multi-line fixes. Both SemFix and DirectFix, however, have limited scalability. SPR [44] also combines condition synthesis with a dynamic analysis of the value each abstract conditional expression should take to make all tests pass, which helps aggressively prune the search space when no plausible repair exists. Angelix [54] addresses this problem by introducing an efficient representation of constraints, and by combining it with a symbolic execution analysis similar to SemFix's. Nopol [91] only targets conditional expressions, and uses a form of angelic debugging [95, 7] to reconstruct the expected value of a condition in passing vs. failing runs; based on it, Nopol synthesizes a new conditional expression using an SMT solver.

Prophet [46] implements a probabilistic model, learned by mining human-written patches, on top of SPR to direct the search towards fixes with a higher chance of being correct. MintHint [28] also builds a statistical model to generate repair *suggestions* consisting of expressions that may be useful in a complete fix. ACS [89] is a recent technique that significantly improves the precision of condition synthesis based on a combination of data- and control-dependency analysis, and mining API documentation and Boolean predicates in existing projects. SketchFix [23] expresses program repair as a sketching problem [75] with "holes" in suspicious statements, and uses synthesis to fill in the holes with plausible replacements. SearchRepair [29] is one of few other approaches based on semantic analysis—as opposed to the more commonly used syntactic analysis. SearchRepair relies on preprocessing a large dataset of programmer-written code snippets, and encoding their behavior as input/output relational constraints; it then generates fixes by searching the dataset for snippets that capture the desired input/output behavior. These techniques learn useful knowledge about *syntactic* features of successful fixes in the past and use that knowledge to improve the quality of fixes proposed by automated program repair.

2.2 Fault Localization

The goal of fault localization (FL) is finding positions in the source code of a faulty program that are responsible for the fault. The concrete output of a fault localization technique is a list of statements, branches, or program states ranked according to their likelihood of being implicated with a fault.

As explained earlier, most previous search-based APR approaches employ off-the-shelf fault localization techniques to obtain a list of program entities that might contain the fault. The list is then used as the target for fix generation in order. Hence, the accuracy of the fault localization result is then crucial for the efficiency of program repair, and the research in fault localization has seen a resurgence as part of an effort to improve automated repair.

Many different types of fault localization techniques have been proposed over the years, among which spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL) have been extensively researched. The basic idea of spectrum-based fault localization [57, 3] is to use coverage information from tests to infer suspiciousness values of program entities (statements, branches, or states). For example, a statement executed mostly by failing tests is more suspicious than one executed mostly by passing tests. Many automated program repair techniques use spectrum-based fault localization algorithms [82, 13, 58, 30, 61, 9]. Generating a correct fix, however, typically requires more information than the suspiciousness ranking provided by spectrum-based techniques: An empirical evaluation of 15 popular spectrum-based fault localization techniques [66] found that the typical evaluation criteria used in fault-localization research (namely, the suspiciousness ranking) are not good predictors of whether a technique will perform well in automated program repair. This observation buttresses our suggestion that fault localization should be *co-designed* with automated program repair to perform better—as we did with retrospective fault localization.

Mutation-based fault localization combines delta debugging [93] and mutation testing [24] and is an effective approach for increasing the accuracy of fault localization. MBFL techniques randomly mutate a faulty program, and assess whether the mutation changes the behavior on passing or failing tests. Metallaxis [59] and

MUSE [56, 22] are two representative mutation-based fault localization techniques. Experiments with these tools indicate that mutation-based fault localization often outperforms spectrum-based fault localization in different conditions [56, 59]. In RESTORE, we used a variant of the Metallaxis algorithm, because it tends to perform better than MUSE with tasks similar to those we need for automated program repair. A main downside of mutation-based fault localization is that it can be a performance hog, because it requires to rerun tests on a large amount of mutants. Thus, a key idea of our retrospective fault localization is to reuse, as much as possible, validation results (which have to be performed anyway for program repair) to perform mutation-based analysis.

In retrospective fault localization, a simple fault-localization process bootstraps a feedback loop that implements a more accurate mutation-based fault localization. RESTORE currently uses a spectrum-based technique for the bootstrap phase (see Section 4.2); however, other fault localization techniques (such as those based on statistical analysis [38, 8], machine learning [5, 87], or deep learning [19]) could be used instead. Even techniques that are not designed specifically for fault localization may be used, as long as they produce a ranked list of suspicious program entities. For example, MintHint [28] performs a correlation analysis to identify expressions that should be changed to fix faults. The expressions, or more generally their program locations, could thus be treated as suspicious entities for the purpose of initiating fault localization.

Chapter 3

Contract Based Program Repair without the Contracts - JAID

The work described in this chapter is published in [9] and [10].

Since search-based APR techniques use a finite collection of tests as the specification (which is usually incomplete) to guide the fixing process, the generated patches are prone to *overfit* the given tests. Indeed, experiments have repeatedly confirmed [49], [67], [73] that automated program repair techniques are prone to produce a significant fraction of *valid but incorrect* repairs. These repairs merely happen to pass all available tests but are clearly inadequate from a programmer's perspective.

AutoFix [79], [62], [61]—a contract based technique—has demonstrated an effective approach to mitigate the over-fitting problem with program state abstractions. It was the first general-propose APR to substantially increase the number of correct fixes—for 25% of 204 bugs in [61]. Using contracts, i.e., assertions specifying *pre-/ post-conditions* and *class invariants*, as the additional information enabled AutoFix to improve the precision of repair generation and validation. AutoFix targets the Eiffel programming language, where contracts are embedded in the program code and routinely written by programmers. With the programmer-written contracts, AutoFix can easily rely on the functions (used as predicates) in the contracts to build an abstraction of program state to guide the fixing. Even if the contracts used by AutoFix are far from being detailed—let alone complete—method specifications, they significantly help increase the fraction of correct fixes that can be generated.

Unfortunately, even such simple contracts are hardly ever available in most widely-used programming languages. Therefore, we generalize some of the techniques used for contract-based program repair to work effectively without user-written contracts and propose JAID—a technique and tool for automated program repair of Java programs. JAID follows the popular *generate-then-validate* approach, and it extract abstractions of program state from regular code with detailed, state-based dynamic program analyses. State abstractions then drive both the generation and the validation stages of the repairing process to produce high-quality fixes.

To assess JAID's capabilities and explore further directions, we conduct a comprehensive study including: 1) an experimental evaluation involving 693 bugs in three different benchmark suites; 2) a detailed assessment of JAID's effectiveness, efficiency and used heuristics; 3) a quantitative comparison with all available APR tools for Java; and 4) a study of main trade-offs involved in APR design.

The results of our study indicate that JAID is a competitive tool, achieving a combination of effectiveness, efficiency, and applicability that often compares favorably to the state-of-the-art. JAID is also the first APR technique that achieves high levels of precision without relying on additional input other than tests and faulty code; in contrast, other recent high-precision APR techniques [34], [89] analyze a large number of project repositories to collect additional information that guides fixing.

The experimental results also outline future directions to improve the overall performance of JAID's algorithm. For example, our experimental results suggest that JAID's overall effectiveness does not depend much on the details of its spectrum-based fault localization algorithm. They also indicate that further substantial progress would probably require to step up the precision of fault localization in a way that it can incorporate additional information (e.g., by-products of APR process).

In the rest of this chapter, we present an example in Section 3.1 to assist a detailed illustration of how JAID work in Section 3.2. The essential algorithm for program state abstraction and how state abstractions guide the fix generation and validation are also explained in Section 3.2. And Section 3.3 presents the comprehensive study that evidences the effectiveness of JAID and explores further direction. Lastly, Section 3.4 summarizes this chapter.

3.1 An Example of JAID in Action

Apache Commons is a widely used Java library that extends Java's standard API with a rich collection of utilities. Class `WordUtil` of package `org.apache.commons.lang` includes a method `abbreviate` to simplify strings with spaces: given a string `str`, lower and upper indexes `lower` and `upper`, and another string `appendToEnd`, the method returns a string obtained by truncating `str` at the first index between `lower` and `upper` where a space occurs, and replacing (or abbreviating) the truncated suffix with `appendToEnd`. For example, `abbreviate("Apache Commons library", 9, 18, "+")` returns the string `"Apache Commons+"`.

Listing 3.1 shows the implementation of `abbreviate` at commit `#cfff06bead` of the library, which is also part of DEFECTS4J's curated collection of defects (bug Lang-45). The implementation begins by handling a number of special cases but, unfortunately, it misses the case when `lower` is greater than `str`'s length: the `index` of the first occurrence of a space from `lower` will then be `-1` (corresponding to a failing search for such a space in the call at line 15), and `upper` will be greater than or equal to `lower` (possibly after being adjusted at lines 11–12), and thus also greater than `str.length()`; in these conditions, the call to method `substring` at line 18 throws an `IndexOutOfBoundsException`.

```java
public static String abbreviate (String str, int lower, int upper, String appendToEnd) {
  if (str == null) {
    return null;
  }
  if (str.length() == 0) {
    return StringUtils.EMPTY;
  }
  if (upper == -1 || upper > str.length()) {
    upper = str.length();
  }
  if (upper < lower) {
    upper = lower;
  }
  StringBuffer result = new StringBuffer();
  int index = StringUtils.indexOf(str, "␣", lower);
  if (index == -1) {
    // throws IndexOutOfBoundsException if lower > str.length()
    result.append(str.substring(0, upper));
  if (upper != str.length()) {
    result.append(StringUtils.defaultString(appendToEnd));
  }
  } else if (index > upper) {
    result.append(str.substring(0, upper));
    result.append(StringUtils.defaultString(appendToEnd));
  } else {
    result.append(str.substring(0, index));
    result.append(StringUtils.defaultString(appendToEnd));
  }
  return result.toString();
}
```

Listing 3.1: Faulty method `abbreviate` from class `StringUtils` in package `org.apache.commons.lang`.

```
> if (lower > str.length()) { lower = str.length(); }
```

Listing 3.2: Programmer-written fix to the fault in `abbreviate`.

```
> if (lower >= str.length()) { lower = str.length(); }
```

Listing 3.3: JAID's correct fix to the fault in `abbreviate`.

The maintainers of Apache Commons fixed this fault in a later version by resetting `lower` to `str.length()` to ensure that the case `lower > str.length()` never occurs (as shown in the patch of Listing 3.2) to be inserted right before line 8 in Listing 3.1. DEFECTS4J includes a test that triggers this fault in `abbreviate`, to avoid reintroducing the same mistake in future revisions of the code.

After running for about 70 minutes, JAID produces a number of fix suggestions for the fault, including the fix in Listing 3.3; this fix is equivalent (nearly identical) to the programmer-written fix, and thus completely removes the source of error by handling the special case correctly. To generate fixes for the buggy method `abbreviate`, JAID only needs the source code of the faulty implementation, as well as the programmer-written tests that exercise the method. DEFECTS4J actually includes only one test—the test triggering the fault—for this bug; JAID can produce a correct fix even with such limited information. And Section 3.2 illustrates how JAID produces the correct fix in detail.

To our knowledge, JAID is the first APR tool that can correctly repair the fault of `abbreviate` (as of 2017 May). No other existing tools even provided so-called test-suite adequate fixes, which spuriously pass all available tests avoiding the failure, but do not correctly fix the behavior in the same way that the developers did. Key to JAID's success is its capability of constructing rich state-based abstractions of a program's behavior, which improves the accuracy of fault localization and guides the creation of state-modifying fixes in response to failing conditions.

3.2 How JAID Works

3.2.1 Overview

JAID follows the popular "generate-then-validate" approach, which first generates a number of candidate fixes, and then validates them using the available test cases;

Figure 3.1 gives an overview of the overall process. Inputs to JAID are a Java program, consisting of a collection of classes, and test cases that exercise the program and expose some failures. One key feature of JAID is how it abstracts and monitors program state (snapshots) in terms of program expressions. All stages of JAID's workflow rely on the (snapshots) abstraction derived as described in Section 3.2.2. Fault localization (Section 3.2.3) identifies states and locations (snapshots) that are suspect of being implicated in the failure under repair. Fix generation (Section 3.2.4 and Section 3.2.5) builds code snippets that avoid reaching such suspicious states and locations by modifying the program state, the control flow, or by other simple heuristics. Generated fixes are validated against the available tests (Section 3.2.6); the fixes that pass validation are presented to the user, heuristically ranked according to how likely they are correct (Section 3.2.7).

Let $\boldsymbol{P}$ be the faulty program under fixing, the buggy method to be fix as `m2f`, `C2f` be the containing class of `m2f`. And let $\boldsymbol{T} = \boldsymbol{T}_{✓} \cup \boldsymbol{T}_{✗}$ be the set of test cases for $\boldsymbol{P}$, where $\boldsymbol{T}_{✓}$ and $\boldsymbol{T}_{✗}$ are the sets of passing and failing test cases, respectively, and $\boldsymbol{T}_{✗} \neq \emptyset$. Similar to other program repair techniques like HDA [34], JAID assumes that the fault can be corrected by changing a single method `m2f` from $\boldsymbol{P}$, and that the method is known *a priori*. We view the localization of method `m2f` within $\boldsymbol{P}$ as an orthogonal research problem, to tackle which promising progress has been made in recent years [31, 4, 37, 94].

3.2.2 Program State Abstraction

JAID bases its program analysis and fix generation processes on a detailed *state-based abstraction* of the behavior of the buggy method `m2f`. For every location ℓ in `m2f`, uniquely identifying a statement in the source code, JAID records the values of a set $\boldsymbol{M}_\ell$ of expressions during each test execution: 1) the exact value of expressions of numeric and Boolean types; 2) the object identifier (or `null`) of expressions of

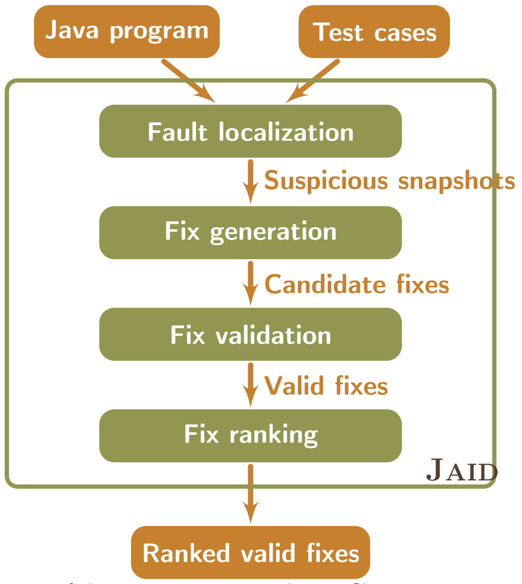


Figure 3.1: An overview of how JAID works. Given a Java program and a set of test cases, including at least one failing test, JAID identifies a number of suspicious snapshots, each indicating a location and an abstraction of the program state at the location that may be implicated in the failure. Based on the snapshot information, JAID generates a number of candidate fixes, which undergo validation against all available tests for the method under repair. Fixes that pass all available tests are considered valid. JAID finally heuristically ranks the valid fixes, and presents the valid fixes to the user in ranking order.

reference types, so that it can detect when a reference is aliased, or is `null`. JAID selects the expressions in $\boldsymbol{M}_\ell$ as follows.

Expressions. A type is *monitorable* if it is a reference type or a primitive type (numeric types such as `int`, and `boolean`). $\boldsymbol{E}_\ell$ denotes the set of all *basic* expressions of monitorable types at ℓ, namely:

- `m2f`'s parameters $\boldsymbol{R}$;
- local variables $\boldsymbol{V}$ declared inside `m2f` that are visible at ℓ;
- fields of class `C2f` $\boldsymbol{F}$ that are visible at ℓ;
- all expressions anywhere inside `m2f` $\boldsymbol{A}$ that can be evaluated at ℓ (that is, they only involve items visible at ℓ), and that don't obviously have side effects (namely, we exclude assignments used as expressions, self increment and

decrement expressions, and creation expressions using `new`).

$\boldsymbol{X}_\ell$ denotes the set of all *extended* expressions of monitorable types at ℓ: for each basic expression of reference type $r \in \boldsymbol{E}_\ell$, $\boldsymbol{X}_\ell$ includes:

- r`.f()` for every argumentless function `f` of the class corresponding to r's type that returns a monitorable type and is callable at ℓ;
- only if r is `this`, r`.a`, for every attribute `a` of the class corresponding to r's type that is readable at ℓ.

For example, the extended expressions $\boldsymbol{X}_8$ at line 8 in method `abbreviate` of Listing 3.1 include `lower` (an argument of `abbreviate`), `str.length()` (a call of function `length()` on `abbreviate`'s argument `str`), and `upper < lower` and `str == null` (both appearing in `abbreviate`).

Purity analysis. One lesson that we can draw from the experience of contract-based APR [61] is that constructing a rich set of expressions that abstract the program state can help support more accurate fault localization and fix generation. Ultimately, the construction of higher-quality "semantic" fixes are less prone to overfitting. However, monitoring a rich set of expressions extracted from the program text does not work as well in languages such as Java as it does in languages that support contracts. In the latter, programmers specifically equip classes with public query methods that are *pure*, which are functions that return value without changing the state of their target objects. Pure methods can be used in the contracts to characterize the program state in response to method calls; hence, these methods are thus easily identifiable and natural candidates to construct state abstractions reliably. In Java, in contrast, programmers need not follow such a discipline of separating pure functions from state-changing procedures, and methods that return a value but have side effects are indeed common. Clearly, a function that is not pure is unsuitable for abstracting and monitoring an object's state.

To identify which expressions can reliably be used for state monitoring, JAID performs a dynamic purity analysis on all expressions that include method invocations. Given an expression r of reference type, the set W_r of r's *watch expressions* consists of:

- all subexpressions S_r of r that do not include method invocations;
- for each subexpression $s \in S_r$, `s.a` for every attribute `a` of the class corresponding to s's type.

Note that watch expressions are constructed so that they are syntactically free from side effects.

An expression r of reference type is then considered *pure* if evaluating it does not alter the value of its watch expressions. Precisely, at every location ℓ in the method `m2f` under repair:

1. first, JAID records the value $\sigma = \langle \sigma_1, \dots, \sigma_m \rangle$ of all watch expressions, where σ_k is the value of $w_k \in W_r$, for $1 \leqslant k \leqslant m$, before evaluating r;
2. then, JAID evaluates r;
3. finally, JAID records again the value $\sigma' = \langle \sigma'_1, \dots, \sigma'_m \rangle$ of all watch expressions, where σ'_k is the value of $w_k \in W_r$, for $1 \leqslant k \leqslant m$, after evaluating r.

If $\sigma = \sigma'$ at every ℓ in every test exercising `m2f`, we call r *pure.*

JAID collects $\boldsymbol{E}_\ell$ and all extended expressions in $\boldsymbol{X}_\ell$ that are pure according to pure analysis as expression-to-monitor $\boldsymbol{M}_\ell$.

State monitoring. This analysis monitors each m in $\boldsymbol{M}_\ell$ with Java Debug Interface (JDI) during test executions. JAID primarily uses JDI's *programmatic* API, which supports retrieving object states and evaluating expressions on them directly without going through a string representation. While for cases that cannot be easily handled through the programmatic interface, JAID relied on JDI's *string-based* expression evaluation methods.

3.2.3 Fault Localization

The goal of fault localization is to identify *suspicious snapshots* indicating locations and states that are likely to be implicated with a fault. A snapshot is a triple $\langle \ell, e, v \rangle$, where ℓ is a location in method `m2f` under repair, e is a Boolean expression, and v is the value (`true` or `false`) of e at ℓ.

Boolean abstractions. The set E_S includes all Boolean expressions that may appear in a snapshot at ℓ; it is constructed by combining the monitored expressions $\boldsymbol{M}_\ell$ to create Boolean expressions as follows:

- for each pair $m_1, m_2 \in \boldsymbol{M}_\ell$ of expressions of the same type, E_S includes m_1`==`m_2 and m_1 `!=` m_2;
- for each pair $k_1, k_2 \in \boldsymbol{M}_\ell$ of expressions of integer type, E_S includes $k_1 \bowtie k_2$, for $\bowtie \in \{$`<`, `<=`, `>=`, `>`$\}$;
- for each expression $b \in \boldsymbol{M}_\ell$ of Boolean type, E_S includes b and `!`b;
- for each pair $b_1, b_2 \in \boldsymbol{M}_\ell$ of expressions of Boolean type, E_S includes b_1 `&&` b_2 and $b_1 \mid\mid b_2$.

Continuing the example of method `abbreviate` in Listing 3.1, M_8 includes expressions such as `lower >= str.length()` and `!(str == null)`.

Snapshot suspiciousness. JAID computes the *suspiciousness* of every snapshot $s = \langle \ell, e, v \rangle$ based on Wong at al.'s fault localization techniques [84]. The basic idea is that the suspiciousness of s combines two sources of information:

- a syntactic analysis of expression dependence, which gives a higher value ed_s to s the more subexpressions e shares with those used in the statements immediately before and immediately after ℓ (this estimates how much s is relevant to capture the state change at ℓ);
- a dynamic analysis, which gives a higher value dy_s to s the more often b evaluates to v at ℓ in a failing test $t_{✖}$; and a lower value to s the more often b

evaluates to v at ℓ in a passing test $t_{\checkmark}$ (this collects the evidence that comes from monitoring the program during passing and failing tests).

The overall suspiciousness $2/(ed_s^{-1} + dy_s^{-1})$ is the harmonic mean of these two sources, but the dynamic analysis has the biggest impact—because ed_s is set up to be a value between zero and one, whereas dy_s is at least one and grows with the number of passing tests .

This approach is similar to AutoFix's [61, Sec. 4.2]—which is also based on [84]—but conspicuously excludes information about the distance between ℓ and the location of failure on the control flow graph of the faulty method. AutoFix identifies failures as contract violations, which tend to be happen closer to where the program state becomes corrupted; by contrast, in JAID's setting—using tests without contracts in Java—failures normally happen when evaluating an assert statement inside a test method, and thus the distance to the location of failure within the faulty method is immaterial, and hardly a reliable indication of suspiciousness.

In the running example of method `abbreviate` in Listing 3.1, the snapshot ⟨8, `lower >= str.length()`, `true`⟩ receives a high suspiciousness score because `lower` and `str.length()` appear prominently in the statements around line 8, and, most important, `lower >= str.length()` holds in all failing and in no passing tests.

3.2.4 Fix Generation: Fix Actions

A snapshot $s = \langle \ell, e, v \rangle$ with high suspiciousness indicates that the program is prone to trigger a failure when the *program state* in some execution is such that e evaluates to v at ℓ. Correspondingly, JAID builds a number of candidate fixes that try to *steer away* from the suspicious state in the hope of avoiding the failure. To this effect, JAID enumerates four kinds of *fix actions*: i). modify the state directly by assignment; ii). affect the state that is used in an expression; iii). mutate a statement; iv). redirect the control flow. Each fix action is a (possibly compound) statement that can replace the statement at ℓ. Actions of kinds i and ii are semantic—they directly target the program state; actions of kind iii are syntactic—they tinker with existing code expressions according to simple heuristics; actions of kind iv are the simplest—they are independent of the snapshot's information. We outline how JAID builds fix actions in the following paragraphs, based on a definition of derived expressions. Section 3.3 discusses which fix actions were the most effective in the experimental evaluation.

Derived expression. Given an expression *exp*, $\Delta_{\ell,exp}$ denotes all derived expressions built from *exp* as follows:

- if *exp* has numeric type, $\Delta_{\ell,exp}$ includes *exp*;
- if *exp* has Boolean type, $\Delta_{\ell,exp}$ includes *exp* and `!`*exp*;
- $\Delta_{\ell,exp}$ also includes t and t`.f(`$\cdots$`)`, for every $t \in \boldsymbol{M}_\ell$ of reference type, where `f` is a function of the class t belongs to. The function is possibly called with actual arguments chosen from the monitored expressions $\boldsymbol{M}_\ell$ of suitable type.

Its *top-level subexpressions* $\mathcal{S}_{exp}$ are the expressions corresponding to the nodes at depth 1 in *exp*'s abstract syntax tree (i.e., the root's immediate children). For example, the top-level subexpressions of `(a + b) < c.d()` are `a + b` and `c.d()`. Then, $\Delta'_{\ell,exp} = \bigcup_{s \in \mathcal{S}_{exp}} \Delta_{\ell,exp}$ denotes all expressions derived from *exp*'s top-level subexpressions.

Modifying the state. For every top-level subexpression *exp* of e, if *exp* is assignable to, JAID generates the fix action exp `=` δ for each $\delta \in \Delta'_{\ell,}$ whose type is compatible with *exp*'s.

In the running example of method `abbreviate` in Listing 3.1, JAID includes the assignment `lower = str.length()` among the fix actions that modify the state at line 8.

Modifying an expression. For every top-level subexpression *exp* of e that is not assignable to, but appears in the statement S at ℓ, JAID generates the fix action `tmp_e = `δ`;` $S[exp \mapsto$ `tmp_e`$]$ for each $\delta \in \Delta'_{\ell,e}$ whose type is compatible with *exp*'s; `tpm_e` is a fresh variable with the same type as *exp*, and $S[exp \mapsto$ `tmp_e`$]$ is the statement at ℓ with every occurrence of *exp* replaced by `tmp_e`—which has just been assigned a modified value.

Mutating a statement. "Semantic" fix actions, that are based on the information captured by the state in suspicious snapshots, are usefully complemented by a few "syntactic" fix actions, which are based on simple mutation operators that capture common sources of programming mistakes such as off-by-one errors. Following an approach adopted by other APR techniques [91, 34], JAID generates mutations mainly targeting *conditional expressions*. Precisely, if the statement S at ℓ is a conditional or a loop, JAID generates fix actions for every *Boolean* subexpression *exp* of e that appears in the conditional's condition or in the loop's exit condition:

1. if *exp* is a comparison $x_1 \bowtie x_2$, for $\bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}$, JAID generates the fix action $S[exp \mapsto (x_1 \bowtie' x_2)]$, for every comparison operator $\bowtie' \neq \bowtie$;
2. JAID also generates the fix actions $S[exp \mapsto \texttt{true}]$ and $S[exp \mapsto \texttt{false}]$, where *exp* is replaced by a Boolean constant.

In addition to targeting Boolean expressions, if the statement S at ℓ includes a *method call* `t.m(`$\texttt{a}_1, \ldots, \texttt{a}_n$`)`, JAID generates the fix action $S[\texttt{m} \mapsto \texttt{x}]$, which calls any applicable method `x` on the same target and with the same actual arguments as `m` in s.

Modifying the control flow. Even though fix actions may indirectly change the control flow by modifying the state or a branching condition, a number of bugs require abruptly redirecting the control flow. To achieve this, JAID also generates the following fix actions independent of the snapshot information:

1. if method `m2f` is a procedure (its return type is `void`), JAID generates the fix

action `return`;

2. if method `m2f` is a function, JAID generates the fix action `return` *exp*, for every basic expression of suitable type available at ℓ;
3. if ℓ is a location inside a loop's body, JAID generates the fix action `continue` and `break`.

3.2.5 Fix Generation: Candidate Fixes

Each fix action (built by JAID as described in the previous section) is a statement that modifies the program behavior at location ℓ in a way that avoids the state implicated by some suspicious snapshot $s = \langle \ell, e, v \rangle$. In most cases, a fix action should not be injected into the program under repair *unconditionally*, but only when state e is actually reached during a computation. A conditional execution would leave program behavior unchanged in most cases, and only address the failing behavior when it is about to happen.

To implement such conditional change of behavior, JAID uses the *schemas* in Figure 3.2 to insert fix actions into the method `m2f` under repair at location ℓ. First, JAID instantiates every applicable schema with each fix action `action`; in addition to the fix action, schemas include the statement `oldStatement` at location ℓ in the faulty `m2f`, and the condition `suspicious`, which is e `==` v as determined by the snapshot's abstract state. Then, JAID builds *fix candidates* by *replacing* the statement at ℓ in `m2f` by each instantiated schema.[1]

Continuing the running example of method `abbreviate` in Listing 3.1, one of the fix candidates consists of the fix action `lower = str.length()` instantiating schema B: the action is executed only if `lower >= str.length()` (from the suspicious snapshot), whereas the existing statement at line 8, as well as the rest of

[1]Since each fix generated by JAID combines one fix action and one schema, it adds at most 5 new lines of codes to a patched method.

```
Schema A: action; oldStatement;
Schema B: if (suspicious) { action; } oldStatement;
Schema C: if (suspicious) { oldStatement; }
Schema D: if (suspicious) { action; }
          else { oldStatement; }
Schema E: /* oldStatement; */ action;
```

Figure 3.2: Schemas used by JAID to build candidate fixes

method `m2f`, is unchanged by the fix.

Two of the five schemas currently used by JAID to build fix candidates inject the fix action unconditionally. On the other hand, different fix actions may determine semantically equivalent fixes when instantiated. JAID performs a lightweight redundancy elimination, based on simple syntactic rules such as that `x == y` is equivalent to `!(x != y)`.

3.2.6 Fix Validation

Even if JAID builds candidate fixes based on a semantic analysis of the program state during passing vs. failing tests, the candidate fixes come with no guarantee of satisfying the tests. To ascertain which candidates are suitable, a fix validation process, which follows fix generation, runs all tests $\boldsymbol{T}$ that exercise the faulty method `m2f` against each generated candidate fix. Candidate fixes that pass all tests $\boldsymbol{T}$ are classified as *valid* (also "test-suite adequate" [49]) and retained; other candidates, which fail some tests, are discarded—as they do not fix the fault, they introduce a regression, or both.

In the example of method `abbreviate` in Listing 3.1, the fix candidate `if (lower >= str.length) lower = str.length()` passes validation, since it fixes the fault and introduces no regression error.

Since JAID commonly generates a large number of candidate fixes for each fault, validation can take up a very large time spent compiling and executing tests, which

```
// class being repaired
class C2f {
  U m2f_(T₁ a₁, T₂ a₂, ...) throws IllegalStateException {
    switch (Session.getActiveFixId()) {
      case 0: return m2f(a₁, a₂, ...);    // call faulty method
      case 1: return m2f_1(a₁, a₂, ...);  // call fix candidate 1
      ⋮
      case n: return m2f_n(a₁, a₂, ...);  // call fix candidate n
      default: throw new IllegalStateException();
    }
  }
}
```

Listing 3.4: How multiple fix candidates are woven into a single class.

may ultimately impair the scalability of JAID's APR. To curtail the time spent compiling, JAID deploys a simple form of dependency injection. All candidate fixes for a method `m2f` become members of `m2f`'s enclosing class `C2f`: candidate fix number `k` becomes a method `m2f_k` with signature the same as `m2f`'s. Then, as shown in Listing 3.4, a method `m2f_`—also with the same signature—dispatches calls to any of the candidate fixes based on the value returned by static method `getActiveFixId()` of class `Session`, which supplies the dependency. This scheme only requires one compilation per method under repair, thus significantly cutting down validation time.

3.2.7 Fix Ranking

Like most APR techniques, JAID's process is based on heuristics and driven by a finite collection of tests, and thus is ultimately *best effort*: a valid fix may still be incorrect, passing all available tests only because the tests are incomplete pieces of specification. JAID addresses this problem by *ranking* valid fixes using the same heuristics that underlies fault localization. Every fix includes one fix action, which was derived from a snapshot s; the higher the suspiciousness of s, the higher the fix is ranked; fixes derived from the same snapshot are ranked in order of generation, which means that "semantic" fixes (modifying state or expressions) appear before "syntactic" fixes (mutating statements or modifying the control flow), and fixes of the same kind are enumerated starting from the syntactically simpler ones.

When the ranking heuristics works, the user only inspects few top-ranked fixes to assess their correctness and whether they can be deployed into the codebase. The experimental evaluation in Section 3.3 comments on the effectiveness of JAID's ranking heuristics.

3.3 Experimental Evaluation

To assess JAID's capabilities and explore further directions, we conduct a comprehensive study on 693 bugs from three different benchmark suites. These faults come from Java applications and libraries of different sizes and complexity in disparate domains; this makes the results of JAID's evaluation more likely to be representative of its general behavior.

In this experiment, we first conduct a detailed assessment of JAID's effectiveness and efficiency from a user's perspective. Precisely, the effectiveness is evaluated in terms of the number of faults that JAID can fix, and the efficiency is measured by the running time of JAID on different bugs. To clarify the advantage and short-coming of our approach among all APR techniques, we also quantitatively compare JAID and all automated program repair tools for Java that are available at the time. This comparison also pinpoints the superiority of different approaches, which may benefit our further development.

Since templates and heuristics construct the major part of the JAID's fixing process, we then analyze about how all templates and heuristics impact on JAID's effectiveness. This analysis zoom in on the relative usefulness of various kinds of templates and heuristics used by JAID's APR algorithm.

Moreover, because accurately identifying program locations that are most closely implicated with a fault greatly affects the effectiveness of the repair process, fault localization is a crucial step of automated program repair. We also study how sensitive is JAID's behavior to the choice of spectrum-based fault localization algorithm. Precisely, we replace JAID's custom fault localization algorithm with other widely used algorithms and monitor how it impacts on JAID's overall effectiveness.

The evaluation shows, among other things, that:

- JAID produced correct fixes for over 15% of the bugs from different benchmark suites;
- JAID could correctly fix more bugs than any other APR tool for Java, including 11 bugs that no other tool can fix (as of May 2019);
- JAID's heuristics are effective and fairly robust (that is, JAID's behavior does not depend on fine-tuning its parameters);
- JAID's effectiveness and efficiency are largely independent of the spectrum-based fault localization algorithm that is used.

These results indicate that JAID is a competitive tool, achieving a combination of effectiveness, efficiency, and applicability that often compares favorably to the state-of-the-art. They also outline directions to further improve the overall performance of JAID's algorithm: improve the precision of fault localization and fix ranking, and reduce analysis times. Particularly, our experiments suggest that JAID's overall effectiveness does not depend much on the details of its spectrum-based fault localization algorithm, which indicates that further substantial progress would probably require to step up the precision of fault localization in a way that it can incorporate such additional sources of information.

3.3.1 Experimental Design

We experimentally evaluated Jaid on 693 faults from three curated collections often used in automated program repair research. These faults come from Java applications and libraries of different size and complexity in disparate domains; this makes the results of Jaid's evaluation more likely to be representative of its general behavior. At the same time, using faults from standard benchmarks makes it possible to meaningfully compare Jaid with the other state-of-the-art APR tools for Java.

Research Questions

The experiments address the following research questions:

RQ1: How *effective* is Jaid? In RQ1, we evaluate the effectiveness of Jaid from a user's perspective—in terms of the number of faults that Jaid can fix.

RQ2: How *efficient* is Jaid? In RQ2, we examine the running time of Jaid on different bugs.

RQ3: How effective is Jaid in comparison to *other APR techniques* for Java? In RQ3, we directly compare Jaid to several other state-of-the-art APR tools for Java.

RQ4: Do all *templates* and *heuristics* have an *impact* on Jaid's effectiveness? In RQ4, we zoom in on the relative usefulness of various kinds of templates and heuristics used by Jaid's APR algorithm.

RQ5: How sensitive is Jaid's behavior to the choice of *fault localization* algorithm? In RQ5, we assess whether replacing Jaid's custom fault localization algorithm with a different one affects its overall effectiveness.

Table 3.1: DEFECTS4J benchmark: how many BUGS, TESTS, and thousands of lines of code (KLOC) DEFECTS4J includes from each PROJECT.

PROJECT	DESCRIPTION	KLOC	TESTS	BUGS
Chart	JFreechart	96	2205	26
Closure	Closure Compiler	90	7927	133
Lang	Apache Commons-Lang	22	2245	65
Math	Apache Commons-Math	85	3602	106
Time	Joda-Time	27	4130	27
Mockito	Mockito	43	1161	38
	TOTAL	320	20109	395

Table 3.2: INTROCLASSJAVA benchmark: for each PROGRAM, how many **b**lack-box and **w**hitebox TESTS exercise it, and how many of its BUGS are triggered by the **b**lackbox and the **w**hite-box tests.

			TESTS		BUGS	
PROGRAM	DESCRIPTION	LOC	**b**	**w**	**b**	**w**
checksum	checksum of a string	18	6	10	7	11
digits	digits of a number	13	6	10	51	60
grade	grade from score	22	9	9	89	88
median	median of 3 numbers	24	7	6	51	48
smallest	min of 4 numbers	24	8	8	47	45
syllables	count vowels	17	6	10	13	12
	TOTAL	118	42	53	258	264
	detectable unique bugs in $\mathbf{b} \cup \mathbf{w}$:				297	

Table 3.3: QUIXBUGS benchmark: the MIN, MEDIAN, MAX, and TOTAL *size* of the faulty methods (in lines of code), and number of *tests* targeting each method.

	MIN	MEDIAN	MAX	TOTAL
Size (LOC)	2	17	45	717
Tests	3	6	13	259

Subjects

Our evaluation uses 693 faults from three widely used benchmarks of Java bugs: Defects4J [27], IntroClassJava [16], and QuixBugs [39].

Defects4J (revision `#895c4e6`) includes 395 bugs from 6 projects: Chart, Closure, Lang, Math, Time, and Mockito; Table 3.1 displays basic statistics about these projects in Defects4J. The bugs included in Defects4J are a diverse sample of real-world bugs, and as such they include both several that admit simple fixes and others that require changes that span multiple methods or even multiple files. Each bug in Defects4J has a unique identifier, corresponds to a buggy version and a programmer-fixed version of the code, and is accompanied by some programmer-written unit tests that exercise the code. In particular, at least one of the given tests triggers the failure on the buggy version.

Our experiments used *all* the 395 bugs in Defects4J as our subjects. Because using all available bugs in a benchmark provides the most comprehensive data about Jaid's performance, and hence makes for a fair comparison with other tools evaluated on the same benchmark. For example, measures such as average running time should uniformly include the time to run a tool on all bugs, including those where the tool will fail to produce any fixes. Including all bugs is also the recommendation of all large-scale replication studies [49, 15, 40]. A more specific reason for not pre-selecting bugs is that sometimes there is no sure way to know in advance on which bugs Jaid will be successful. For example, Section 3.3.2 examines five "hard" bugs that Jaid unexpectedly managed to fix with fixes that were semantically equivalent to syntactically much more complex fixes written by developers.

IntroClassJava is a direct Java translation [16] of the IntroClass benchmark [36], which collects several buggy student-written solutions to six small assignments given in an introductory, undergraduate programming course. As such, programs in Intro-

ClassJava are a meaningful sample of the kinds of mistakes commonly made by novice programmers writing small programs. Table 3.2 displays basic statistics about the 297 buggy programs translated to Java in IntroClassJava.

Each program in IntroClassJava comes with two JUnit test suites (also translated from C): a black-box one written by the instructor based on the program's specification, and a white-box one generated using a symbolic execution tool. Following the practice of other studies [36, 32, 83], Jaid had access only to the black-box tests; the white-box tests feature only in the experimental evaluation, where we used them to determine which of the fixes outputted by Jaid were correct. Since 39 buggy programs in IntroClassJava do not cause any black-box tests to fail (that is, only white-box tests trigger failures), we excluded them from the experiments and used the remaining 258 program variants as subjects in the experiments.

QuixBugs contains 40 faulty programs taken from the Quixey Challenge [39], where programmers had one minute to produce a fix given an implementation of a classic algorithm with a bug on a single line. The algorithms include classic algorithm such as Dijkstra's shortest path on a graph, building the minimum spanning tree, and sorting algorithms; therefore, the buggy programs in QuixBugs are representative of programming mistakes that are commonly made when implementing algorithms that are challenging to get right. Table 3.3 displays basic statistics about the programs in QuixBugs.

Each faulty program in QuixBugs comes with a correct reference implementation, as well as passing and failing tests; thus, the textual diff between a buggy program and the corresponding reference implementation plays the role of the programmer-written fix for the bug. Our experiments used all the 40 bugs in QuixBugs, together with all available tests.

Other Automated Program Repair Tools for Java

We quantitatively compared JAID to 16 tools(described in Chapter 2) for APR of Java programs; we considered all tools working on Java that used at least one of the three benchmarks (DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS) in their published experimental evaluation. Table 3.4 lists the 16 tools[2]; if an experimental evaluation of tool T on benchmark B is publicly available, the table indicates, at row T and column B, the source of the experimental results that we used in the comparison with JAID.

While most tools use DEFECTS4J as benchmark in their experiments, it is possible that the experimental evaluations of different tools select different *subsets* of bugs in DEFECTS4J. This may happen for two reasons: first, since the DEFECTS4J benchmark is occasionally extended with new projects, it may happen that an experimental evaluation on an older version of DEFECTS4J simply did not have access to some bugs that are available in DEFECTS4J at the time of writing. Second, an experimental evaluation may deliberately exclude some bugs from the experiments if the technique being evaluated or its implementation are not easily applicable to those bugs. For example, [49] excludes project Closure because of difficulties in executing its tests.

Despite these , our quantitative comparison with other tools is sound:

- Since bugs that are excluded *a priori* from an evaluation can normally be considered beyond a tool's current capabilities, we compute *recall* relative to the same largest set of DEFECTS4J bugs, which is a superset of all those used in any evaluations. In contrast, excluding bugs from an evaluation does not negatively affect a tool's *precision*—in fact, it puts JAID at a disadvantage because we insisted on running it on every available bug in DEFECTS4J.

[2]SketchFixPP is a variant of the SketchFix technique discussed in the same paper [23].

- We report absolute numbers of valid and correct fixes, as well as the bugs that each tool *uniquely* fixes.
- We ascertain that the version of DEFECTS4J used in our experiments does not differ substantially, on the same bugs, from those used in the other tools' experiments.

We did not perform any quantitative comparison of other measures, running time in particular, as these would require to replicate experiments with other tools with common settings—which is not always possible or easy, since not all publications come with a complete replication package. Since improving performance is not a primary concern in automated program repair research at the moment, and publications rarely emphasize performance data, the comparison of JAID with other tools focuses on metrics that are generally accepted as practically interesting.

Fault-localization techniques

JAID employs a custom spectrum-based [86] fault localization technique, while most other APR tools directly reuse standard fault localization approaches. To find out whether JAID's behavior depends significantly on how fault localization works (and hence to answer RQ5), we ran experiments where JAID uses other well-known spectrum-based fault localization techniques instead of its custom algorithm.

Table 3.5 lists the five spectrum-based techniques [60] used to replace JAID's. All of them work by computing a *suspiciousness* score of each program entity (in the case of JAID, *snapshots* which include a location and part of the state—as described in Section 3.2.3) based on its coverage by passing and failing tests.

Table 3.4: Automated program repair tools for Java used in the comparison with JAID. For each benchmark, the table reports the source of a tool's evaluation results used for a quantitative comparison with JAID; no reference means the tool has not been evaluated on the benchmark.

TOOL	DEFECTS4J	INTROCLASSJAVA	QUIXBUGS
ACS	[89]		
Astor			[50]
CapGen	[83]	[83]	
Elixir	[70]		
HAD	[34]		
JFix		[32]	
jGenProg	[49]		
jKali	[49]		
Nopol	[49]		[92]
S3		[33]	
SimFix	[25]		
SketchFix	[23]		
SketchFixPP	[23]		
ssFix	[88]		
xPar	[34, 89]		

Table 3.5: Spectrum-based fault localization ALGORITHMs used to replace JAID's. Each algorithm defines a formula to compute the SUSPICIOUSNESS of any state snapshot s as a function of $\mathsf{P}(s)$ (the number of passing tests where s is observed) and $\mathsf{F}(s)$ (the number of failing tests where s is observed), and relative to the total number of passing (P) and failing (F) tests.

ALGORITHM	SUSPICIOUSNESS OF SNAPSHOT s
Tarantula [26]	$\frac{\mathsf{F}(s)/\mathsf{F}}{\mathsf{F}(s)/\mathsf{F} + \mathsf{P}(s)/\mathsf{P}}$
Ochiai [2]	$\frac{\mathsf{F}(s)}{\sqrt{\mathsf{F}\cdot(\mathsf{F}(s)+\mathsf{P}(s))}}$
Op2 [57]	$\mathsf{F}(s) - \frac{\mathsf{P}(s)}{\mathsf{P}+1}$
Barinel [1]	$1 - \frac{\mathsf{P}(s)}{\mathsf{P}(s) + \mathsf{F}(s)}$
DStar [85] (with $* = 2$)	$\frac{\mathsf{F}(s)^*}{\mathsf{P}(s) + \mathsf{F} - \mathsf{F}(s)}$

Experimental Setup

All the experiments ran on a cloud infrastructure, with each run of JAID using exclusively one virtual machine instance, configured to use one core of an Intel Xeon Processor E5-2630 v2, 8 GB of RAM, Ubuntu 14.04, and Oracle's Java JDK 1.8.

Each experiment targets one subject bug k, and runs JAID with buggy code bug_k and tests T_k as input; the output is a ranked list of valid fixes for the bug. The process to determine which of JAID's fixes are *correct* follows standard research practices:

- The fix of a bug in DEFECTS4J or QUIXBUGS is correct if manual inspection convincingly indicates that it is *semantically equivalent* to the programmer-written fix $\mathsf{correct}_k$. We allot around 5 minutes per fix to determine semantic equivalence; if we cannot conclude that the fix is equivalent after 5 minutes, we classify it as incorrect. This conservative assessment implies that a fix classified as correct is a fix suggestion that could have been deployed (or is very close to one that could have been deployed).

- The fix of a bug in INTROCLASSJAVA is correct if it passes all available white-box tests (which JAID had not access to), in addition to the black-box tests that JAID uses directly for validation. Programs and bugs in INTROCLASSJAVA are sufficiently simple that passing all white-box tests provides high confidence in their correctness.

We did not set a *time limit* in our experiments: since JAID ranks all generated snapshots according to their suspiciousness (see Section 3.2.3), and then depends on the ranking to guide the following stages, setting an arbitrary cutoff time may prevent JAID from generating a complete ranking. Instead, we limited the search space in our experiments by configuring JAID so that it uses at most 1500 snapshots in order of suspiciousness; then, the following stages (Figure 3.1) all run to completion.

The number 1500 was chosen heuristically; some of the experiments reported in Section 3.3.2 indicate that this choice achieves a reasonable trade-off, but also that JAID's overall performance is fairly robust with respect to the choice of how many snapshot to process.

3.3.2 Experimental Results

This section presents the results of the experimental evaluation of JAID carried out according to the design of Section 3.3.1. Averages are measured using the *median* by default, with exceptions explicitly pointed out.

RQ1: Effectiveness

Table 3.6 displays the key results of the experimental evaluation of JAID's effectiveness. As shown there, JAID produced *valid* fixes for 189 bugs in total: 94 bugs in DEFECTS4J, 84 bugs in INTROCLASSJAVA, and 11 bugs in QUIXBUGS. More significantly, it produced *correct* fixes for 113 bugs: 35 in DEFECTS4J, 69 in INTROCLASSJAVA, and 9 in QUIXBUGS. These numbers correspond to an overall:

precision: 59.5% the percentage of bugs with a valid fix for which JAID outputs at least a correct fix

recall: 15.4% the percentage of all bugs for which JAID outputs at least a correct fix

As we discuss in Section 3.3.2, JAID's effectiveness is competitive with the state of the art; JAID is capable of producing fixes of high quality in a significant number of cases on code of various complexity and maturity level.

Precision and recall are much higher with benchmarks INTROCLASSJAVA and QUIXBUGS than with DEFECTS4J. This difference is likely the result of two aspects. First, DEFECTS4J programs are often larger and require more complex fixes than

Table 3.6: Effectiveness of JAID: main experimental results for each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS, as well as OVERALL (bottom row in each table).

(a) Number of BUGS for which JAID produced VALID FIXES; number of ALL valid fixes produced for all bugs; and MEDIAN number of valid fixes produced per bug for which at least a valid fix was produced.

	VALID FIXES		
BENCHMARK	BUGS	ALL	MEDIAN
DEFECTS4J	94	16762	23.5
INTROCLASSJAVA	84	4243	28.5
QUIXBUGS	11	2187	10.0
OVERALL	189	23192	22.0

(b) Number of bugs for which JAID produced CORRECT FIXES in ANY position, FIRST position, a TOP-10 position; median POSITION of the first correct fix in JAID's output; and number of ALL correct fixes produced for all bugs; and the MEDIAN number of correct fixes produced per bug for which at least a correct fix was produced.

	CORRECT FIXES					
BENCHMARK	ANY	FIRST	TOP-10	POSITION	ALL	MEDIAN
DEFECTS4J	35	14	25	3	139	2
INTROCLASSJAVA	69	30	43	2	829	5
QUIXBUGS	9	4	9	2	22	2
OVERALL	113	48	77	2	990	3

(c) Precision and recall obtained by JAID. PRECISION is the ratio $100 \cdot$ ANY/BUGS denoting the percentage of bugs with a valid fix that is also correct; and RECALL is the ratio $100 \cdot$ ANY/TOTAL denoting the percentage of TOTAL bugs with a correct fix.

BENCHMARK	PRECISION	RECALL
DEFECTS4J	37.2	8.9
INTROCLASSJAVA	82.1	26.7
QUIXBUGS	81.8	22.5
OVERALL	59.5	15.4

Table 3.7: Bugs in DEFECTS4J and QUIXBUGS for which JAID failed to produce any valid fix. The total number of such bugs (ALL) is split according to where JAID's ineffectiveness originates: in an unsuccessful SETUP, in an ineffective FAULT LOCALIZATION, or in a limited FIX SPACE. Some bugs are the combined result of ineffective fault localization and limited fix space, and thus are counted in both columns.

	FAULTS NOT FIXED			
	ALL	SETUP	FAULT LOCALIZATION	FIX SPACE
DEFECTS4J	104	33	41	63
QUIXBUGS	29	3	8	24
TOTAL	133	36	49	87

those in the other two benchmarks. Second, tests in INTROCLASSJAVA and QUIXBUGS are more thorough, and hence provide stronger specifications of the intended program behavior, and support a more effective test-based validation that prunes out valid but incorrect fixes.

Still, JAID can be effective even with weak oracles: it correctly fixed 5 bugs in DEFECTS4J that include only one failing test (and no passing tests), ranking the correct fix first in two cases.

Among the 35 bugs from DEFECTS4J that JAID correctly fixes, 5 are from project Chart, 13 from project Closure, 6 from project Lang, 9 from project Math, 1 from project Time, and 1 from project Mockito. In particular, Mockito bugs were not used to evaluate most other APR tools. A recent study [78] shows that Nopol can correctly fix 1 bug while SimFix and CapGen cannot propose any valid fix to bugs in Mockito even if all the buggy locations are analyzed. Compared with that, JAID generated valid fixes to 4 Mockito bugs (and correct fixes to 1).

When JAID is ineffective. To better understand the limitations of JAID, we manually analyzed the bugs where JAID failed to produce any valid fix. In the analysis, we restrict ourselves to the faults from DEFECTS4J and QUIXBUGS, since only bugs in these two benchmarks are accompanied by programmer-written fixes. We also excluded from this manual analysis faults whose programmer-written fixes:

- modify more than one method (108 bugs excluded);
- modify source code outside the only buggy method (e.g., class fields and/or `import` statements) (13 bugs excluded); or
- modify more than four lines of code, according to the dissection data of Defects4J [74] (99 bugs excluded).

Faults whose fixes have these characteristics are clearly outside the current capabilities of Jaid's algorithm,[3] and hence it is unsurprising that Jaid failed to produce any fix for them.

Examining the remaining 104 bugs that Jaid could not fix, we identified three phases from which Jaid's ineffectiveness commonly originates (see Table 3.7 for a summary of the number of bugs in each category):

Setup: Jaid uses a simple Python script to automatically extract project configuration information from the Defects4J framework. The script works well on most Defects4J bugs, but when it fails to properly setup the environment it prevents Jaid from running at all.

An unsuccessful setup prevented 36 bugs from being fixed; these failures are the result of practical limitations of the current implementation of Jaid, but they do not reflect any fundamental limitation of Jaid's approach.

[3]We could trivially lift some of these limitations, for example by supporting the generation of fixes that combine multiple actions. However, such changes would make the fix space much larger, without also providing a scalable mechanism to explore it efficiently. Thus, effectively lifting these limitations would require modifications of the approach that go beyond simple extensions of the search space.

Fault localization: JAID's repair process uses fault localization as a crucial starting point: when the fault localization algorithm fails to track the relevant failing condition in a snapshot expression, or ranks the corresponding snapshot past the cutoff, the following fix generation step cannot be successful because it does not have the necessary ingredients.

Ineffective fault localization prevented 49 bugs from being fixed. Even though JAID uses only the 1500 most suspicious snapshots for fix generation, the cutoff is not a significant limitation in practice: if we increase it to allow 3000 snapshots, JAID correctly fixes 1 more bug (`Lang39`); if we remove the cutoff entirely, JAID correctly fixes 2 more bugs (`Closure104` and `Lang24`). Thus, the current cutoff value is a reasonable balance between costs and effectiveness; and improving the effectiveness of fault localization in JAID is likely to require more radical changes to the kind of information it uses. We discuss further details about the effectiveness of fault localization in JAID in Section 3.3.2.

Fix space: even with perfect fault localization, some fixes may require program modifications that are inexpressible using JAID's fix actions and schemas, and hence fall outside its fix space.

A limited fix space prevented 87 bugs from being fixed. Examples of features outside JAID's fix space that were used in some human-written fixes include: `new` expressions with type casting; calls to methods with arguments; constants declared in specific classes; adding a `case` to an existing `switch` statement; adding an `if` with a compound then or else block; adding a compound statement (such as a `try/catch` block, or a loop).

While extending JAID so that it generates such complex statements as fix actions is not technically complex, the expanded fix space would become too large to effectively search in a reasonable time using JAID's heuristics. Extending

JAID's search space selectively—focusing on fixing only certain categories of bugs—is an interesting direction for future work; but JAID's current focus is on being "general-purpose".

Ranking of fixes. When it is successful, JAID often produces several valid fixes—122.7 (= 23192/189) per fixed bug on average in our experiments—and a much smaller number of correct fixes—8.8 (= 990/113) per fixed bug on average in our experiments. When JAID outputs several valid fixes for the same bug, it ranks them according to a simple heuristics that tries to put on top those that are more likely to be correct.

The stacked histogram in Figure 3.3 depicts the distribution of the top ranks of correct fixes. Among the 113 bugs that JAID correctly fixed, the median position of the first correct fix was 2 (from the top); the correct fix appeared at the top position in 48 bugs; and among the top-10 valid fixes for 77 bugs. For the remaining 36 bugs that were correctly fixed, the correct fix appears further down in the output list; in 26 of these cases, the correct fix turns out to be a "syntactic" one, but several valid, incorrect "semantic" fixes are generated and ranked higher.[4]

These results indicate the importance of ranking to ensure that the correct fixes are easier to spot among several valid but incorrect ones. JAID's ranking heuristics often do a good job, but there is room for improvement. In particular, ranking may benefit from mining additional information about common features of programmer-written fixes, as was done in the other repair tools like ACS, HDA, SimFix, and CapGen.

Syntactic complexity of fixes. In DEFECTS4J and QUIXBUGS, we classify a fix as correct if it is *semantically* equivalent to the fix written by programmers for the same bug. A potential risk is that JAID generates fixes that are semantically

[4]Section 3.2.4 explains what syntactic and semantics fix actions are. Section 3.3.2 gives detailed statistics on the usage of different types of fix actions in generated correct fixes.

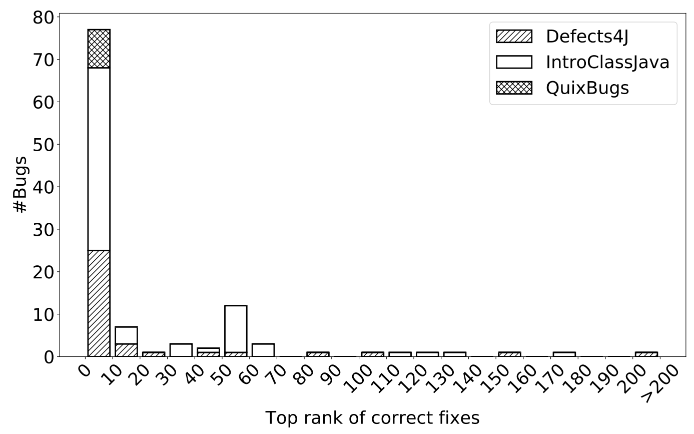


Figure 3.3: Distribution of the top rank of correct fixes of bugs.

equivalent but *syntactically* very different—in particular, needlessly complex and less readable. In practice, this is unlikely to happen because JAID cannot generate fixes that are syntactically very complex; for the simpler fixes it can generate, semantic and syntactic equivalence tend to coincide.

There are a few exceptions to this general observation: the programmer-written fixes for 5 of the 35 DEFECTS4J bugs that JAID can correctly fix involve modifications of multiple lines at one or several locations, or even deletion of a whole method. Thus, JAID's correct fixes for these bugs were syntactically much simpler than, but still semantically equivalent to, the programmer-written fixes. On the one hand, this is encouraging because it proves the flexibility of JAID's repair algorithm. On the other hand, one could argue that the more complex programmer-written fixes are in some cases preferable because they are refactoring aspects of the program's design in order to improve the code beyond the single functionality that is being repaired.

For example, the programmer-written fix of bug Closure 46 involves deleting a 17-line long method `getLeastSupertype` from class `RecordType`. Method `getLeast-Supertype` overrides the method with the same name in superclass `Prototype-ObjectType`, so the fix replaces, implicitly by inheritance, all invocations to the overriding method `RecordType.getLeastSupertype` with calls to the overridden one

`PrototypeObjectType.getLeastSupertype`. Instead of removing method `Record-Type.getLeastSupertype`, JAID fixes the same bug by changing `RecordType.get-LeastSupertype`'s body so that it explicitly calls the overridden method `Prototype-ObjectType.getLeastSupertype` as `super.getLeastSupertype` and just returns the call's result. In this case, the programmer-written fix has a better design since it does not leave dead code in the repaired program, but the fix produced by JAID is still completely behaviorally correct and hence practically useful.

JAID *produced a correct fix for over 15% of all bugs it analyzed, achieving a precision of nearly 60%. It ranked most correct fixes high in the output list (in position 2 on average).*

RQ2: Efficiency

We measure the efficiency of JAID in terms of *running time*: overall running time, and running time until the first fix is generated.

Overall running time. Figure 3.4 shows the distribution of JAID's overall running time in every experiment. The distribution is clearly skewed to the left, indicating that the running time of JAID on most bugs is limited. For example, JAID ran for up to 60 minutes on about 43% of all bugs, and for up to 240 minutes on about 89% of all bugs.

JAID's running time distribution for the INTROCLASSJAVA benchmark looks different from the other benchmarks: in the latter, JAID ran for less than 60 minutes on the majority of bugs; but in INTROCLASSJAVA, the most frequent running times are between 60 and 120 minutes. This is counterintuitive given that programs in INTROCLASSJAVA are generally simpler and smaller than those in the other benchmarks. We found out this behavior is due to a feature of programs in INTROCLASSJAVA, which all use a `Scanner` object to read user input from the console. Since the `Scanner` class defines many observer methods of the kinds used by JAID's heuristics

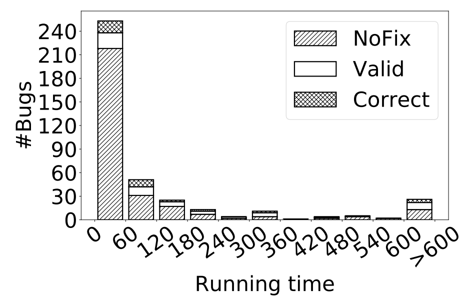
Table 3.8: Statistics on JAID's overall running time per bug (in minutes): MINimum, MAXimum, MEAN, MEDIAN, STandard DEViation, and SKEWness. Each statistics is computed over ALL bugs, bugs with a VALID fix, and bugs with a CORRECT fix.

	MIN	MAX	MEAN	MEDIAN	STDEV	SKEW
ALL	0.0	3403.6	136.3	76.3	283.8	6.3
VALID	0.6	3403.6	194.0	95.3	356.4	5.5
CORRECT	2.1	1777.7	154.5	92.8	252.0	4.7

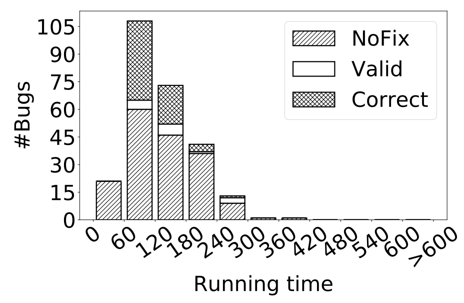
to construct snapshot expressions and to generate fix actions (see Section 3.2.2 and Section 3.2.4), JAID generated more than 10,000 candidate fixes for several bugs in INTROCLASSJAVA, thus leading to a longer cumulative running time.

The overall running times of JAID are not short in absolute terms, but they are to be expected in a tool that relies extensively on *dynamic analysis*, which requires to repeatedly execute several variants of heavily instrumented programs. JAID's performance is consistent regardless of whether the bugs will eventually be fixed: if we look at only bugs that JAID could correctly fix, it ran for up to 100 minutes on about 55% of them, and for up to 200 minutes on about 88% of them. In all, these running times are suitable for JAID's intended mode of usage as a batch (not interactive) analysis tool.

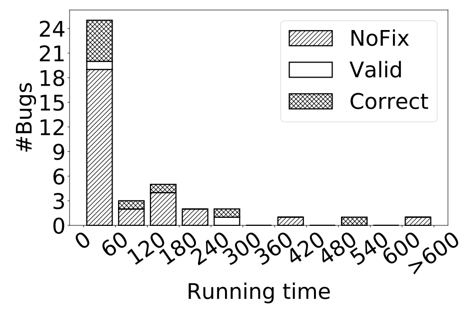
Table 3.8 breaks down statistics on the overall running time per bug into all sessions, those that produced a valid fix, and those that produced a correct fix. JAID ran in around 76 minutes per any bug on average; and in around 95 minutes per bugs on which it is successful (producing a valid or correct fix). JAID is unsurprisingly significantly slower than tools based on constraint solving and other static techniques (see Section 2.1.2); for example, Nopol [91] takes around 22 minutes per bug on average—on what, we assume, is comparable hardware. JAID's running time are, however, in line with other techniques mainly based on dynamic analysis—such as jGenProg [49] which takes about one hour per bug.



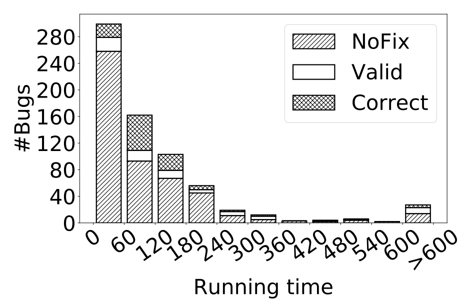
(a) Defects4J bugs.



(b) IntroClassJava bugs.



(c) QuixBugs bugs.

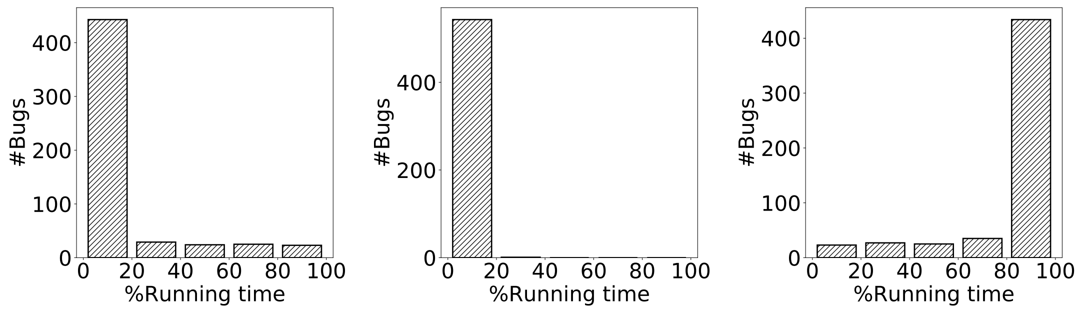


(d) All bugs.

Figure 3.4: Distribution of Jaid's running time per bug. The height of each bar spanning x coordinates a and b marks the total number of bugs whose overall fixing time was between a and b minutes. Segments of different hatch partition the number of bugs into those for which Jaid produced no fixes, only valid fixes, or correct fixes.

There is room for improving Jaid's performance by fine-tuning its implementation. To better understand the main performance bottlenecks, Figure 3.5 breaks down the running time into the different phases of Jaid's overall process: fault localization, fix generation, and fix validation. By far, *Fix validation* remains, the most time consuming phase, taking about 76% of the overall running time on all the bugs. A straightforward way to practically reduce this is to run validation of candidates in parallel, which could be implemented in future versions of Jaid.

The other phases of Jaid take proportionally much less time: fault localization accounts for no more than 20% of the running time for 443 faults, which indicates Jaid's fault localization is reasonably efficient on most faults. In a minority of cases, however, fault localization may become a bottleneck: Jaid's fault localization



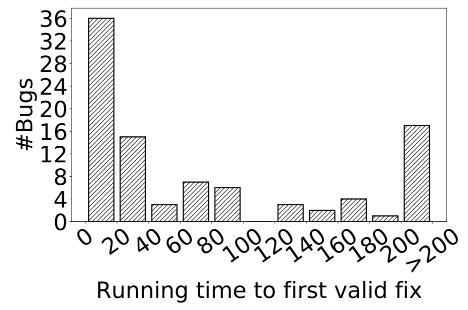
(a) Fault localization. (b) Fix generation. (c) Fix validation.

Figure 3.5: Distributions of JAID's running time per bug in each main phase of the fixing process (fault localization, fix generation, and fix validation). The height of a bar spanning x coordinates a and b denotes the number of bugs whose running time in that phase (fault localization, fix generation, and fix validation) was between a% and b% of the overall running time. The 36 faults whose fixing failed during setup phase (Section 3.3.2) are excluded from the figure.

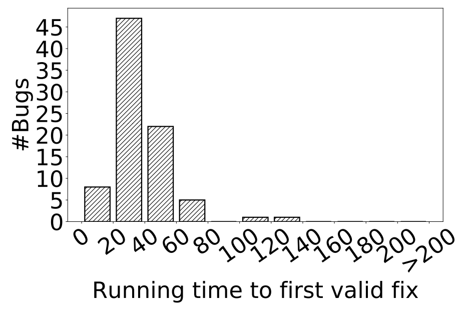
took over 60% of overall running time for 48 faults that required many snapshot expressions, had many tests, or both. One way of improving the efficiency of fault localization in these cases is therefore to *select* snapshot expressions and tests (for example, based on information gathered during fixing). As for the time spent on fix generation, it never takes more than 15% of the overall running time.

Time to first fix. In its current implementation, JAID always runs to completion, stopping only after it has validated all the generated candidate fixes. From a user's point of view, however, what matters is the time JAID takes to generate the *first valid* or the *first correct* fix: as soon as a valid fix is available, the user can start inspecting it to see if it's suitable; and as soon as a correct fix is available, the rest of JAID's output becomes redundant.

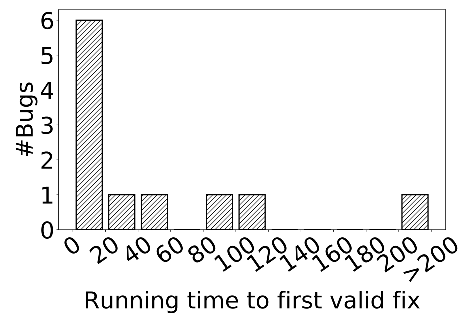
To assess JAID's efficiency from this more practical viewpoint, Figure 3.6 and Figure 3.7 picture the distributions of running time to a valid fix and to a correct fix. The x-axis of the figures gives the running time per fix (in minutes) and the y-axis gives the number of bugs. And Table 3.9 provides related statistics on the running



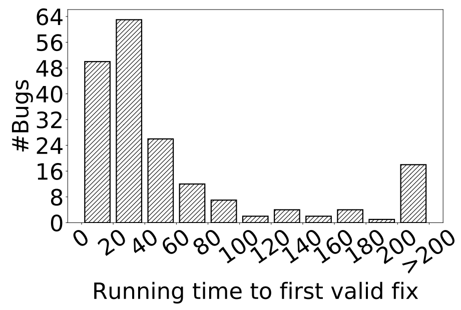
(a) DEFECTS4J bugs.



(b) INTROCLASSJAVA bugs.



(c) QUIXBUGS bugs.

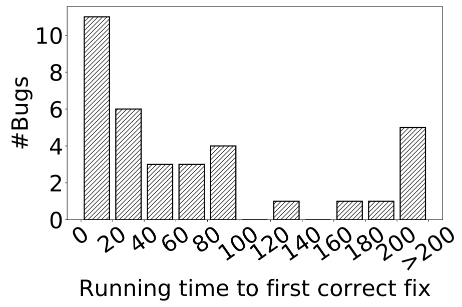


(d) All bugs.

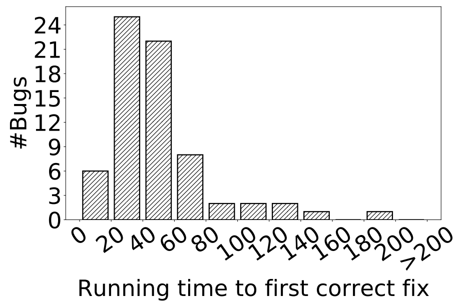
Figure 3.6: Distribution of JAID's running time until it finds a valid fix over all bugs for which JAID finds at least a valid fix. The height of a bar spanning x coordinates a and b marks the number of bugs whose running time until JAID output a valid fix was between a and b minutes.

time per fix across all sessions that produced a valid fix and those that produced a correct fix. The average running time to a valid fix is less than 30 minutes for most faults, and hence the distributions in Figure 3.6 are strongly skewed to the left. A similar trend is visible in Figure 3.7 for the time to a correct fix, even though the skewedness is less pronounced in this case: while the majority of correct fixes can be produced in less than 90 minutes, there remains a "long tail" of correct fixes that take considerably more time.

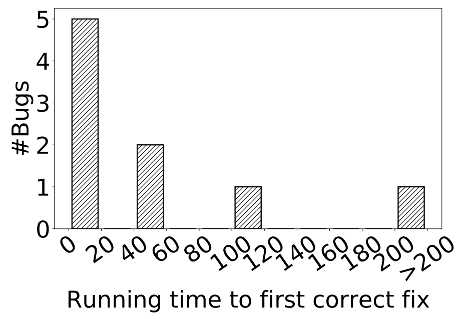
JAID validates candidate fixes in the same order of suspiciousness value of their snapshots: the more suspicious a snapshot, the earlier its derived candidate fixes are validated. In contrast, other automated program repair techniques [25, 34] adjust the



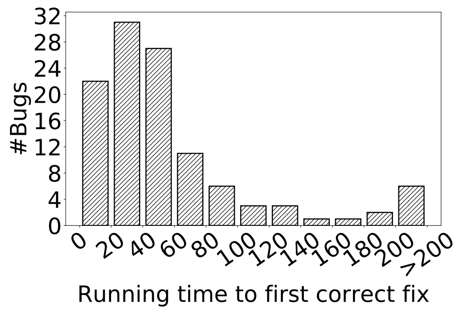
(a) DEFECTS4J bugs.



(b) INTROCLASSJAVA bugs.



(c) QUIXBUGS bugs.



(d) All bugs.

Figure 3.7: Distribution of JAID's running time until it finds a correct fix over all bugs for which JAID finds at least a correct fix. The height of a bar spanning x coordinates a and b marks the number of bugs whose running time until JAID output a correct fix was between a and b minutes.

validation order based on information obtained by mining other fixes and the relations between different fixes. JAID could add a similar prioritization approach to its validation phase. Specifically, it could associate a priority order based on the relation between, on the one hand, fix actions (Section 3.2.4) and fix schemas (Section 3.2.5), and, on the other hand, the characteristics of the bug being fixed in comparison with the features of other known bugs. For example, if fixing bugs involving null-pointer dereferencing usually requires to execute a call only conditionally on its target being non-null, Jaid's schema B in Figure 3.2 should be given higher priority.

On an average bug, JAID ran for around 76 minutes in total, but took only 2 minutes to produce the first valid fix, and 24 minutes to produce the first correct fix (when it could find one).

Table 3.9: Statistics on JAID's running time (in minutes) until a VALID or a CORRECT fix is found: MINimum, MAXimum, MEAN, MEDIAN, STandard DEViation, and SKEWness. The statistics are computed over bugs with a VALID fix (row VALID), and bugs with a CORRECT fix (row CORRECT).

	MIN	MAX	MEAN	MEDIAN	STDEV	SKEW
VALID	0	1777	45.4	2	180.2	7.6
CORRECT	0	1777	78.7	24	221.0	6.4

RQ3: Tool comparison

Table 3.10 quantitatively compares JAID to 16 other APR tools for Java on bugs from the three benchmarks.

JAID can produce *valid* fixes for several more bugs than any other tools; in DEFECTS4J, for example, JAID produced valid fixes for 56% (= (94 – 60)/60) more bugs than ssFix, which does so for 60 bugs. This indicates that JAID explores a large space of possible fixes—arguably larger than other tools. JAID outperforms any other tools also in terms of *correct* fixes, even though JAID's advantage is more limited here; in DEFECTS4J, for example, JAID's recall is 0.3% higher (1 more bug correctly fixed) than the runners up SimFix and SketchFixPP, and it is 2.3% higher (9 more bugs correctly fixed) than the next-best tool Elixir.

When it comes to *precision*, several other tools perform better than JAID on DEFECTS4J bugs, especially if we only consider correct fixes that are ranked high (in the top-10 or even in first position). JAID's precision is better than ssFix's, Nopol's, jKali's, and jGenProg's, but it is worse than the other tools for which this measure is available. All tools that outperform JAID in terms of precision are specifically designed to achieve a high precision, and have access to information mined from existing human-written patches to help identify candidate fixes that are more likely to be correct. JAID's precision may similarly improve if its algorithm also had access to the same kind of machine-learned information.

Table 3.10: A quantitative comparison of JAID with 16 other tools for automated program repair, based on their published experimental evaluations (see Table 3.4 for sources of the comparison data), and partitioned into sections for each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS. For each APR TOOL, the table reports the number of bugs that the tool could fix with a VALID fix; the number of bugs that the tool could fix with a CORRECT fix; and the resulting PRECISION (CORRECT/VALID) and RECALL (CORRECT/TOT, where TOT is the total number of bugs from the benchmark that are used in the experiments). For tools whose data about the POSITION of fixes in the output is available, the table reports number of bugs with CORRECT fixes, PRECISION, and RECALL separately for fixes ranked in ANY POSITION, in the TOP-10 POSITIONS, and in the FIRST POSITION. The rightmost column UNIQUE lists the number of distinct bugs that *only* the tool can fix correctly. Question marks represent data not available for a tool.

APR TOOL	VALID	ANY POSITION			TOP-10 POSITIONS			FIRST POSITION			UNIQUE
		CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	
DEFECTS4J											
JAID	94	35	37.2%	8.9%	25	26.6%	6.3%	14	14.9%	3.5%	11
ACS	23	18	78.3%	4.6%	18	78.3%	4.6%	18	78.3%	4.6%	11
CapGen	25	22	88.0%	5.6%	22	88.0%	5.6%	21	84.0%	5.3%	3
Elixir	41	26	63.4%	6.6%	26	63.4%	6.6%	26	63.4%	6.6%	?
HDA	?	23	?	5.8%	23	?	5.8%	13	?	3.3%	3
jGenProg	27	5	18.5%	1.3%	5	18.5%	1.3%	5	18.5%	1.3%	1
jKali	22	1	4.5%	0.3%	1	4.5%	0.3%	1	4.5%	0.3%	0
Nopol	35	5	14.3%	1.3%	5	14.3%	1.3%	5	14.3%	1.3%	2
SimFix	56	34	60.7%	8.6%	34	60.7%	8.6%	34	60.7%	8.6%	12
SketchFix	26	19	73.1%	4.8%	?	?	?	9	34.6%	2.3%	0
SketchFixPP	?	34	?	8.6%	?	?	?	?	?	?	3
ssFix	60	20	33.3%	5.1%	20	33.3%	5.1%	20	33.3%	5.1%	1
xPar	?	4	?	1.0%	4	?	1.0%	?	?	?	?
INTROCLASSJAVA											
JAID	84	69	82.1%	26.7%	43	51.2%	16.7%	30	35.7%	11.6%	?
CapGen	?	25	?	9.7%	?	?	?	?	?	?	?
JFix	?	19	?	7.4%	?	?	?	?	?	?	?
S3	?	22	?	8.5%	?	?	?	?	?	?	?
QUIXBUGS											
JAID	11	9	81.8%	22.5%	9	81.8%	22.5%	4	36.4%	10.0%	?
Astor	11	6	54.5%	15.0%	?	?	?	?	?	?	?
Nopol	4	1	25.0%	2.5%	?	?	?	?	?	?	?

JAID's precision is lagging behind other tools only on DEFECTS4J bugs; in contrast, it is consistently *higher* than other tools on INTROCLASSJAVA[5] and QUIXBUGS bugs. This difference is probably due to the kinds of tests that accompany bugs in INTROCLASSJAVA and QUIXBUGS: thorough tests, which serve as strong oracles. JAID's validation is entirely based on dynamic analysis, and hence using high-quality tests lowers overfitting and increases precision.

JAID's effectiveness on INTROCLASSJAVA and QUIXBUGS also suggests that its repair algorithm may be less prone to *overfitting* than other tools that are more focused on achieving a high precision on DEFECTS4J. A recently published large-scale experiment [15] targeted 11 repair tools for Java (ARJA, two implementations of GenProg for Java, two implementations of Kali for Java, an implementation of RSRepair for Java, Cardumen, jMutRepair, Nopol, DynaMoth, and NPEFix) that were primarily evaluated on DEFECTS4J in their original publications, and ran them on different benchmarks—including INTROCLASSJAVA and QUIXBUGS. While these experiments only considered the number of *valid* fixes (without analyzing correctness), they clearly showed that the number of bugs with valid fixes for "all 11 tools is significantly higher for bugs from DEFECTS4J compared to the other four benchmarks"—which include INTROCLASSJAVA and QUIXBUGS. This is in contrast to JAID, which built proportionally *more* valid (and correct) fixes for bugs in other benchmarks: in [15]'s experiments, none of the 11 tools could build valid fixes for more than 10% of bugs in INTROCLASSJAVA and QUIXBUGS; in our experiments, JAID built valid fixes for 28% of bugs in both INTROCLASSJAVA and QUIXBUGS.

Unique fixes. Tools that have been evaluated on the DEFECTS4J benchmark often list the identifiers of the bugs they correctly fixed; therefore, we can get a more nuanced idea of which bugs each tool can fix. Figure 3.8 displays this information

[5]While precision measures of other tools on INTROCLASSJAVA bugs are not available, JAID's precision is quite high in absolute numbers (over 82%), and hence it is likely to be overall competitive.

in a readable format; and column UNIQUE in Table 3.10 summarizes the number of bugs that each tool can fix that no other tools can. This data suggests that tools are often *complementary* in the specific bugs they are successful on: JAID fixes 11 bugs that no other tool can fix; SimFix fixes another 12; ACS fixes 11; CapGen, SketchFixPP, and HDA fixes 3 each; Nopol fixes 2; ssFix and jGenProg fixes 1 each. The complementarity between JAID and other tools is not accidental but depends on different tools focusing on different fix spaces and fix ingredients.

Consider the 23 bugs that only ACS (11 bugs) or SimFix (12 bugs) can fix: 9 require inserting statements that throw appropriate exceptions, which is something only ACS can do at the moment; 4 require changing the code at two locations simultaneously, which is something very few tools other than SimFix are capable of; 1 requires inserting a snippet with more than four lines of code, which is beyond the capabilities of most tools; and the remaining 9 require various other ingredients currently outside JAID's fix space. The complementarity implies that each technique is successful in its own domain, and suggests that *combining* techniques based on mining (such as SimFix, CapGen, HDA, and ACS) with JAID's techniques is likely to yield further improvements in terms of overall effectiveness.

Other tools. We cannot quantitatively compare APR tools that target other programming languages since they were evaluated on different benchmarks [36]. Nevertheless, just to give an idea, Angelix [54] and Prophet [46] achieve a precision of 35.7% and 42.9%, and a recall of 9.5% and 17%, on 105 bugs in the C GenProg benchmark [35]; AutoFix [61] achieves a precision of 59.3% and a recall of 25% on 204 bugs from various Eiffel projects with contracts. GenProg, AE [81], and TrpAutoRepair [64] produced [36] valid fixes for 37%, 20%, and 32% of the bugs in the original IntroClass C benchmark; the experiments [36] do not analyze how many of these fixes are *correct*.

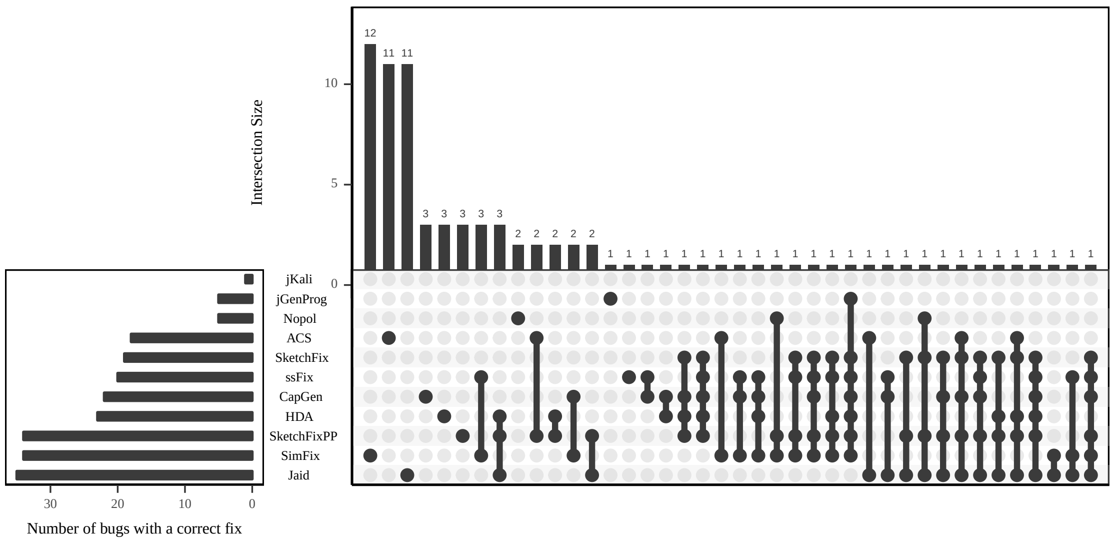


Figure 3.8: Partitioning of Defects4J bugs according to which tool can fix them. Each vertical bar measures the number of bugs that a certain combination of tools (indicated by connected dots in the lower part of the diagram) can correctly fix that no other tools can. For example, the leftmost column indicates that SimFix can correctly fix 12 bugs that no other tool can; the eighth column from the left indicates that HDA, SketchFixPP, and Jaid can all fix the same 3 bugs that no other tools can fix. The horizontal bars on the left report how many bugs in total each tool can fix.

Jaid *produced correct fixes for more bugs than any other automated repair tools for Java—and fixed 11 bugs in* Defects4J *that no other tools can fix.* Jaid *is less precise than some other tools on* Defects4J *bugs, but it is more precise on* IntroClassJava *and* QuixBugs *bugs.*

RQ4: Templates and Heuristics

Templates. As explained in Section 3.2.4, Jaid uses three kinds of *fix actions* as templates to build patches: *semantic* actions, *mutation* actions, and *control-flow* actions.

semantic actions are based on Jaid's rich state-based abstractions, and build patches that modify the state (directly or indirectly) using the expressions captured by snapshots and ranked by dynamic analysis;

mutation actions are simple, "syntactic" mutations of an existing statement that capture common sources of programming mistakes such as off-by-one errors;

control-flow actions modify the program's control flow by changing a branching condition or by adding an abrupt termination statement (`break` or `return`).

Table 3.11 shows how often each kind of fix action was used to produce *correct* fixes. Semantic and mutation actions are the most frequently used fix action kinds, but control-flow actions are also needed for a significant fraction of correct fixes. This indicates that the three kinds of actions are often complementary and all contribute to JAID's effectiveness.

JAID patches a buggy program by injecting fix actions using one of the five different *fix schemas* in Figure 3.2. A fix schema may execute the fix action unconditionally (schemas A and E) or guarded by an `if` condition (schemas B, C, and D); and may execute it instead of an existing statement (schemas D and E) or in addition to it (schemas A, B, and C). Table 3.11 indicates that schemas B and E are used a bit more often, but there is no schema that is overwhelmingly more useful, and all are required to be able to fix a wide choice of bugs.

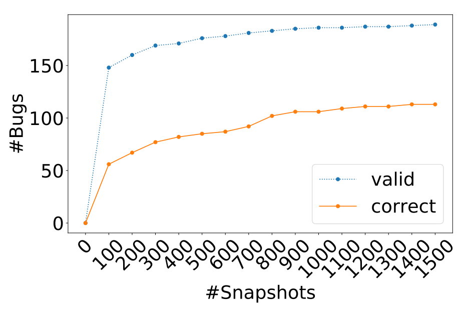
In total, 168 correct fixes instantiated schemas A, B, D, or E, which all require a synthesized fix action as a component, indicating the importance of effective fix action generation to successful program repair. The table also suggests that both conditional modifications (instantiating schemas B, C, or D) and unconditional modifications (instantiating schemas A or E) are required to target a wide choice of bugs.

Number of snapshots. JAID builds a variable number of *snapshots* to abstract program state based on its heuristics; in its default settings (Section 3.3.1), it uses up to 1500 snapshots to drive the rest of the fixing process. It's clear that the number and variety of snapshots may affect the effectiveness (more snapshots means more precise abstractions) and efficiency (more snapshots means longer analysis times) of

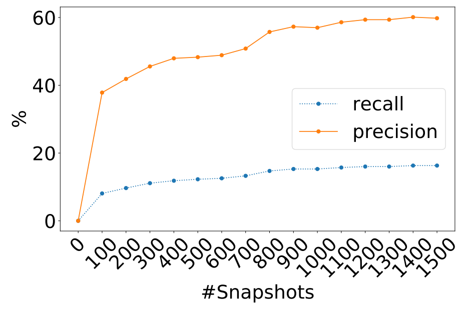
Table 3.11: Templates used more commonly by JAID to produce *correct* fixes. For bugs in each benchmark, the table lists the numbers of correct fixes produced by JAID using a semantic action (s), a mutation action (m), or a control-flow action (c); and the number of correct fixes that use one of the five schemas A–E in Figure 3.2. Since JAID may produce multiple correct fixes for the same bug, the total number of fix actions or fix schemas listed is greater than the number of bugs correctly fixed by JAID.

BENCHMARK	FIX ACTION			FIX SCHEMA				
	s	m	c	A	B	C	D	E
DEFECTS4J	14	17	13	5	15	4	6	18
INTROCLASSJAVA	71	42	2	13	36	6	24	42
QUIXBUGS	4	3	2	2	4	1	0	3
OVERALL	89	62	17	20	55	11	30	63

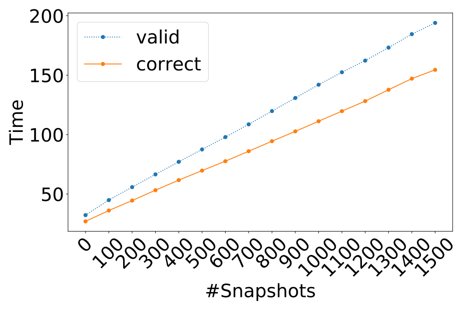
the fixing process. To understand the trade-off in a quantitative way, we ran a series of experiments where JAID used a different number of snapshots to fix all 693 bugs in the three benchmarks DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS. Figure 3.9 plots various measures of JAID's performance (number of valid and correct fixes, overall precision and recall, running time per bug, and running time until a valid or correct fix is found) as a function of the number of used snapshots—from 100 to 1500 in 100-snapshot increments. Overall, the data confirms the intuition that more snapshots means longer running times (Figure 3.9c and Figure 3.9d) but also a greater chance of success (Figure 3.9a) and better precision (Figure 3.9b). However, while the running time grows uniformly proportionally to the number of snapshots, there are diminishing returns in adding more snapshots to increase precision and recall, since as the search space becomes larger it is harder to search it effectively. Thus, further substantial progress in effectiveness is not only a matter of extending the search space but requires more structured ways of exploring it.



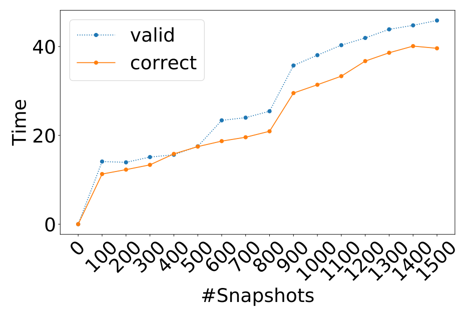
(a) Number of bugs with valid or correct fixes.



(b) Precision and recall.



(c) Average running time per bug.



(d) Average time until a valid or correct fix is found.

Figure 3.9: How effectiveness and efficiency of JAID depend on the number of snapshots used for fixing. Each figure plots the number of bugs with a valid or correct fix (Figure 3.9a), overall precision and recall (Figure 3.9b), average running time per bug (Figure 3.9c), and average time until a valid or a correct fix is built (Figure 3.9d) as a function of the number of snapshots used for fixing. The data comes from experiments using all 693 bugs in the three benchmarks DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS.

JAID's templates are all needed to be able to effectively fix bugs with different characteristics. Reducing the number of snapshots used by JAID improves its running time but also reduces the number of bugs that can be fixed.

RQ5: Fault Localization

Do JAID's effectiveness and efficiency depend on the fault-localization algorithm it uses? To answer this question, we ran five different variants of JAID, each using one of the fault localization algorithms of Table 3.5 instead of JAID's default algorithm

Table 3.12: How JAID's effectiveness changes using different fault localization techniques. For each of the five spectrum-based Fault Localization algorithms of Table 3.5, the table reports the number of fixes in each benchmark that JAID could repair with a Valid or a Correct fix in ALL benchmarks, and in each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS individually. The first row refers to JAID using its original fault-localization technique (as in the rest of the experimental evaluation).

	ALL		DEFECTS4J		INTROCLASSJAVA		QUIXBUGS	
FL	C	V	C	V	C	V	C	V
JAID	113	189	35	94	69	84	9	11
Barinel	111	191	33	96	69	84	9	11
DStar	111	191	33	96	69	84	9	11
Ochiai	111	191	33	96	69	84	9	11
Op2	111	191	33	96	69	84	9	11
Tarantula	111	191	33	96	69	84	9	11

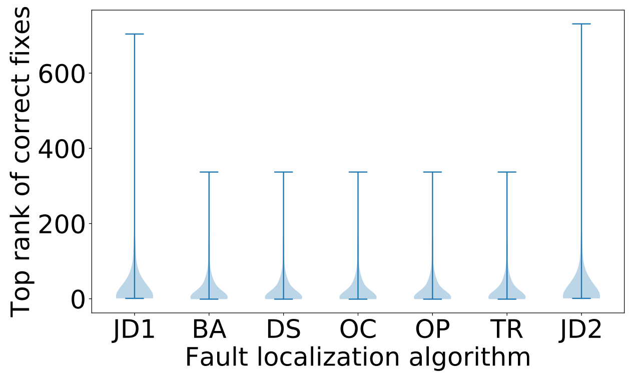
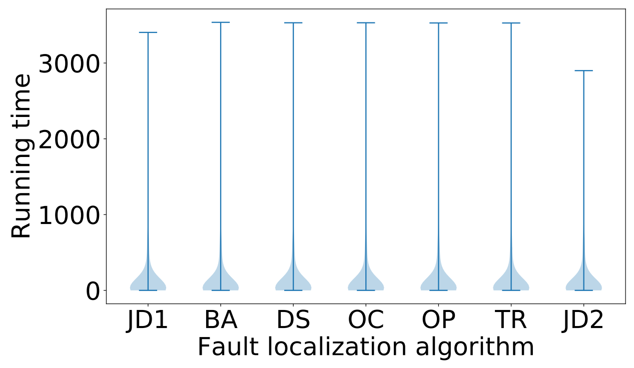
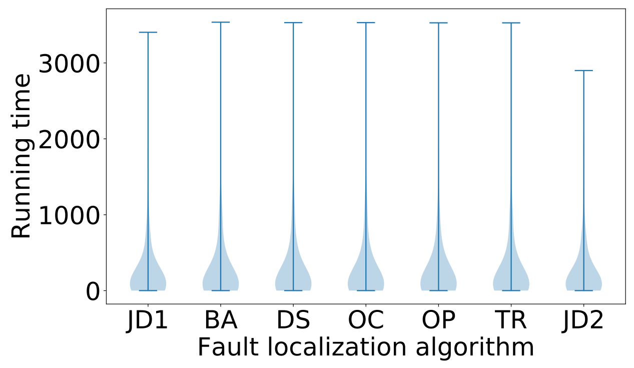


Figure 3.10: Violin plots of the top rank of correct fixes for all bugs fixed by JAID using different fault localization algorithms: JAID's custom algorithm (JD), Barinel (BA), DStar (DS), Ochiai (OC), Op2 (OP), and Tarantula (TR).

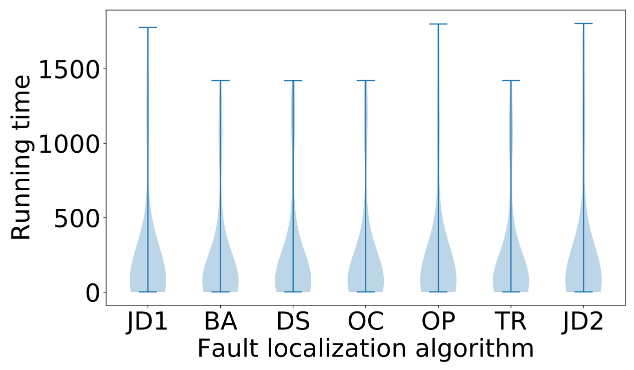
(used in the rest of the experiments). Table 3.12 shows the number of faults fixed (with a correct fix, or just with a valid fix) in each case. The differences between fault-localization techniques are very small: there are only three bugs in total that JAID can or cannot fix depending on the fault localization algorithm it uses. Precisely, JAID failed to produce any valid fixes for bug `Lang53` in DEFECTS4J when using its



(a) All bugs.



(b) Bugs with a valid fix.



(c) Bugs with a correct fix.

Figure 3.11: Violin plots of the overall running time of JAID using different fault localization algorithms: JAID's custom algorithm (JD), Barinel (BA), DStar (DS), Ochiai (OC), Op2 (OP), and Tarantula (TR).

default fault localization algorithm, since the one snapshot from which valid fixes could be derived was not ranked among the top 1500 most suspicious; in contrast, JAID could only correctly fix bugs `Closure33` and `Chart9` in DEFECTS4J using its default fault localization algorithm, since the other five algorithms ranked the "useful" snapshots too low to be used.

Figure 3.10 looks at how the *top rank* of a correct fix varies with the choice of fault localization algorithm. Here too differences between JAID's default algorithm and any other algorithm are quite limited—with JAID's default algorithm leading to a greater variance in the rank but no noticeable differences on average.

Finally, Figure 3.11 plots JAID's running time using different fault localization algorithms. Once again, there are no marked differences between JAID's default algorithm and any other algorithm. The overall differences among different techniques on the bugs are only marginal. We also applied the non-parametric Mann-Whitney U-test to check for the statistical difference among the fixing time of JAID. None of the p-value produced by the test was smaller than 0.02, therefore we conclude the differences are not statistically significant. Comparisons of the fault localization techniques based on bugs from benchmarks INTROCLASSJAVA and QUIXBUGS produce similar results.

In summary, JAID's overall technique is remarkably robust with respect to the choice of fault localization algorithm. Note, however, that all fault localization algorithms are *spectrum-based*, and hence process the same information (the spectrum over the available executions) using slightly different heuristic formulas (see Table 3.5). Using radically different fault localization approaches [86] may still have a significant impact on the performance of automated program repair.

JAID*'s effectiveness and efficiency are largely independent of the spectrum-based fault localization algorithm that is used.*

3.4 Summary

We presented Jaid, a new technique and tool for the automated repair of Java programs. Jaid automatically builds rich state-based abstractions of faulty object-oriented programs, which buttress effective fault localization, and fix generation and ranking processes. Our comprehensive experimental evaluation of Jaid involves 693 bugs from the Defects4J, IntroClassJava, and QuixBugs benchmark suites. Jaid produced correct fixes for 113 bugs with a precision of nearly 60%.

Major progress in the precision and applicability of automated program repair typically requires tapping into additional sources of information about program behavior: Jaid relies on a rich state-based dynamic analysis; other approaches mine repositories of code and fixes [34, 89]. Moreover, our experiments suggest that Jaid's overall effectiveness does not depend much on the details of its spectrum-based fault localization algorithm; they also indicate that further substantial progress would probably require to step up the precision of fault localization in a way that it can incorporate such additional sources of information.

Chapter 4

Enhancing Program Repair with Retrospective Fault Localization-RESTORE

The work described in this chapter is published in [90].

Automated program repair has the potential to transform programming practice: by automatically building fixes for bugs in real-world programs, it can help curb the large amount of resources (in time and effort) that programmers devote to debugging. A crucial ingredient of most repair techniques—especially of search-based approaches like JAID—is *fault localization.* Imitating the debugging process followed by human programmers, fault localization aims to identify program locations that are implicated with a fault and where a patch should be applied. Fault localization in program repair has to satisfy two apparently conflicting requirements: it should be accurate (leading to few locations highly suspicious of error), but also efficient (not taking too much running time).

In this chapter, we introduce a novel fault localization approach—called *retrospective fault localization*—that improves accuracy while simultaneously boosting efficiency by *integrating* closely within standard automated program repair techniques. By providing a more effective fault localization process, retrospective fault localization expands the space of possible fixes that can be searched practically. Retrospective fault localization leverages mutation-based fault localization [59, 59] to boost localization accuracy. Since mutation-based fault localization is notoriously time consuming, a key idea is to perform it as a *derivative* of the usual program repair process. Precisely, retrospective fault localization introduces a *feedback loop* that reuses (instead of just discarding) the candidate fixes that fail validation to enhance the precision of fault localization. Candidate fixes that pass some tests that the original (buggy) program failed are probably closer to being correct, and hence they are used to refine fault localization so that other similar candidate fixes are more likely to be generated.

We implemented retrospective fault localization in a tool called RESTORE, built on top of JAID [9] (described in Chapter 3). We also conducted experiments with real-world bugs from the DEFECTS4J benchmark [27] to evaluate RESTORE. The results indicate that retrospective fault localization significantly improves the overall effectiveness of program repair in terms of correct fixes (for 41 faults in DEFECTS4J, 8 more than any other automated repair tool for Java as of May 2019) and boosts its efficiency (cutting JAID's running time to a third or less). Other measures of performance suggest that retrospective fault localization improves the efficiency of automated program repair by supporting accurate fault localization with comparatively moderate resources.

In the rest of this chapter, we present an example (in Section 4.1) to explain how RESTORE works (in Section 4.2), and demonstrate its consistent performance improvements on standard benchmarks of real-world bugs (in Section 4.3). Lastly, Section 4.4 summarizes this chapter.

```
1 private void processRequireCall(NodeTraversal t, Node n, Node parent) {
2   ProvidedName provided = providedNames.get(...);
3   ...
4   if (provided != null) {
5     parent.detachFromParent();
6     compiler.reportCodeChange();
7   }
8 }
```

Listing 4.1: Faulty method `processRequireCall` from class `ProcessClosure-Primitives` in project *Closure Compiler*.

```
if (provided != null || requiresLevel.isOn()) {
```

Listing 4.2: Fix written by tool developers (replacing line 4 in Listing 4.1), and also produced by RESTORE.

4.1 An Example of RESTORE in Action

The *Closure Compiler* is an open source tool that optimizes JavaScript programs to achieve faster download and execution times. One of the refactorings it offers—renaming classes so that namespaces are no longer needed—is based on class `Process- ClosurePrimitives` whose methods modify calls to common namespace manipulation APIs. In particular, method `processRequireCall` processes calls to the `goog.require` API and determines if they can be removed without changing program behavior.

Listing 4.1 shows part of the method's implementation, which is defective:[1] according to the tool documentation, a call to `goog.require` should be removed (lines 5 and 6) if: (*i*) the required namespace can be resolved successfully (`provided != null`), *or* (*ii*) the tool is configured to remove all the calls to `goog.require` unconditionally (`requiresLevel.isOn()`). But the code in Listing 4.1 only checks condition (*i*) on line 4, and hence does not remove unresolvable calls even when condition (*ii*) holds.

Using some of the tests that come with *Closure Compiler*'s source code, the RESTORE produces the fix shown in Listing 4.2, which is identical to the one written

[1]Fault *Closure113* in DEFECTS4J [27].

by *Closure Compiler*'s tool developers—and completely fixes the bug. At the time of writing, Restore is the only automated program repair tool capable of correctly fixing this bug[2].

The features of method `processRequireCall` and its enclosing class `Process-ClosurePrimitives` contribute to making the bug challenging for generate-and-validate automated repair tools. First, class and method are relatively large (Class `ProcessClosurePrimitives` has 1233 lines and method `processRequireCall` has 40 lines), which is a challenge in and of itself for precise fault localization. Second, attribute `requiresLevel` is never referenced in the faulty version of `processRequireCall` and is used only once after initialization in the whole class; thus, expression `requires-Level.isOn()`—which is needed for the fix—is unlikely to be selected by techniques that look for fixing "ingredients" mainly in a fault's context.

Restore's retrospective fault localization is crucial to ensure that the necessary fixing expression is found in reasonable time: Restore takes around 32 minutes to produce the fix in Listing 4.2) and to rank it first in the output. This indicates that Restore's search for fixes is not only efficient but also effective. Lastly, Section 4.4 summaries this chapter.

4.2 How Restore Works

Retrospective fault localization is applicable in principle to any generate-and-validate automated program repair technique to improve its efficiency. To make the presentation more concrete, we focus on how retrospective fault localization is applicable on top of our automated program repair tool—Jaid [9] (described in Section 3.2). We call the resulting technique, and its supporting tool, Restore.

[2]Nopol was able to produce a valid, but incorrect, fix to the fault [14].

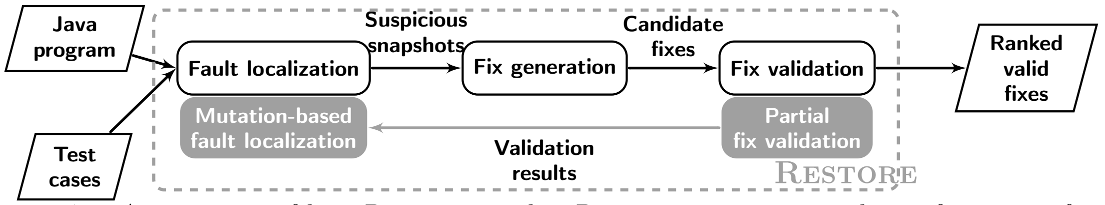


Figure 4.1: An overview of how RESTORE works. RESTORE can improve the performance of any generate-and-validate automated program repair tool. Such a tool inputs a faulty program and some test cases exercising the program. The first, crucial, step of fixing is *fault localization*, which determines a list of snapshots: program states that are indicative of error; for each suspicious snapshot, *fix generation* builds a number of candidate fixes of the input program by exploring a limited number of program mutations that may avoid the suspicious states; *fix validation* reruns the available tests on each candidate built by fix generation; only candidates that pass all tests are *valid fixes*, which are the tool's output to the user.
RESTORE kicks in during the first run of such a program repair tool, by introducing a feedback loop (in grey) that improves the effectiveness of fault localization. RESTORE performs a *partial fix validation*, whose goal is quickly identifying candidate fixes that fail validation—which are treated as *mutants* of the input program; information about how mutants' behaviors differ from the input program supports a *mutation-based fault localization* step that sharpens the identification of suspicious snapshots. As we demonstrate in Section 4.3, RESTORE's feedback loop significantly improves effectiveness and efficiency of automated program repair.

4.2.1 Overview

Figure 4.1 illustrates how RESTORE works at a high level, and how it enhances a traditional automated program repair technique by retrospective fault localization (boxes in grey in Figure 4.1).

Input. RESTORE inputs a Java program $\boldsymbol{P}$ (a collection of classes), with a faulty method `m2f`, and a set $\boldsymbol{T}$ of test cases exercising $\boldsymbol{P}$. Since each run of RESTORE actually only uses tests that exercise `m2f`, we assume, without loss of generality, that $\boldsymbol{T}$ only includes such tests.

Fault localization identifies program locations and states (called *snapshots*) that are indicative of faulty behavior. According to heuristics based on dynamic and static measures, each snapshot receives a *suspiciousness score*—the higher, the more suspicious; snapshots ranked according to their suspiciousness score are input to the next step: fix generation.

Fix generation builds several modifications of input program $\boldsymbol{P}$ for each snapshot in order of suspiciousness. The modifications try to mutate $\boldsymbol{P}$'s behavior in a way that avoids reaching the suspicious snapshot's state. Fix generation's output is a sequence of *candidate fixes* that needs to be validated.

(Full) fix validation tests each candidate fix to determine whether it actually fixes the fault exposed by $\boldsymbol{T}_{\times}$. In traditional automated program repair, fix validation runs all available tests $\boldsymbol{T}$ against each fix candidate, and only outputs candidates that pass all tests—ranked according to the suspiciousness of the snapshots they were derived from. Hence, fix validation is often the most time-consuming step of traditional automated program repair. Since it is done downstream from fix generation—as the last step of the whole fixing process—validation requires a large number of fix candidates to maximize the chance of finding some valid, possibly correct, fixes, which exacerbates the performance problem.

Partial fix validation is the lightweight form of validation of candidate fixes used by RESTORE to support retrospective fault localization. By only running a subset of the available tests $\boldsymbol{T}$, partial fix validation aims to quickly detect *behavioral changes* in some of the candidates with respect to the program $\boldsymbol{P}$ under fix.

Mutation-based fault localization improves the precision and effectiveness of fault localization by using *retrospective* information coming from partial validation. Based on this information, the suspiciousness score of snapshots is revised to become more discriminatory.

Exploring a larger fix space. With retrospective fault localization, the top-ranked snapshots have a *higher chance* of leading to *valid fixes* when used in the following phases of the repair technique—and thus to correct fixes ranked high in the overall output. Conversely, a higher-precision fault localization technique means that *fewer candidates* need to be generated and (fully) validated, leading to an overall faster process. In turn, RESTORE's more efficient search of the fix space allows it to explore a *larger space* in comparable—often *shorter*—time, ultimately leading to discovering fixes that are outside JAID's fix space.

4.2.2 Retrospective Fault Localization

The ultimate goal of automated program repair is finding fixes that are not only valid—pass all available tests—but *correct*—equivalent to those a competent programmer, knowledgeable of the program $\boldsymbol{P}$ under repair, would write. The traditional automated program repair process presented in Chapter 3 can be quite effective at producing correct fixes but is limited in practice by two related requirements:

1. since the accuracy of fault localization greatly affects the chances of success of the whole repair process, we would like to have a fault localization technique that incorporates as much information as possible;
2. since the process is open loop (no feedback), we have to generate as *many candidate fixes* as possible to maximize the chance of finding a correct one.

Improving accuracy and generating many candidate fixes both exacerbate the already significant problem of *long validation times* (for example, validation takes up 92.8% of JAID's overall running time [9]). More crucially, they require to bound the search space of possible fixes to a *size* that can be feasibly explored. But, by definition, shrinking the fix space makes some bugs impossible to fix.

Retrospective fault localization, as implemented in RESTORE, addresses these two requirements with complementary solutions:

1. it performs a preliminary *partial fix validation*, which runs much faster than full validation and whose primary goal is to supply more dynamic information to fault localization;
2. using the information from partial validation, it complements JAID's fault localization with precise *mutation-based fault localization*.

Such a feedback-driven mutation-based fault localization drives more efficient further iterations of fix generation, producing a much smaller, often higher-quality, number of candidate fixes that can undergo full validation taking a reasonable amount of time. The greater efficiency is then traded off against fix space size: RESTORE can afford to explore a *larger space of candidate fixes*, thus ultimately fixing bugs that are out of JAID's (and other repair tools') capabilities.

4.2.3 Initial fix generation

The initial iteration of fix generation in RESTORE works similarly to basic automated program repair: fault localization (Section 3.2.3) assigns a basic suspiciousness score $su_B(s)$ to every snapshot s (using spectrum-based fault localization as in JAID); and fix generation (Section 3.2.4 and Section 3.2.5) builds fix candidates for the most suspicious snapshots.

As we have already remarked, JAID's spectrum-based fault localization often takes a major part of the total fixing time, as it involves monitoring the values of many snapshot expressions in every test execution; for example, it takes 51%–99% of JAID's total time on 16 hard faults [9]. To cut down on this major time cost, RESTORE *selects* a subset $\boldsymbol{T}_B$ of all tests $\boldsymbol{T}$ to be used in basic fault localization using nearest neighbor queries [68]. The selected tests $\boldsymbol{T}_B$ include all failing tests $\boldsymbol{T}_{\times}$ as well as the passing tests with the *smallest distance* to those failing. The distance between two tests t_1, t_2 is calculated as the Ulam distance[3] $U(\phi(t_1), \phi(t_2))$, where $\phi(t)$ is a sequence with all basic blocks of `m2f`'s control-flow graph sorted according to how many times each block is executed when running t. This way, passing tests that are behaviorally similar to failing tests are selected as "more useful" for fault localization since they are more likely to be sensitive to fixes of the fault. Take,

[3]The Ulam distance [12] of two sequences is the minimum number of delete, shift, and insert operations to go from one sequence to another. For example, the Ulam distance $U(s_1, s_2)$ of $s_1 = a\,b\,c\,t\,u$ and $s_2 = a\,b\,t\,c\,u$ is 2 (delete c from s_1 and insert it back after t).

for example, the conditional at lines 4–6 in Listing 4.2; two tests t_1 and t_2 such that `provided != null` at line 4 both execute the conditional block, and hence will have a shorter Ulam distance than t_1 and another test t_3 that skips the conditional block (such that `provided == null` at line 4). Subset $\boldsymbol{T}_B$ is used only to bootstrap Restore's initial fix generation without dominating the overall running times.

During initial fix generation, Restore builds fix candidates for the $N_1 = N_S \cdot N_P$ most suspicious snapshots (whereas Jaid builds candidates for the N_S most suspicious snapshots). Parameter N_P is 10% (i.e., $N_P = 0.1$) by default; this works because retrospective fault localization can be as effective as Jaid's basic fault localization with a fraction of the snapshots.

4.2.4 Partial fix validation

Partial fix validation aims at quickly extracting dynamic information about the many candidate fixes built by the initial iteration of fix generation. To strike a good balance between costs (time spent on running tests) and benefits (information gathered to guide mutation-based fault localization), partial fix validation follows the simple strategy of running only the tests $\boldsymbol{T}_{✖}$ that were failing on the input program $\boldsymbol{P}$. This is efficient—because $|\boldsymbol{T}_{✖}|$ is often much smaller than $|\boldsymbol{T}_{✔}|$—and still has a good chance of providing valuable information for fault localization, since it detects whether the failing behavior has changed in some of the fix candidates.

If a candidate fix happens to pass all tests $\boldsymbol{T}_{✖}$, it immediately undergoes full validation (Section 4.2.8) for better responsiveness of the fixing process (outputting valid fixes as soon as possible).

4.2.5 Mutation-based fault localization

In mutation-based fault localization [59, 56], we compare the dynamic behavior of many different *mutants* of a program.

A mutant is a program variant produced by changing the program's code in some ways—for example, by changing a comparison operator. A mutant M of a program $\boldsymbol{P}$ is *killed* by a test t when M behaves differently from $\boldsymbol{P}$ on t; that is, either $\boldsymbol{P}$ passes t while M fails it, or $\boldsymbol{P}$ fails t while M passes it. A killed mutant M indicates that the locations where M syntactically differs from $\boldsymbol{P}$ are likely (if M fails) or unlikely (if M passes) to be implicated with the failure triggered by t.

RESTORE's retrospective fault localization treats candidate fixes as *higher-order mutants*—that is, mutants of the input program $\boldsymbol{P}$ that may include *multiple* elementary mutations—and interprets partial fix validation results of those higher-order mutants in a similar way to help locate faults more accurately. In particular, adapting [59]'s heuristics to our context, we assign a suspiciousness score $su_M(C)$ to each *candidate fix* C:

$$su_M(C) \;=\; \frac{|\boldsymbol{T}_{\text{✗}} \cap \mathit{killed}(C)|}{\sqrt{|\boldsymbol{T}_{\text{✗}}| \cdot |\mathit{killed}(C)|}}\,, \tag{4.1}$$

where $\mathit{killed}(C) \subseteq \boldsymbol{T}_{\text{✗}}$ is the set of all tests that kill C—and thus $\boldsymbol{T}_{\text{✗}} \cap \mathit{killed}(C)$ are the tests that fail on input program $\boldsymbol{P}$ and pass on C. Formula (4.1) assigns a higher suspiciousness to a candidate fix the more failing tests it manages to pass, indicating that C might be closer to correctness than $\boldsymbol{P}$.

In order to combine the output of mutation-based and basic fault localization, we assign a suspiciousness score $su_M(s)$ to each *snapshot* s based on the suspiciousness (4.1) of *candidates*. Each candidate fix D is generated from some snapshot $\sigma(D)$; let $SU(D)$ be the largest suspiciousness score of all candidate fixes E generated from the same snapshot $\sigma(D)$ as D:

$$SU(D) \;=\; \max_{E} \left\{ su_M(E) \mid \sigma(E) = \sigma(D) \right\}.$$

Then, the mutation-based suspiciousness score $su_M(s)$ of a *snapshot* $s = \langle \ell, e, v \rangle$ is the average of $SU(D)$ across all candidate fixes D generated from a snapshot with the same location ℓ as s (and any expression and value):

$$su_M(\langle \ell, e, v\rangle) \;=\; \underset{D}{\text{mean}} \left\{ SU(D) \mid \sigma(D) = \langle \ell, *, * \rangle \right\} . \tag{4.2}$$

The maximum selects, for each snapshot, the candidate fix generated from it that is more "successful" at making failing tests pass. Then, all snapshots with the same location get the same "average" suspiciousness score. Intuitively, the average pools the information from different fixes that target different locations and pass partial validation.

Finally, we combine the basic suspiciousness score su_B and the mutation-based suspiciousness score su_M into an overall total ordering of snapshots according to their suspiciousness:

$$s_1 \preceq s_2 \;\triangleq\; \begin{array}{rl} & \big(\ell_1 \neq \ell_2 \;\wedge\; su_M(s_1) \geqslant su_M(s_2)\big) \\ \vee & \big(\ell_1 = \ell_2 \;\wedge\; su_B(s_1) \geqslant su_B(s_2)\big) \end{array} ,$$

where $s_1 = \langle \ell_1, e_1, v_1 \rangle$ and $s_2 = \langle \ell_2, e_2, v_2 \rangle$. That is, snapshots referring to different locations are compared according to their mutation-based suspiciousness, and snapshots referring to the same location are compared according to their basic suspiciousness—because they have the same mutation-based suspiciousness score. RESTORE assigns a basic suspiciousness score to each *snapshot* as fault localization discussed in Section 3.2.3; whereas the mutation-based suspiciousness score (4.2) is the same, by definition, for all snapshots with the same location.

An example of how MBFL works. To get a more intuitive idea of how mutation-based fault localization can help find suitable fix locations in RESTORE, let's consider again fault *Closure113* in DEFECTS4J—shown in Figure 4.1 and discussed in Section 4.1. A single failing test case $\boldsymbol{T}_{✖} = \{t_{✖}\}$ triggers the fault by reaching line 4 with `provided == null`: execution skips the *then* branch (lines 5 and 6), which eventually leads to a failure.

During the initial round of fix generation, Restore does not produce any valid fix, because a key fix ingredient (expression `requiresLevel.isOn()`) is further out in the fix search space. However, it generates 16 candidate fixes that happen to pass the originally failing $t_{\times}$ because they all force execution through lines 5 and 6 by changing condition `provided != null` on line 4. For example, one such fixes replaces it with `provided != null || provided == null`. None of these 16 candidates is valid (because they all fail other, previously passing, tests) but, instead of simply being discarded, they all are reused as evidence—to increase the suspiciousness score of line 4:

1. $su_M(C) = 1$ for each of these 16 candidates, because $|\boldsymbol{T}_{\times}| = 1$ and $\mathit{killed}(C) = \boldsymbol{T}_{\times}$;
2. $SU(C) = su_M(C)$ for the same candidates, because they all have the same (maximum) value of suspiciousness;
3. $su_M(\langle \ell = 4, *, * \rangle) = 1$ for all snapshots that target line 4.

Since no other candidates generated in this round change the suspiciousness of other locations, the net result is that the following iterations of fix generation will preferentially target fixes at line 4. This biases the search for fixes so that Restore goes deeper in this direction of the fix search space, which eventually leads to generating the correct fix shown in Listing 4.2—which indeed targets line 4 with a suitable condition.

4.2.6 Retrospective loop iteration

Equipped with the refined fault localization information coming from mutation-based fault localization, Restore decides whether to iterate the retrospective fault localization loop—entering a new round of initial fix generation (Section 4.2.3)—or to just use the latest fault localization information to perform a final fix generation (Section 4.2.7). While the retrospective feedback loop could be repeated several times (until all snapshots are used to build candidates), we found that there are diminishing returns in performing many iterations. Thus, the default setting is to stop iterating as soon as mutation-based fault localization assigns a *positive* suspiciousness score $su_M(s)$ to *some* snapshot s; if no snapshot gets a positive score, we repeat initial fix generation.

4.2.7 Final fix generation

Snapshots ranked according to the $\preceq$ relation drive the final generation of fixes. Final fix generation runs when retrospective fault localization has successfully refined the suspiciousness ranking of snapshots (Section 4.2.6)—hopefully identifying few promising snapshots. Thus, final fix generation generates fixes *only* for snapshots corresponding to the N_L most suspicious locations—with $N_L = 5$ by default.

During final fix generation, RESTORE can even afford to trade off some of the greater precision brought by retrospective fault localization for a *larger fix space* to be explored: whereas JAID builds fix candidates based only on expressions found in method `m2f` (the method being fixed), RESTORE may also consider expressions found anywhere in `m2f`'s enclosing *class* `C2f`. RESTORE can efficiently search such a larger fix space, thus significantly expanding its overall fixing effectiveness.

4.2.8 (Full) fix validation

The final validation is, as in basic automated program repair, full—that is, uses *all* available tests $\boldsymbol{T}$ and validates candidate fixes that pass all of them. This validation has a higher chance of being significantly faster than in basic automated program repair: first, it often has to consider fewer candidate fixes (Section 4.2.7) selected according to their mutation-based suspiciousness; second, several candidate fixes have already undergone partial validation against failing tests $\boldsymbol{T}_{✖}$ (Section 4.2.4), and thus only need to be validated against the originally passing tests $\boldsymbol{T}_{\checkmark}$.

Fixes that pass validation are output to the user in the same order of suspiciousness $\preceq$ as the snapshots used to generate them. Thus, RESTORE's overall output is a list of valid fixes ranked according to suspiciousness.

4.3 Experimental Evaluation

We implemented the RESTORE technique in a tool, also called RESTORE, based on the JAID program repair system. Our experimental evaluation assesses to what extent RESTORE is an effective automated program repair tool by comparing:

- RESTORE's results on high-level metrics, such as *bugs correctly fixed*, to other program repair tools for Java;
- RESTORE's results on fine-grained metrics, such as the effectiveness of *fault localization*, to JAID—a state-of-the-art repair tool for Java which RESTORE directly extends;

Overall, the evaluation indicates that RESTORE is a substantial advance in general-purpose automated program repair for Java. Different parts of the evaluation have different levels of granularity, so that the we can also track *which* ingredients used by RESTORE are effective and which metrics they impact.

4.3.1 Experimental Design

The setup of experiment is consistent with the setup description in Section 3.3.1 if not specify explicitly.

Research Questions

The experiments address the following research questions:

RQ1: What is RESTORE's *effectiveness* in fixing bugs?

In RQ1, we consider RESTORE from a user's perspective: how many valid and correct fixes it can generate.

RQ2: What is RESTORE's *performance* in fixing bugs?

In RQ2, we consider RESTORE's efficiency: how quickly it runs versus how large a fix space it explores.

RQ3: How well does retrospective *fault localization* (RFL) work in RESTORE?

In RQ3, we zoom in on RESTORE's fault localization technique to assess how efficiently it drives the search for a valid fix.

Subjects. Experiments use real-world faults in version `#a910322b`[4] of the DEFECTS4J curated collection [27] (described in Section 3.3.1), which includes 357 faults in 5 projects.

Comparison to other tools. We compare RESTORE's results on high-level metrics to the 13 state-of-the-art automated program repair systems for Java listed in Table 4.1. To our knowledge these 13 tools include all recent Java repair tools evaluated on DEFECTS4J and published, at the time of writing, in major software engineering conferences in the last couple of years.

Statistics

For each measure m taken during the experiments (e.g., time T), let $J_{m,k}$ and $R_{m,k}$ denote the value of m in JAID's and in RESTORE's run on fault k. We compare RESTORE to JAID using these metrics (illustrated and justified below) [21]:

$\frac{\sum \textbf{Restore}}{\sum \textbf{Jaid}}$**:** the ratio $\sum_k J_{m,k} / \sum_k R_{m,k}$ expressing the *relative cost* of RESTORE over JAID for measure m.

[4]This is an early version of DEFECTS4J, which do not contain the `Mockito` repository.

mean(**Jaid** – **Restore**)**:** the *mean difference* (using arithmetic mean) $\text{mean}_k(J_{m,k} - R_{m,k})$ expressing the *average additional cost* of Jaid over Restore for measure m.

$b_l, \widehat{b}, b_h$**:** the estimate $\widehat{b}$ and the 95% probability interval (b_l, b_h) of the *slope* b of the linear regression $R_{m,k} = a + b \cdot J_{m,k}$ expressing Restore's measure m as a linear function of Jaid's.

$\widehat{\chi}, \chi_h$**:** for the same linear regression, the estimate $\widehat{\chi}$ and the 95% probability upper bound χ_h of the crossing ratio (where the regression line crosses the "no effect" line).

Each summary statistics compares Restore to Jaid on faults on which the statistics is defined for both tools; for example, the mean difference of measure C (rank of first correct fix) is over the 23 faults that *both* Restore and Jaid can correctly fix.

Interpretation of linear regression. A linear regression $y = a + b \cdot x$ estimates coefficients a (intercept) and b (slope) in a way that best captures the relation between x and y. A linear regression algorithm outputs *estimates* $\widehat{a}$ and $\widehat{b}$ and *standard errors* ϵ_a and ϵ_b for both coefficients: the "true" value of a coefficient c lies in interval (c_l, c_h), where $c_l = \widehat{c} - 2\,\epsilon_c \leqslant \widehat{c} \leqslant \widehat{c} + 2\,\epsilon_c = c_h$, with 95% probability.

In our experiments, values of x measure Jaid's performance and values of y measure Restore's; thus, the linear regression line expresses Restore's performance as a linear function of Jaid's. The line $y = x$ (that is, $a = 0$ and $b = 1$) corresponds to *no effect*: the two tool's performances are identical. In contrast, lines that lie *below* the "no effect" line indicate that Restore measures consistently *lower* than Jaid; since for all our measures "lower is better", this means that Restore performs better than Jaid. Plots such as those in Figure 4.3 display the estimated regression line with a shaded area corresponding to the 95% probability error interval; thus we can visually inspect whether the difference with respect to the dashed "no effect" line is significant with 95% probability by checking whether the shaded area lies under the dashed line.

Analytically, Restore is *significantly better* than Jaid at the 95% probability level if the 95% probability upper bound b_h on the regression slope's estimate satisfies $b_h < 1$: the slope is different from (in fact, less than) the "no difference" value 1 with 95% probability.

Since this notion of significant difference does not consider the intercept, it only indicates that Restore's is better *asymptotically*; to ensure that the difference is significant in the range of values that were actually measured, we consider the *crossing ratio* $\widehat{\chi} = (\overline{x} - \min(\text{Jaid}))/(\max(\text{Jaid}) - \min(\text{Jaid}))$, which expresses the coordinate $x = \overline{x}$ where the regression line $y = \widehat{a} + \widehat{b}x$ crosses the "no effect" line $y = x$ relative to Jaid's range of measured values (the crossing ratio upper bound χ_h is computed similarly but using the upper bounds a_h and b_h of a's and b's 95% probability intervals). A large crossing ratio means that Restore is better than Jaid only on "hard" faults, whereas a small crossing ratio means that Restore is consistently better across the experimented range, as illustrated in the example of Figure 4.2.

Summarizing data with linear regression. Using linear regression to model data that doesn't "look" linear may seem unsound. However, it is not a problem in our case given how we use linear regression: not to *predict* the performance of Restore on yet to be seen inputs, but simply to *summarize* the experimental data in a way that accounts for some measurement errors (and hence is more robust than just summarizing the raw data). After all, the essence of linear regression is a mechanism to "learn about the mean and variance of some measurement, using an additive combination of other measurements" [69], which is all we use it for in analyzing our experimental data.

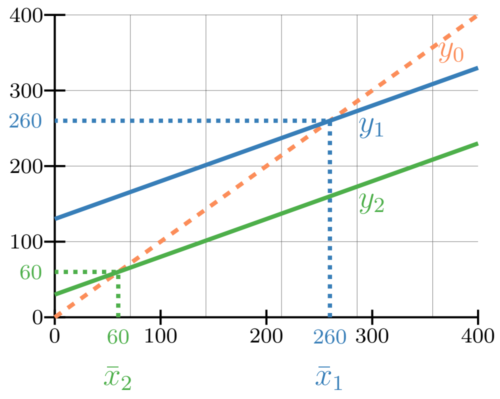


Figure 4.2: Visual explanation of linear regression lines. The two regression lines $y_1 = 130 + 0.5\,x$ and $y_2 = 30 + 0.5\,y$ have the same slope but different intercepts. Therefore, y_2 crosses the "no effect" line $y_0 = x$ at $\bar{x}_2 = 60$, much earlier than y_1 that crosses it at $\bar{x}_1 = 260$. The crossing ratio scales the crossing coordinates $\bar{x}_1$ and $\bar{x}_2$ over the range of values on the x axis. If the range is the whole x axis from 0 to 400, the crossing ratios are simply $\chi_1 = \bar{x}_1/400 = 0.15$ and $\chi_2 = \bar{x}_2/400 = 0.65$, which indicate that y_1 is above y_0 for only 15% of the data, and y_2 for 65% of the data.

4.3.2 Experimental Results

RQ1: Effectiveness

RQ1 assesses the *effectiveness* of RESTORE in terms of the *valid* and *correct* fixes it can generate.

Since most automated program repair tools for Java have been evaluated on the same DEFECTS4J bugs as RESTORE, we can compare *precision* and *recall* of the various tools in Table 4.1.[5] RESTORE and JAID [6] can output multiple, ranked valid fixes for the same bugs; in contrast, other tools often stop after producing one valid fix. We keep this discrepancy into account in Table 4.1 by reporting different values of precision and recall according to whether we consider all valid fixes, only those in

[5]Since these experimental all refer to the same set of bugs (without cross-validation), precision and recall have a narrower scope as effectiveness metrics here than they have in the context of information retrieval.

[6]RESTORE is built on top of JAID with the version published on [9], therefore we refer to the same version of JAID's experiment data in this chapter, rather than the newest result resented in Section 3.3 and [10].

Table 4.1: A quantitative comparison of RESTORE with 13 other tools for automated program repair on DEFECTS4J bugs, based on their published experimental evaluations (see Table 3.4 for sources of the comparison data). For each program repair TOOL, the table references the source of its experimental evaluation data reported here: the number of bugs that the tool could fix with a VALID fix; the number of bugs that the tool could fix with a CORRECT fix; and the resulting PRECISION (CORRECT/VALID) and RECALL (CORRECT/357, where 357 is the total number of DEFECTS4J faults used in the experiments). For tools whose data about the POSITION of fixes in the output ranking is available, the table breaks down the data separately for fixes ranked in ANY POSITION, in the FIRST POSITIONS, and in the TOP-10 POSITION. (These measures do not change for tools that output at most one fix per fault.) The rightmost column UNIQUE lists the number of distinct bugs that *only* the tool can correctly fix. Question marks represent data not available for a tool.

TOOL	VALID	ANY POSITION			FIRST POSITION			TOP-10 POSITION			UNIQUE
		CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	
RESTORE	98	41	42%	11%	19	20%	5%	29	30%	8%	8
ACS	23	18	78%	5%	18	78%	5%	18	78%	5%	12
CapGen	25	22	88%	6%	21	84%	6%	22	88%	6%	3
Elixir	41	26	63%	7%	26	63%	7%	26	63%	7%	0
HDA	?	23	?	6%	13	?	4%	23	?	6%	3
JAID	31	25	81%	7%	9	29%	3%	15	48%	4%	1
jGenProg	27	5	19%	1%	5	19%	1%	5	19%	1%	1
jKali	22	1	5%	0%	1	5%	0%	1	5%	0%	0
Nopol	35	5	14%	1%	5	14%	1%	5	14%	1%	2
SimFix	56	34	61%	10%	34	61%	10%	34	61%	10%	12
SketchFix	26	19	73%	5%	9	35%	3%	?	?	?	0
SketchFixPP	?	34	?	10%	?	?	?	?	?	?	2
ssFix	60	20	33%	6%	20	33%	6%	20	33%	6%	1
xPar	?	4	?	1%	?	?	?	4	?	1%	0

the top-10 positions, or only those produced in the top position (the first produced).

Valid fixes. RESTORE produced at least one valid fix for 97 faults in DEFECTS-4J. As shown in Table 4.1, that is more than any other automated repair tools for Java.

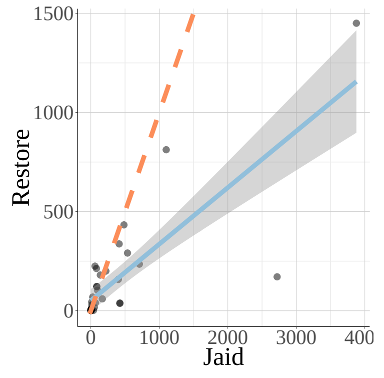
On the 36 faults that JAID can also handle, RESTORE often produces *fewer valid fixes* than JAID: overall, RESTORE produces 56% (1 − 0.44) fewer valid fixes than JAID; and produces more valid fixes for only 13 faults. As we'll see later, RESTORE also produces *more* correct fixes than JAID; thus, fewer valid fixes per bug can be read as an advantage in these circumstances.

Correct fixes. Restore produced at least one correct fix for 41 faults in Defects4J—when considering all fixes for the same bug. As shown in Table 4.1, that is more than any of the other automated repair tools for Java, and constitutes a 21% increase (7 faults) over the runners-up SimFix and SketchFix according to this metric. Restore correctly fixed 8 faults that *no other tool* can currently fix, in addition to the 6 faults that only Restore and Jaid can fix. This indicates that Restore's fix space is somewhat *complementary* to other repair tools for Java.

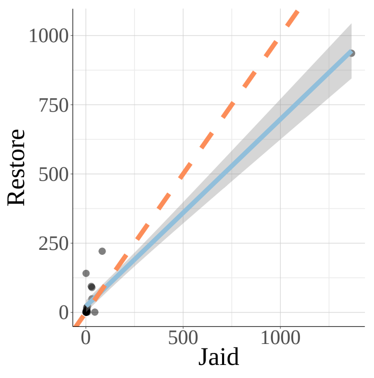
The output list of valid fixes should ideally rank correct fixes *as high as possible*—so that a user combing through the list would only have to peruse a limited number of fix suggestions. For the 23 faults that both Restore and Jaid correctly fix, the two tools behave similarly on the majority of bugs: Restore ranks the first correct fix 1 position higher than Jaid on average; and ranks it lower in 11 faults. Even thought this difference between the two tools is limited, Restore still fixes 18 more bugs than Jaid, and ranks first 8 of them. In addition, Figure 4.3b suggests that Restore's advantage over Jaid emerges with "harder" faults with many valid fixes—where a reliable ranking is more important for practical usability.

Precision. While it can correctly fix more bugs, Restore has a *precision* that is lower than other repair tools. In designing Restore we primarily aimed at extending the fix space that can be explored effectively by leveraging retrospective fault localization; since there is a trade off between explorable fix space and precision, the latter is not as high as in other tools that targeted it as a primary goal.

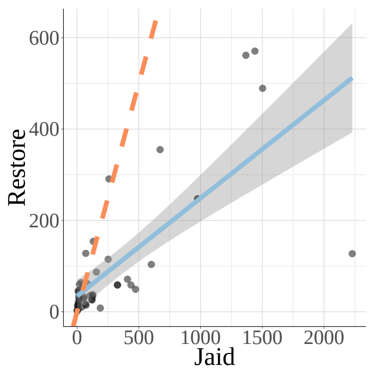
Extended fix space. Restore explores a larger fix space than Jaid, since it can also use expressions outside method `m2f` in the same class to build fixes (Section 4.2.7). In all experiments when Restore could produce valid fixes, 68,344 candidate fixes produced during final fix generation belong to the extended fix space (and hence cannot be produced by Jaid). Among them, 2,049 candidates are valid



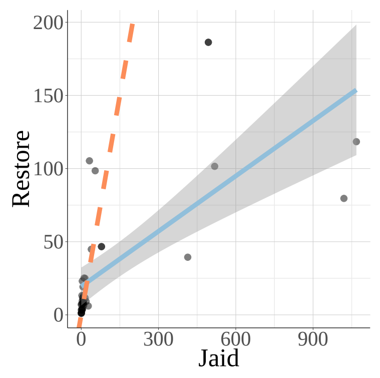
(a) #v: number of valid



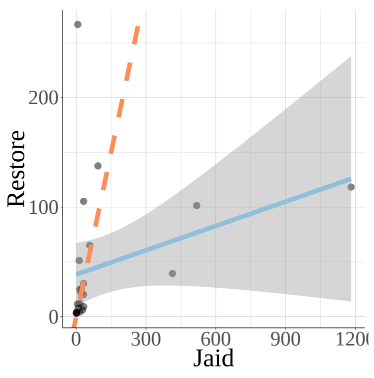
(b) c: rank of correct



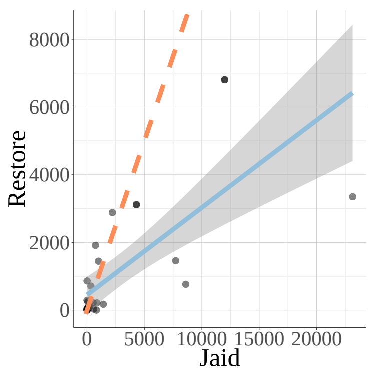
(c) t: total time



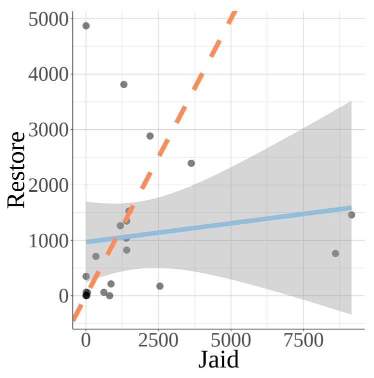
(d) t2v: time to valid



(e) t2c: time to correct



(f) c2v: checked to valid



(g) c2c: checked to correct

Figure 4.3: Comparison of Jaid and Restore on various measures. For each measure m, a point with coordinates $x = J_{m,k}, y = R_{m,k}$ indicates that Jaid costed $J_{m,k}$ of m on fault k while Restore costed $R_{m,k}$ of m on fault k. The dashed line is $y = x$; the solid line is the linear regression with y dependent on x.

(corresponding to 52 faults); and 9 are correct (one for each of 9 faults). In all, the extended fix space enabled Restore to generate valid fixes for 17 more bugs than

JAID, correct fixes for 9 more bugs than JAID; and correct fixes for 5 of the 8 bugs that only RESTORE can correctly fix among all tools (Table 4.1).

Multi-line fixes. Four of the bugs correctly fixed by RESTORE (*Closure40*, *Closure46*, *Closure115*, and *Closure128*) have programmer-written fixes in DEFECTS-4J that change *multiple lines*. For example, project developers fixed the buggy method of bug *Closure128*:

```
1 static boolean isSimpleNumber(String s) {
2   int len = s.length();
3   for (int index = 0; index < len; index++) {
4     char c = s.charAt(index);
5     if (c < '0' || c > '9') return false;
6   }
7   return len > 0 && s.charAt(0) != '0';
8 }
```

by adding `if (len == 0) return false;` before line 3 *and* changing line 7 to `return len == 1 || s.charAt(0) != '0';`. RESTORE, instead, just changed line 7 to

```
if (len == 1) return true;
else return len > 0 && s.charAt(0) != '0';
```

RESTORE's conditional return is equivalent to the program-mer-written fix even though it only modifies one location. Such complex fixes demonstrate how RESTORE manages to combine bug-fixing effectiveness and competitive performance: this fix was the first valid fix in the output, generated in less than 10 minutes.

RESTORE *can correctly fix 41 faults in* DEFECTS4J *when allowing multiple fixes for the same bug; 19 of these faults are fixed by the first fix output by* RESTORE. RESTORE *trades off a lower precision for a larger fix space, which includes correct fixes for 8 faults that no other tools can fix.*

RQ2: Performance

RQ2 assesses the *performance* of RESTORE in terms of its running time.

Total time. RESTORE's wall-clock total running time per fault ranged between 1.5 minutes and 21 hours, with a median of 53 minutes. This means that RESTORE achieves a speedup of 3.1 (1/0.32) over Jaid; Figure 4.3c indicates that the major difference in favor of Restore is particularly marked for the *harder* faults—which generally require long running times.

Comparing with other tools in terms of running time would require to replicate their evaluations using uniform experimental settings—something we did not do in this experimental evaluation. Nevertheless, it is plausible other tools have an overall significant running time too: HDA, ACS, ssFix, Elixir, CapGen, and SimFix are all based on mining external code to learn common features of correct fixes; this process is likely time consuming—even though it would be amortized over a consequent long run of the tools—but is not present in Restore (or Jaid). This indicates that Restore's performance is likely to remain competitive overall, and that retrospective fault localization can bring a performance boon. Performing more fine-grained experimental comparisons belongs to future work.

Time to valid/correct. Especially important for a repair tool's practical usability is the *time elapsing until* a fix appears in the *output*. All else being equal, shorter times mean that users can start inspecting fix suggestions earlier—possibly supporting a more interactive usage—so that the whole repair process can be sped up. On average, Restore outputs the first *valid* fix 83 minutes before Jaid—a 3.4 speedup (1/0.29) according to the linear regression line; and the first *correct* fix 64 minutes before Jaid—a 2.3 speedup (1/0.43). While Figure 4.3d and Figure 4.3e suggest that these averages summarize a behavior that varies significantly with some faults, it is clear that Restore's is *substantially faster* in many cases—especially with the "harder" faults that require long absolute running times. Cutting the running times in less than half on average in these cases results in speedups that often span one order of magnitude, and sometimes even two orders of magnitudes.

Restore's performance is the combined result of exploring a larger fix space than Jaid (which takes more time) and using retrospective fault localization (which speeds up fault localization). That RESTORE finds many more correct fixes while simultaneously often drastically decreasing the running times indicates that its fault localization techniques bring a decidedly positive impact with no major downsides.

RESTORE *is usually much faster than* JAID *even though it explores a larger fix space: 3.1 speedup in total running time; 3.4 speedup in time to the first valid fix; 2.3 speedup in time to the first correct fix.*

RQ3: Fault Localization

Retrospective fault localization is RESTORE's key contribution: a novel fault localization technique that naturally integrates into generate-and-validate program repair algorithms. RQ1 and RQ2 ascertained that retrospective fault localization indirectly improves program repair by supporting searching a larger fix space while simultaneously improving performance. In RQ3 we look into how retrospective fault localization is *directly* more efficient.

Checked to valid/correct. To this end, we follow [66]'s survey of fault localization in automated program repair and compare the number of fixes that are *checked* (generated and validated) until the first *valid* (C2V, called NFC in [66]) and the first *correct* (C2C) fix is generated. The smaller these measures the more efficiently fault localization drives the search for a valid or correct fix.

RESTORE needs to check 57% fewer $(1 - 0.43)$ fixes than JAID until it finds the first valid fix. RESTORE significantly improves measure C2C too: it needs to check 36% $(1 - 0.64)$ fewer fixes than JAID until it finds the first correct fix. Even though JAID is more efficient on some faults, Figure 4.3f and Figure 4.3g show that RESTORE prevails in the clear majority of cases, as well as in the harder cases that require to check many more candidate fixes (exploring a larger search space); the difference is clearly statistically significant (slope under 0.4 with 95% confidence, and the overlap of regression line and "no effect" line is only for small absolute values of C2V and

C2C, as also reflected by the crossing ratio). These results are direct evidence of retrospective fault localization's greater precision in searching for fault causes.

Candidate fixes as mutations. Retrospective fault localization treats candidate fixes as mutants. As described in Section 4.2.5, a candidate that passes at least one previously failing test (during partial validation) increases the suspiciousness ranking of all snapshots associated with the candidate's location. Such candidate fixes sharpen fault localization, and hence we call them *sharpening* candidates. If a sharpening candidate is furthermore associated with a location where a correct fix can be built (according to the correct fixes actually produced in the experiments or in DEFECTS4J) we call it *plausible*.

Table 4.2 measures sharpening and plausible candidates in different categories. Only 2% of all candidates are sharpening; however, the percentage grows to 9% for faults RESTORE can build a valid fix for; and to 12% for faults RESTORE can build a correct fix for. These cases are those where retrospective fault localization achieved progress; in some cases (*plausible* candidates) it even led to finding program locations where a correct fix can be built. Table 4.2 also shows that sharpening and plausible candidates are 9% for faults with a single failing test case in DEFECTS4J. These can be considered "hard" faults because of the limited information about faulty behavior; retrospective fault localization can perform well even in these conditions.

Table 4.3 looks at RESTORE's fault localization feedback loop, which is repeated until retrospective fault localization has successfully refined the suspiciousness ranking. While some faults require as many as ten iterations, in most cases only one iteration is needed to achieve progress. This suggests that candidate fixes are often "good mutants" to perform fault localization—and they provide information that is complementary to that available with simpler spectrum-based techniques.

Table 4.2: How retrospective fault localization achieves progress. Each row focuses on faults in one category: those that RESTORE can repair with a CORRECT fix; with a VALID fix; ALL faults in DEFECTS4J; and those with a SINGLE failing test. In each category, the table reports how many faults are in total (#); for how many RESTORE's fault localization can find a location suitable to build a correct fix (LOCALIZED, either because RESTORE actually built a correct fix or because the DEFECTS4J reference fix modifies that location); the number of CANDIDATES used as mutants in retrospective fault localization; how many of these candidates are SHARPENING and PLAUSIBLE.

	#	LOCALIZED	CANDIDATES	SHARPENING	PLAUSIBLE
CORRECT	41	41	23,529	2,582	511
VALID	98	75	84,989	7,348	2,762
ALL	357	107	495,359	9,854	3,377
SINGLE	74	57	61,530	5,307	2,108

Table 4.3: How many times retrospective fault localization iterates. Among all faults in DEFECTS4J that RESTORE could repair with a VALID or a CORRECT fix, how many ITERATIONS RESTORE's feedback loop went through to sharpen fault localization.

	ITERATIONS									
	1	2	3	4	5	6	7	8	9	10
VALID	86	3	0	0	3	1	2	0	1	2
CORRECT	35	2	0	0	1	1	1	0	1	0

RESTORE*'s retrospective fault localization improves the efficiency of the search for correct fixes: on average, 57% fewer fixes need to be generated and checked until a valid one is found. The candidate fixes generated by* RESTORE *are effective as mutants to perform fault localization.*

4.4 Summary

We presented *retrospective fault localization*: a novel fault localization technique that integrates into the standard generate-and-validate process followed by numerous automated program repair techniques. By executing a form of mutation-based testing using byproducts of automated repair, retrospective fault localization delivers accurate fault localization information while curtailing the otherwise demanding costs of running mutation-based testing. Our experiments compared RESTORE—implementing retrospective fault localization — with 13 other state-of-the-art Java program repair tools—including JAID, upon which RESTORE's implementation is built. They showed that RESTORE is a state-of-the-art program repair tool that can search a large fix space—correctly fixing 41 faults from the DEFECTS4J benchmark, 8 that no other tool can fix—with drastically improved performance (speedup over 3, and candidates that have to be checked cut in half).

Chapter 5

Directing Patch Search with Repeated Learning - PRIDE

Both JAID and RESTORE are search-based APR: They first generate candidate fixes from a large search space and then validate them using the given test cases. As revealed by Long et al. [45] and our previous experimental evaluation, correct fixes are sparse in the huge search space of search-based techniques. Therefore, although larger fix spaces typically contain more correct fixes for more faults, the effectiveness and efficiency of a repair tool may even drop dramatically if an increase to its fix space is not accompanied by corresponding enhancements to its search algorithm. This scenario leads to an inevitable problem when improving the capability of APR: how to direct the search so that it can efficiently find fixes most likely to be correct.

Almost all search-based APR techniques rely on some form of patterns, either automatically learned or manually identified, from programmer-written fixes in the past in constructing candidate fixes. Some also utilize the frequency information about those patterns to generate and/or validate candidate fixes in particular orders so that fixes that are more likely to be correct are processed earlier. While such generic information about fix patterns is clearly critical for the success of program repair, it alone may not be enough to guarantee an effective and efficient fixing process. After all, how a bug should be fixed is ultimately determined by the nature of the bug and the code context where it occurs, and the generic patterns can be instantiated using many different program entities and in many different ways.

One way to factor the bug-specific information into the repair process is by incorporating a feedback loop between fix validation and fix generation. For example, GenProg [82, 80] and HDA [34] evolves candidate fixes to pass all the tests in iterations, where fixes that can pass more tests are more likely used as the basis to derive new fixes. While these techniques enable the overall fixing process to benefit from intermediate fixing results and have helped program repair produce exciting results, they suffer from two major limitations. First, although a candidate fix that can make more tests pass may be closer to being correct, always reusing the fix as a whole can be too coarse-grained and counter-productive, since most likely those fixes also contain parts that correct fixes should not have. Second, it is not clear how the successful fixing experience with such a technique on one fault may help increase the chance of success on other faults.

To overcome the limitations, we present in this chapter the PRIDE (Program Repair wIth repeateD lEarning) technique for improving the effectiveness and efficiency of automated program repair. The key novelty of PRIDE is twofold. First, it defines a rich set of features that not only characterize the composition of fixes but also relate the components of fixes to their context code and test executions. Second, it introduces a learning-to-rank model to enable program repair to *repeatedly* learn valuable knowledge from both completed and ongoing fixing processes about the constitution of high quality fixes. The model is trained on data about candidate fixes from past fixing processes and updated when data about candidate fixes processed in new fixing sessions become available.

We have implemented the PRIDE technique into a tool with the same name based on the RESTORE search-based APR tool. PRIDE takes as the input a faulty Java program, a set of test cases where at least one fails due to the bug under fixing, and a fix ranking model that incorporates the knowledge learned from previous fixing processes, and it outputs a list of fixes to the program that can make all the tests pass and an updated fix ranking model that can be used as input to drive future program repair with PRIDE.

To evaluate the effectiveness and efficiency of PRIDE, we applied it to repair 983 bugs from 4 popular benchmarks in APR, namely DEFECTS4J, INTROCLASSJAVA, QUIXBUGS, and BEARS. PRIDE proposed valid fixes to 243 bugs and correct fixes to 137 bugs, significantly outperforming RESTORE and the other existing APR techniques for Java programs. To demonstrate the generality of the PRIDE technique, we also incorporated it into the SIMFIX APR technique and produced SIMFIX∗. Compared with SIMFIX, SIMFIX∗ proposed correct fixes to one more fault in repairing DEFECTS4J bugs and achieved 3.3 speedup in terms of repair time needed to find the first correct fix.

The rest of this chapter is organized as the following. Section 5.1 illustrates how PRIDE automatically suggests high quality fixes to real-world bugs from users' perspective. Section 5.2 describes in detail the steps PRIDE goes through when fixing bugs and how the steps are connected. Section 5.3 discusses the experiments we conducted to evaluate PRIDE and the results. Section 5.4 summarizes this chapter.

5.1 An Example of PRIDE in Action

Apache Commons Math is a lightweight, self-contained Java library that provides mathematics and statistics components to address the most common problems in the Java programming language. Class `Complex` in the library was designed to model complex numbers, i.e., numbers with real and imaginary parts. Method `add` of the class takes another complex number `rhs` as the parameter and should return the sum of `this` complex number and `rhs`.

```java
public Complex add(Complex rhs) throws NullArgumentException {
  MathUtils.checkNotNull(rhs);
  return createComplex(real + rhs.getReal(), imaginary + rhs.getImaginary());
}
```

Listing 5.1: Faulty method `add` from class `Complex` in project *Commons-Math*.

```java
if(isNaN || rhs.isNaN){
  return NaN;
}
```

Listing 5.2: Fix written by developers (to be inserted before line 3 in Listing 5.1).

```java
if(rhs.isNaN() || this.isNaN()){
  return NaN;
}
```

Listing 5.3: Correct fix generated by PRIDE (to be inserted before line 3 in Listing 5.1).

Listing 4.1 shows the complete implementation of the method. The method first checks if the parameter is a null reference. If yes, a `NullArgumentException` is thrown. Otherwise, the real and imaginary parts of the two complex numbers `this` and `rhs` are added together and the result values are used to create a new complex number as the return object. This simple implementation, however, is defective[1]: A Complex object is *not a number*, or `NaN`, if its field `NaN` is set to true, and the class header comment says that all mathematical operations should return `NaN` if either of the operand complexes is `NaN`, but the implementation of method `add` missed such condition.

To correct the bug, programmers inserted an if statement, as shown in List 5.2, to method `add`. PRIDE managed to produce a *correct* fix for the bug, shown in Listing 5.3, in just around 5 minutes. The generated fix is in a different form than the human-written one but has equivalent semantics.

Quite a few existing APR tools can produce a correct fix to this fault, and they achieve that by exploiting knowledge learned before fixing to guide fix generation [25,

[1]Fault *Math53* in DEFECTS4J [27]

```
if(!this.isNaN()){
  return NaN;
}
```

a)

```
if(rhs.isNaN()){
  return NaN;
}
```

b)

Figure 5.1: Two sharpening fixes generated by Pride (to be inserted before line 3 in Listing 5.1).

34, 83]. Unlike those APR tools, Pride also utilizes information gathered from the current fixing process to make the search for correct fixes more effective and efficient. Particularly, Pride goes through several iterations of fix generation and validation in fixing this fault. In the first iteration, Pride identified two sharpening fixes a) and b) shown in Figure 5.1. Although neither of the two fixes is correct, they hinted at both the location and the component expressions of the correct fix. Based on that information, Pride was able to rank the fix shown in Listing 5.3 on the top of candidate fixes to be validated in the second iteration. The fix turned out to be valid, since it made all the input tests pass, and manual inspection of the fix reveals later that it is also correct.

5.2 How Pride Works

5.2.1 Overview

Figure 5.2 gives an overview of program repair with Pride. Given a faulty Java program to repair, the other inputs required to run Pride include a set of test cases for the program, where at least one is failing and reveals the fault under fixing, and a fix ranking model encoding knowledge learned from previous fixing processes about fixes. Outputs from a successful execution of Pride include a (possibly empty) list of valid fixes that can make all the test cases pass and an updated fix ranking model incorporating knowledge learned from the current fixing process. The updated model can be used as part of the input to future runs of Pride.

During fixing, Pride first applies fault localization to identify a list of program

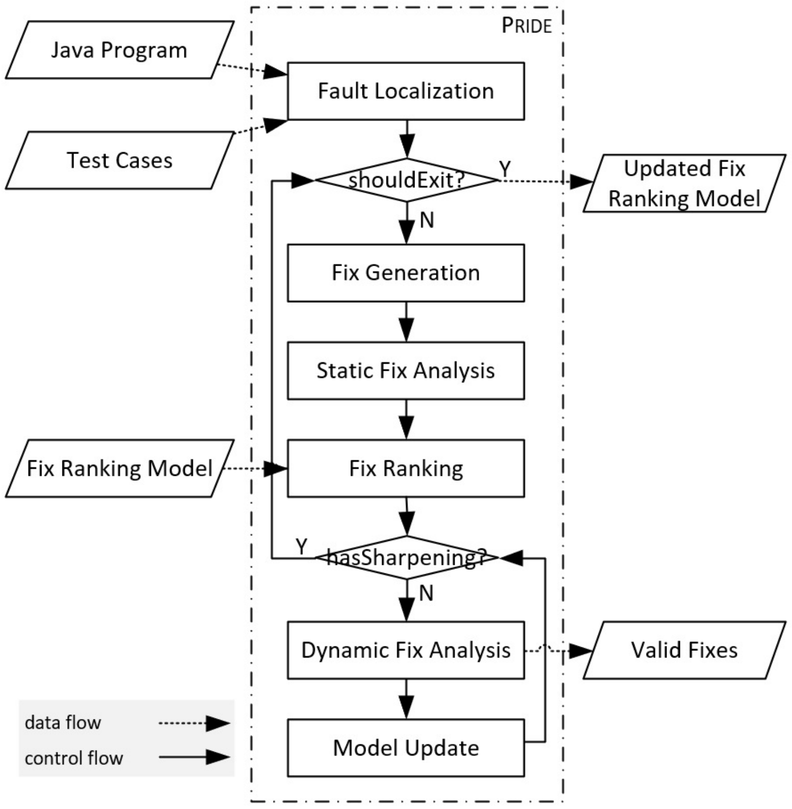


Figure 5.2: Overview of PRIDE.

entities that may be the cause of the fault under repairing, and then enters an iterative process, where candidate fixes are generated and analyzed, and information gathered from intermediate fixing results is utilized to update the fix ranking model and to steer the fixing process. Particularly, in each iteration, PRIDE generates a small number of candidate fixes, statically analyzes those fixes, sorts the fixes using the fix ranking model, and then dynamically analyzes the fixes in the ranking order. If candidate fixes that may help sharpen our understanding about how correct fixes should be like are discovered during dynamic analysis, PRIDE updates the fix ranking model to incorporate the knowledge and enters the next iteration. By repeatedly learning from the past and on-going fixing processes, PRIDE tunes its search for correct fixes to become more effective and efficient.

In the rest of this section, we describe the individual steps in program repair with

PRIDE and explain how they are connected to produce high quality fixes efficiently.

Note that, PRIDE has been designed to loosely couple with specific fault localization and/or fix generation strategies, so that fault localization and fix generation in most existing G&V APR techniques can be easily integrated with PRIDE. In such cases, the other techniques are referred to as the base APR techniques. This design enables PRIDE to help make a wide range of G&V APR techniques become more effective and efficient in navigating their fix spaces.

5.2.2 Fault Localization

A common initial step in most existing G&V APR techniques is fault localization, which serves to identify a list of program entities, i.e., statements, expressions, etc., that may be the cause of the fault under repairing. Similarly, PRIDE first invokes the fault localization process of its base APR technique to produce a list $\boldsymbol{L}$ of $\boldsymbol{P}$'s entities sorted in decreasing order of their likelihood of being faulty. The fault localization result provides initial clues about which fixes should be considered in repairing and in which order.

5.2.3 Fix Generation

To make it easy for PRIDE to adopt existing fix generation techniques, we use a general model to abstract the candidate fixes that PRIDE handles. More concretely, PRIDE models a candidate fix f as a triple $\langle l, s_o, s_n\rangle$, where l is the location at which the fix should be applied, s_o is a sequence of faulty statements at location l of $\boldsymbol{P}$, while s_n is the sequence of statements with which s_o should be replaced when the fix is applied. That is, a fix $f = \langle l, s_o, s_n\rangle$ will replace s_o at l with s_n. Specially, the fix inserts s_n at location l when s_o is an empty sequence, and it deletes s_o at l when s_n is empty. We use $\boldsymbol{F}$ to denote the set of all candidate fixes a base APR technique is able to generate in fixing faulty program $\boldsymbol{P}$.

Note that, although such modelling enables PRIDE to handle fixes most existing G&V APR techniques can generate, it does prevent PRIDE from being able to process larger fixes, e.g., the multi-hunk fixes proposed by Hercules [71]. We leave the generalization of PRIDE to handle fixes affecting multiple methods for future work.

5.2.4 Static and Dynamic Fix Analysis

In fix analysis, PRIDE gathers static and dynamic information about the generated fixes as well as their relationships with the corresponding contexts and the other fixes.

Given a fix $f = \langle l, s_o, s_n \rangle$, the actual changes introduced by f could be much smaller than s_o or s_n. For example, if a fix just negates the condition of an if statement, s_o and s_n would need to include the original and modified if statement, which contain many program elements that are not part of the actual change. In view of that, PRIDE first employs the gumtree-spoon-ast-diff (GSAD) tool [17] to gain a finer-grained understanding of the differences between s_o and s_n, and then extracts the features of f based on the differences.

Given two abstract syntax trees (ASTs) T_1 and T_2 as the input, GSAD constructs an edit script which, when applied, can transform T_1 into T_2. GSAD considers four types of edit actions to AST nodes when transforming T_1, namely to ADD a node, to DELETE a node, to UPDATE the value of a node, and to MOVE a node to a new location, and each action in the edit script is associated with an AST node on T_1 and an action type. Since any change to a child AST node also constitutes a change to the parent AST node, actions in an edit script can be organized into a tree-like structure.

To get a finer-grained understanding of the differences between s_o and s_n, PRIDE first turns s_o and s_n into two block statements by putting each of them into a pair of curly braces, and then feeds the ASTs T_o and T_n for the two block statements into GSAD to produce the edit script needed to transform the former to the latter. Next, PRIDE performs a depth-first traversal on the edit action tree to gather ADD and UPDATE actions to statement nodes and top-level expression nodes. Here, top-level expressions are expressions whose parent nodes are not expressions. For example, the condition expression of an `if` statement is a top-level expression, since its parent is the containing `if` statement. PRIDE ignores DELETE and MOVE actions during the traversal because, according to our experience, they rarely occur in correct fixes, and it ignores all edit actions on subexpressions mainly because we believe all components of an expression are closely related and should be treated as a whole. For each ADD or UPDATE action on a top-level expression, PRIDE obtains a pair $\langle e, e' \rangle$ of expressions, where e is the original expression before the change (in the case of an ADD action, e is `null`) and e' is the result expression after the change; We refer to e as the *source*, and e' as the *destination*, expression of the edit action, respectively. For each ADD or UPDATE action on a statement, PRIDE generates a descriptor in the form "ACTTYPE-NODTYPE" for the action, where "ACTTYPE" is the type of the edit action and "NODTYPE" is the type of the statement.

To model how variables (i.e., local variables, parameters, and class fields) are typically utilized together in l's containing method m, PRIDE builds the *correlated* relation between variables whose scopes overlap with m via intraprocedural, lightweight static analysis. More concretely, PRIDE regards two variables v_1 and v_2 as being *correlated*, denoted as $v_1 \sim v_2$, if and only if 1) there is a data-dependency relation between them in m or 2) they have been used together in a simple statement or a branching condition inside m.

To help better understand the potential influence of f on m, PRIDE also gathers the set ξ of statements in m that may affect or be affected by changes to s_o. More concretely, PRIDE uses each statement in s_o as the criterion to conduct intraprocedural forward and backward slicing, as was done in [83], to collect the corresponding forward and backward slices, and ξ is calculated as the union of all the slices produced for all statements in s_o.

In summary, Pride gathers the following information about fix f via static analysis. Set $\boldsymbol{E} = \{\langle e_i, e'_i \rangle\}$ contains all pairs of source and destination expressions derived from f's edit actions on top-level expressions; Set $\boldsymbol{D}$ contains all descriptors for f's edit actions on statements; Set $\boldsymbol{E_s} = \{x | \exists y : \langle x, y \rangle \in \boldsymbol{E}\}$ contains all source expressions from $\boldsymbol{E}$; Set $\boldsymbol{E_d} = \{x | \exists y : \langle y, x \rangle \in \boldsymbol{E}\}$ contains all destination expressions from $\boldsymbol{E}$; Set ξ contains all context statements that may affect or be affected by changes to s_o.

During dynamic fix analysis, Pride validates the generated candidate fixes. To validate a candidate fix f, Pride first applies f to the faulty program $\boldsymbol{P}$ to produce its fixed version $\boldsymbol{P_f}$ and then runs all the tests in $\boldsymbol{T}_{\times}$ on $\boldsymbol{P_f}$. If fix f can make an originally passing (or failing) test t fail (or pass), that fix f is said to be *killed* by test t. In particular, if fix f can make at least one failing test from $\boldsymbol{T}_{\times}$ pass, we say the fix is *sharpening*, as it *may* help sharpen our understanding about the fault and how it should be fixed. In case f can make all tests in $\boldsymbol{T}_{\times}$ pass, Pride also runs $\boldsymbol{P_f}$ against the tests from $\boldsymbol{T}_{\checkmark}$ until any of them fails or all have passed. After the validation of f is completed, if f is valid, i.e., it can make all tests in $\boldsymbol{T}$ pass, it will be reported to users directly.

5.2.5 Fix Ranking

It is a common task in G&V APR to rank fixes so that candidate fixes that are more likely to be correct can be generated and validated earlier during fixing. In the case of Pride, it builds a learning-to-rank model to capture the knowledge about candidate fixes from past fixing processes and applies the model to rank the fixes generated in repairing new bugs.

Static and Dynamic Features

PRIDE employs a rich set of features to characterize candidate fixes generated by its base APR technique, and the features can be classified into two broad categories, namely the static and dynamic features, based on whether they are derived from static or dynamic information about the fixes. In this section, we explain in detail the 17 features PRIDE extracts from each candidate fix and we will use the following notations when describing how PRIDE calculates the features.

Given a collection Ψ of program elements, i.e., statements or expressions, we define three functions V, W, and O such that $V(\Psi)$, $W(\Psi)$, and $O(\Psi)$ will return the set of variables accessed, the set of AST node types occurring, and the set of operators and methods invoked, in those elements, respectively. Given a characterization function ϕ that maps each fix $f \in \boldsymbol{F}$ to a set of characteristics that f has, we refer to all characteristics that ϕ could possibly return as ϕ-characteristics. Note that any characterization function ϕ can be naturally extended so that it also maps a collection of fixes to the union of those fixes' ϕ-characteristics. Given a fix f, a fix characterization function ϕ, and a set $\boldsymbol{F_s}$ ($\boldsymbol{F_s} \subseteq \boldsymbol{F}$) of sharpening candidate fixes (see Section 5.2.4), the extent to which f and fixes in $\boldsymbol{F_s}$ share common ϕ-characteristics, denoted as $\Delta_{\boldsymbol{F_s},\phi}(f)$, is calculated as $\Sigma_{z\in\phi(f)}Q(z)/\Sigma_{z\in\phi(\boldsymbol{F_s})}Q(z)$, where $Q(z) = |\{x : x \in \boldsymbol{F_s} \wedge z \in \phi(x)\}|$ counts the number of sharpening fixes with characteristic z.

Static Features PRIDE extracts in total eight static features for each candidate fix. Among those features, three are *context-independent* since they reflect basic information about individual candidate fixes, while the other five are *context-dependent* since they relate individual candidate fixes to their contexts.

Context-independent static features include sCOMP, sVARCORR1, and sVAR-

CORR2, with feature IDs F1, F2, and F3, respectively. Particularly, given fix $f = \langle l, s_o, s_n \rangle$, feature sCOMP characterizes f's complexity, and it is simply calculated as $|O(\boldsymbol{E_d})|$; Feature sVARCORR1 characterizes the correlation between variables accessed in f's source and destination expressions, and it is calculated as $|\{\langle v_1, v_2 \rangle : \langle v_1, v_2 \rangle \in N \wedge v_1 \sim v_2\}|/|N|$, where $N = V(\boldsymbol{E_s}) \times V(\boldsymbol{E_d})$; Feature sVARCORR2 characterizes to what extent variables used in the source and destination expressions of individual edit actions are correlated, and it is calculated as $|\{\langle v_1, v_2 \rangle : \langle v_1, v_2 \rangle \in N' \wedge v_1 \sim v_2\}|/|N'|$, where $N' = \{\langle v_1, v_2 \rangle : v_i \in V(\{e\}) \wedge v_j \in V(\{e'\}) \wedge \langle e, e' \rangle \in \boldsymbol{E}\}$.

Context-dependent static features include sVAROCCU, sASTOCCU, sOPOCCU, sTOKENSIM, and sMISIM, with feature IDs F4 through F8. Particularly, given fix $f = \langle l, s_o, s_n \rangle$, feature sVAROCCU captures the percentage of variables used in $\boldsymbol{E_d}$ that are also used in f's context statements in $\boldsymbol{X}$, and it is calculated as $|V(\boldsymbol{X}) \cap V(\boldsymbol{E_d})|/|V(\boldsymbol{E_d})|$; Feature sASTOCCU captures the percentage of AST node types occurring in $\boldsymbol{E_d}$ that also occur in $\boldsymbol{X}$, and it is calculated as $|W(\boldsymbol{X}) \cap W(\boldsymbol{E_d})|/|W(\boldsymbol{E_d})|$; Feature sOPOCCU captures the percentage of operators and methods invoked in $\boldsymbol{E_d}$ that are also invoked in $\boldsymbol{X}$, and it is calculated as $|O(\boldsymbol{X}) \cap O(\boldsymbol{E_d})|/|O(\boldsymbol{E_d})|$; Feature sTOKENSIM captures to what extent s_n is similar to other code fragments in the current project under fixing, and it is calculated as the maximum similarity value between s_n and any code snippet in the project; PRIDE follows the approach in [88] to determine the similarity value of two code snippets. Feature sMISIM captures to what extend method invocations in $\boldsymbol{E_d}$ are similar to other method invocations in the current project under fixing, and it is calculated as the maximum similarity value between method invocations in $\boldsymbol{E_d}$ and those in other parts of the project. The similarity between two method invocations is determined in the same way as how the similarity between code snippets is determined in calculating feature sTOKENSIM.

Dynamic Features Pride derives in total 9 features for each candidate fix from the dynamic analysis of the fixes, including dLoc1, dLoc2, dVarOccu, dExprOccu, dDescOccu, dExprDescOccu, dReplacing, dReplaced, and dMIOccu, with feature IDs f9 through f17.

Two dynamic features concern how likely a fix location is actually faulty. Given fix $f = \langle l, s_o, s_n \rangle$, feature dLoc1 characterizes the likelihood of l being faulty, and it has as its value l's suspiciousness score produced by fault localization (see Section 5.2.2), while feature dLoc2 reflects how likely location l is faulty based on fix validation results. Previous research on Restore [90] has shown that fix validation results can be used in APR to improve the accuracy of fault localization. In this work, Pride adopts the approach of Restore to improving the accuracy of fault localization in G&V APR. Restore treats each fix as a higher order mutation to the faulty program and interprets fix validation results in a similar way as in mutation-based fault localization [59, 56]. In particular, Restore calculates the suspiciousness score of fix f as $|\boldsymbol{T}_{\text{✗}} \cap \mathit{killed}(f)| / \sqrt{|\boldsymbol{T}_{\text{✗}}| \cdot |\mathit{killed}(f)|}$ [90], where $\mathit{killed}(f) \subseteq \boldsymbol{T}_{\text{✗}}$ is the set of all tests that kill f, and thus $\boldsymbol{T}_{\text{✗}} \cap \mathit{killed}(f)$ are the tests that fail on input program $\boldsymbol{P}$ but pass on $\boldsymbol{P}_f$. Feature dLoc2 has the suspiciousness score produced in this way as its value.

The other six dynamic features reflect to what extent each candidate fix shares common characteristics with sharpening fixes, and the calculation of all these features is based on characterization functions and the Δ notation, which are introduced at the beginning of Section 5.2.5.

Given fix $f = \langle l, s_o, s_n \rangle$, feature dVarOccu concerns the variables accessed by f's destination expressions, and it is calculated as $\Delta_{\boldsymbol{F_s},\alpha}(f)$, where $\alpha(f)$ returns the set $V(\boldsymbol{E_d})$ of variables accessed in f's destination expressions; Feature dExprOccu concerns the subexpressions of f's destination expressions and is calculated as $\Delta_{setF_s,\beta}(f)$, where $\beta(f)$ returns the set of subexpressions involved in $\boldsymbol{E_d}$;

Feature dDescOccu concerns descriptors of f's edit actions and is calculated as $\Delta_{setF_s,\gamma}(f)$, where $\gamma(f)$ returns the set of action descriptors from f's edit script; Feature dExprDescOccu concerns descriptors of f's edit actions at the expression level and is calculated as $\Delta_{setF_s,\rho}(f)$, where $\rho(f)$ returns the set of expression-level action descriptors from f's edit script; Feature dReplaced concerns source expressions involved in fix f and is calculated as $\Delta_{setF_s,\delta}(f)$, where $\delta(f)$ returns the set $\boldsymbol{E_s}$ of source expressions associated with f; Feature dReplacing concerns destination expressions involved in fix f and is calculated as $\Delta_{setF_s,\epsilon}(f)$, where $\epsilon(f)$ returns the set $\boldsymbol{E_d}$ of destination expressions associated with f; Feature dMIOccu concerns method invocations introduced by a fix and is calculated as $\Delta_{\boldsymbol{F_s},\zeta}(f)$, where $\zeta(f)$ returns the set of method invocations contained in the destination expressions $\boldsymbol{E_d}$ of f.

How the dynamic features, except dLoc1 and dLoc2, are calculated implies that, after new sharpening candidate fixes are discovered during validation, Pride needs to gather the related information and update the dynamic features of all the generated candidate fixes. To support the efficient (re)calculation of these features, Pride keeps track of the number of times each characteristic occurs in all sharpening fixes.

Learning-to-Rank

Pride applies learning-to-rank techniques to rank fixes. Learning-to-rank (LTR) is a supervised machine learning technique for solving ranking problems in the field of information retrieval [42]. There are two phases in Pride's application of LTR to ranking candidate fixes: a learning phase and a ranking phase. In the learning phase, Pride extracts features from a group of bugs and candidate fixes with desirability labels as the training data, and builds an optimal ranking model that predicts desirability levels for fixes, w.r.t. the bugs and a given loss function. In the ranking

Table 5.1: Format of the training data.

PRJ	BUG	STATIC FEATURES			DYNAMIC FEATURES			LABEL
		FEATURE$_1$	...	FEATURE$_8$	FEATURE$_9$	...	FEATURE$_{17}$	
P_1	b_1	$v_1^{1,1}$	...	$v_8^{1,1}$	$v_9^{1,1}$	...	$v_{17}^{1,1}$	$y^{1,1}$
P_1	b_2	$v_1^{1,2}$	...	$v_8^{1,2}$	$v_9^{1,2}$	...	$v_{17}^{1,2}$	$y^{1,2}$
P_2	b_1	$v_1^{2,1}$	...	$v_8^{2,1}$	$v_9^{2,1}$	...	$v_{17}^{2,1}$	$y^{2,1}$
P_2	b_2	$v_1^{2,2}$	...	$v_8^{2,2}$	$v_9^{2,2}$	...	$v_{17}^{2,2}$	$y^{2,2}$
				...				

phase, new bugs and candidate fixes are fed to the learned ranking model to produce a ranked list of the fixes in decreasing order of their desirability.

LTR approaches can be categorized into three groups: pointwise approaches, pairwise approaches, and listwise approaches [42]. In [4, 37], pairwise LTR were applied to software fault localization with a loss function that calculates the number of times that valid fixes get ranked before invalid ones while sorting candidate fixes for fix validation. In program repair, since users usually would stop looking at the remaining valid fixes once he/she has found the first correct one, it is more useful to have correct fixes ranked as close to the top as possible. PRIDE therefore adopts listwise learning to rank fixes and uses the *normalized discounted cumulative gain* (NDCG) as the metric. The NDCG is a measure of ranking quality, where relevance values, or gains, of search results are discounted at lower rank positions and the discounted cumulative gain at a position is normalized w.r.t. the maximum possible DCG at that position. Intuitively, the NDCG prefers to have highly relevant items appearing earlier in the result list, which aligns well with the expectations we have for program repair results.

Table 5.1 gives the format of the input data for training the fix ranking model. Each input entry corresponds to one candidate fix generated for a particular bug, identified by the corresponding project id and bug id: The entry contains the values for the 17 features as defined in Section 5.2.5 and is assigned an integer value as the label to indicate the usefulness of the candidate fix in correcting the associated bug. There are in total four different labels: value 10 indicates a correct fix, value 5 indicates a valid but incorrect fix, value 3 indicates an invalid but sharpening fix, while value 0 indicates a fix that is not killed by any failing test. When no sharpening fix has been found in fixing a bug, all the dynamic features of the bug's candidate fixes have value `nan`.

5.2.6 Implementation

We have implemented the Pride technique into a tool with the same name, using the Restore APR tool [90] as a base. Restore is a state-of-the-art G&V APR tool for Java, and here we decide to use Restore as a base APR tool for Pride because it is among the most effective APR tools in repairing Defects4J bugs and its implementation is publicly available[2].

To automatically repair a faulty program, Restore first applies spectrum-based fault localization to rank the program states observed during test executions in decreasing order of their likelihood of being faulty, and then generates and validates candidate fixes in an iterative way. In each iteration, Restore first generates and validates a small batch of new candidate fixes, then uses the information about sharpening fixes discovered during fix validation to fine-tune fault localization, and reorders the remaining candidate fixes based on the refined fault localization results so that fixes that are more likely to be correct are validated earlier in the future.

Similar to Restore, when repairing a faulty program, Pride also starts with fault localization and then it generates and validates candidate fixes in an iterative fashion. A key difference between the two is that, only the location information of

[2] `http://tiny.cc/9xff3y`

sharpening fixes is exploited to reorder fixes in RESTORE, while PRIDE employs the input fix ranking model to prioritize candidate fixes for validation. Particularly, in each iteration, PRIDE first generates 10% more candidate fixes, applies the model to rank the fixes, validates fixes within the top 30% of the ranked list, and updates the ranking model if any sharpening fix is detected during fix validation.

The PRIDE tool employs the WALA[3] libraries to analyze the definitions and uses of variables in Java programs, and it utilizes XGBoost [11]—a widely-used gradient boosting tree implementation—to train the models and rank the candidate fixes, with NDCG being the objective function. XGBoost implements the LambdaMart [6] algorithm to perform listwise learning to rank. Model update in PRIDE is based on the training continuation feature of XGBoost.

5.3 Experimental Evaluation

We experimentally evaluate the effectiveness and efficiency of the PRIDE tool using Java bugs from 4 popular benchmarks.

5.3.1 Experimental Design

Research Questions

The experiments address the following research questions:

RQ1: How effective and efficient is PRIDE in repairing faulty Java programs?

RQ2: How does PRIDE compare with existing APR tools in repairing faulty Java programs?

RQ3: How useful are the individual features in the learning-to-rank model?

RQ4: Is PRIDE generally applicable to G&V APR techniques?

[3]T.J. Watson Libraries for Analysis. `http://wala.sf.net`.

In RQ1, we examine from a user's perspective the usefulness of PRIDE as a G&V APR tool. In RQ2, we directly compare PRIDE to several other state-of-the-art APR tools for Java. Since PRIDE ranks fixes based on a total of 17 static and dynamic features of those fixes, we investigate to what extent these features are useful in RQ3. In RQ4, we look for evidence that PRIDE is applicable also to other APR techniques.

Subjects

Our experiments targeted a total of 983 faults collected from four widely used benchmarks of Java bugs: DEFECTS4J [27], INTROCLASSJAVA [16], QUIXBUGS [39], and BEARS [48]. DEFECTS4J [27] (revision #895c4e6) includes 395 bugs from 6 open source Java projects, namely Chart, Closure, Lang, Math, Time, and Mockito, and these bugs have been routinely used in evaluating various G&V APR techniques in the past. Faulty programs in DEFECTS4J contain 129,592 lines of code on average. INTROCLASSJAVA [16] is a direct Java translation of the IntroClass benchmark [36], which contains 297 buggy student-written solutions to 6 small assignments given in an introductory, undergraduate programming course. Faulty programs in INTROCLASSJAVA have on average 230 lines of code. QUIXBUGS [39] contains 40 faulty programs taken from the Quixey Challenge [39], where programmers had one minute to produce a fix given an implementation of a classic algorithm with a bug on a single line. Faulty programs in QUIXBUGS contain on average 190 lines of code. BEARS [48] contains 251 bugs mined from the repositories of 72 different GitHub projects based on commit building state from Travis Continuous Integration. Faulty programs in BEARS contain on average 62,597 lines of code. The diversity in the nature and complexity of the subject faults ensures that the experiments are representative of PRIDE's behavior in different conditions.

Setup

To answer RQ1, we apply PRIDE to repair all the subject faults and compare the repair results produced by PRIDE with those produced by RESTORE. To prepare the input fix ranking model, we revise PRIDE so that it always validates candidate fixes in their generation order until all candidate fixes are validated or the time limitation set for repairing is exhausted. We refer to this version of PRIDE as PRIDE-. PRIDE- requires no existing fix ranking model as the input. We then apply PRIDE- to repair all the subject faults in DEFECTS4J, manually examine the output valid fixes, label valid fixes that are semantically equivalent to programmer-written fixes provided in the DEFECTS4J benchmark as *correct*, and gather the features of all the fixes, be it invalid, valid, or correct, from the fixing processes. The time limitation for running PRIDE- was set to be 15 hours.

PRIDE relies on fix ranking models to decide which fixes are more likely to be correct. It is therefore important to include data about a good number of correct fixes when training the ranking model. Since the total number of faults that PRIDE- was able to correctly fix is not large, we perform leave-one-out [20], instead of the standard 10-fold, cross validation across the subject faults. That is, for each fault f in DEFECTS4J, we use the data collected from running PRIDE- on all the other DEFECTS4J faults for training a fix ranking model and use the model as the input to PRIDE for repairing f. In this way, we will have more training data about correct fixes for each repair task. While the training of the input fix ranking model for each repair task in the experiments are done separately and take some time, the cost is not necessary when applying PRIDE in practice: Each run of PRIDE, except for the very first one, can simply take the output fix ranking model from the previous run as the input. For time reasons, we do not repeat the same process to gather features from fixing the faults in other benchmarks. Instead, we use all the data collected from running PRIDE- on DEFECTS4J faults for training another fix ranking model and always use this model as the input to PRIDE for repairing all faults in the other 3 benchmarks.

To answer RQ3, we revise PRIDE to produce 17 distinct variants. Compared with PRIDE, each variant uses a fix ranking model with one of the features excluded. Given a fix ranking model that does not include feature with id FID, we refer to the variant of PRIDE using that model as PRIDE-FID. For example, feature SLOC is not used in the model of variant PRIDE-F1. We apply all the variants of PRIDE to repair the DEFECTS4J bugs and compare the results produced.

To support our claim that PRIDE is applicable to program repair techniques other than RESTORE and address RQ4, we implemented the PRIDE technique atop the SimFix [25] automated program repair system. SimFix is another effective G&V APR techniques for Java in fixing DEFECTS4J bugs, and its source code and replication package are also publicly available. To repair a faulty program, SimFix first applies spectrum-based fault-localization to rank statements according to their suspiciousness. For each statement above a certain suspiciousness rank, SimFix searches for "donor code" (code snippets in the same project that are similar to those close to the suspicious statement), extracts modification patterns from the donors, and builds candidate fixes by matching these patterns to the suspicious statement. As soon this process determines one fix that is valid (i.e., passes all available tests), SimFix stops.

We refer to the modified version of SIMFIX by adding PRIDE as SIMFIX*. Compared with SIMFIX, SIMFIX* shares the same fault localization process, but it does not generate and validate candidate fixes strictly in decreasing order of their locations' suspiciousness values. As expected, SIMFIX* generates and validates candidate fixes in an iterative fashion. In each iteration, SIMFIX* generates candidate fixes on 5 more faulty locations, applies the input model to rank the fixes, validates fixes among the top 30% of the ranked list, and updates the ranking model if any sharpening fix is detected during fix validation. In view that SimFix puts much more emphasis on the first valid fix—it terminates once a valid fix is found, we disable fix ranking in SimFix∗ before the first batch of candidate fixes is validated, so that the tool has a chance to bootstrap the learning process. To better understand the impact of Pride on SimFix, we ony terminate SimFix when it has produced 5 valid fixes or used up all the given time, i.e., 5 hours, as in [25], in fixing a bug. We evaluate SimFix∗ on just Defects4J faults and leave a more comprehensive experimental evaluation of SimFix∗ for future work, because SimFix relies on Defects4J commands to compile fixes and run tests, and it is non-trivial to apply the tool to repair faults in other benchmarks.

Each experiment aims at fixing one subject fault, where Pride takes as the input the buggy program, the tests, and a fix ranking model, and outputs a sorted list of valid fixes for the bug and an updated fix ranking model. We set a time out of 6 hours, as was done in [47]. When fixing is over, we manually examine the reported valid fixes in order until the list has been exhausted or a correct fix is found. We consider a fix as correct if it is semantically equivalent with the programmer-written fix to the bug, which is consistent with how correct fixes are identified in other experiments on program repair tools.

We record the following metrics from each experiment: the number of *valid* fixes in the result; the rank of the first *correct* fix in the result; the overall wall-clock fixing time; the fixing time until the first *valid* fix is found; and the fixing time until the first *correct* fix is found.

All the experiments ran on the author's institution's cloud infrastructure. Each experiment used exclusively one virtual machine instance, running Ubuntu 14.04 and Oracle's Java JDK 1.8 on one core of an Intel Xeon Processor E5-2630 v2 with 8 GB of RAM.

Table 5.2: Experimental results of PRIDE and RESTORE in repairing faults from the 4 benchmarks. All times are in minutes.

BENCHMARK	#BUG	#LOC	PRIDE						RESTORE					
			#V	#C	#C@F	#C@10	T2V$_{MED}$	T2C$_{MED}$	#V	#C	#C@F	#C@10	T2V$_{MED}$	T2C$_{MED}$
DEFECTS4J	395	129,592	110	53	28	42	13.5	14.1	98	41	19	29	11.6	21.3
INTROCLASSJ	297	230	95	71	47	57	9.3	10.4	82	68	31	43	4.3	9.7
QUIXBUGS	40	190	14	10	8	9	5.5	5.5	11	9	4	8	8.9	15.1
BEARS	251	62,597	24	3	0	3	71.3	1.6	10	1	0	0	3.2	13.8
Overall	983	–	243	137	83	111	9.9	10.4	201	119	54	80	6.5	14.8

5.3.2 Experimental Results

In this section, we report the experimental results and answer the research questions.

RQ1: Effectiveness and Efficiency

Table 5.2 summarizes the experimental results with PRIDE. For each BENCHMARK, the table gives the numbers of faults from the benchmark for which PRIDE proposed valid (#V) fixes, the numbers of bugs that PRIDE could fix with a correct fix when fixes at any (#C), only the first (#C@F), or the top-10 (#C@10) positions are considered, and the median time PRIDE spent to produce the first valid (T2V) and correct (T2C) fixes. Given that PRIDE uses RESTORE as its base APR technique, the table also lists the same measures achieved by RESTORE on the same faults.

Within the given time limit, PRIDE produced valid and correct fixes to 243 and 137 faults, respectively, significantly outperforming RESTORE, which produced valid and correct fixes to 201 and 119 faults, respectively. A closer look at the faults that were repaired with valid and/or correct fixes reveals that the two tools actually explored different areas of their fix spaces. Particularly, PRIDE failed to produce any valid fixes to 26 faults where RESTORE proposed valid fixes, while RESTORE failed to produce any valid fixes to 58 faults where PRIDE proposed valid fixes. Similarly, PRIDE failed to produce any correct fixes to 19 faults where RESTORE proposed correct fixes, while RESTORE failed to produce any correct fixes to 37 faults where

Pride proposed correct fixes. On the one hand, such result clearly shows that Pride is more effective than Restore in proposing high quality fixes overall. On the other hand, it also indicates that there is plenty of space to further improve the fix ranking mechanism implemented in Pride. Pride not only proposed valid and correct fixes to more faults than Restore, it also produced more correct fixes at the top of the result lists of valid fixes. In particular, Pride can correctly fix 29 and 21 faults than Restore, if we only consider fixes at the first or the top-10 positions, respectively.

The median time elapsed until Pride produced a valid and correct fix is 9.9 and 10.4 minutes, respectively, which is comparable with that of Restore's. Pride's performance is the combined result of exploring a larger fix space than Restore and using a model to rank candidate fixes. The fact that Pride found many more correct fixes while without spending much more time suggests the fix ranking model brings a positive impact with no major downsides.

Histograms in Figure 5.3 shows how the measures achieved by Pride and Restore compare on the same faults. For each measure m, a bar of height h at an x-coordinate range r indicates that the ratio between m's values produced by Pride and Restore was within range r on h bugs. We can observe from the figures that, on faults where both Pride and Restore can propose valid/correct fixes, Pride tends to generate more valid fixes and produce correct fixes not only with better ranks but also in shorter times.

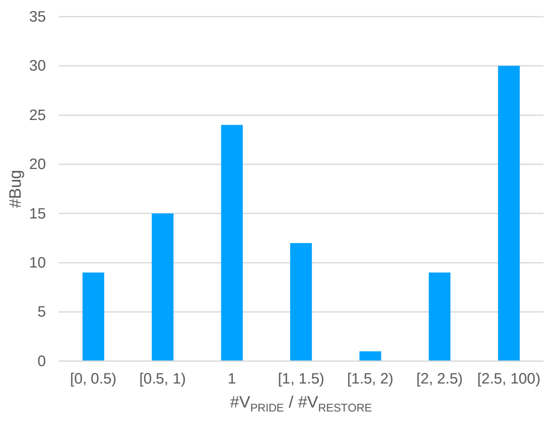
Pride *proposed valid and correct fixes to 243 and 137 bugs, respectively, significantly outperforming* Restore. *For 83 and 111 bugs, the correct fixes proposed by* Pride *were at the first and within the top-10 positions, respectively. The median repairing time with* Pride *was comparable with that of* Restore*'s.*

RQ2: Efficiency

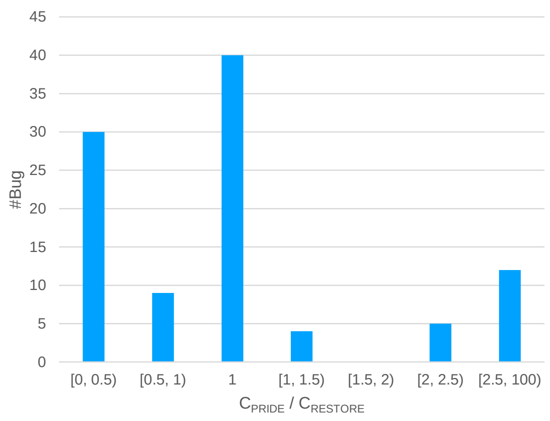
Table 5.3 compares the fixing results of Pride with 17 recent APR tools for Java on bugs from the four benchmarks. For each program repair tool (tool), the table lists

Table 5.3: A quantitative comparison of PRIDE with 17 other tools in repairing bugs from the 4 benchmarks.

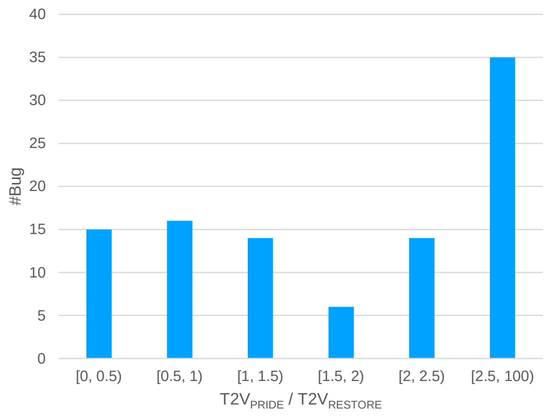
TOOL	#V	ANY			FIRST			TOP-10		
		#C	P	R	#C	P	R	#C	P	R
DEFECTS4J										
PRIDE	110	53	48%	13%	28	25%	7%	42	38%	11%
ACS [89]	23	18	78%	5%	18	78%	5%	18	78%	5%
CapGen [83]	25	22	88%	6%	21	84%	6%	22	88%	6%
CoCoNuT [47]	85	44	52%	11%	44	52%	11%	44	52%	11%
Elixir [70]	41	26	63%	7%	26	63%	7%	26	63%	7%
HDA [34]	?	23	?	6%	13	?	4%	23	?	6%
HERCULES [71]	63	46	73%	12%	46	73%	12%	46	73%	12%
JAID [10]	94	35	37%	9%	14	15%	4%	25	27%	6%
RESTORE [90]	98	41	42%	11%	19	20%	5%	29	30%	8%
SIMFIX [25]	56	34	61%	10%	34	61%	10%	34	61%	10%
SketchFix [23]	26	19	73%	5%	9	35%	3%	?	?	?
SketchFixPP [23]	?	34	?	10%	?	?	?	?	?	?
ssFix [88]	60	20	33%	6%	20	33%	6%	20	33%	6%
TBar [72]	81	42	50%	11%	42	50%	11%	42	50%	11%
INTROCLASSJAVA										
PRIDE	95	71	75%	24%	47	49%	16%	57	60%	19%
JAID	84	69	82%	27%	30	36%	12%	43	51%	17%
CapGen	?	25	?	10%	?	?	?	?	?	?
JFix [32]	?	19	?	7%	?	?	?	?	?	?
RESTORE	82	68	83%	23%	31	38%	10%	43	52%	14%
S3 [33]	?	22	?	9%	?	?	?	?	?	?
QUIXBUGS										
PRIDE	14	10	71%	25%	8	57%	20%	9	64%	23%
CoCoNuT	20	13	65%	30%	13	65%	30%	13	65%	30%
JAID	11	9	82%	23%	4	36%	10%	9	82%	23%
Astor [50]	11	6	55%	15%	?	?	?	?	?	?
Nopol [49, 92]	4	1	25%	3%	?	?	?	?	?	?
RESTORE	11	9	82%	23%	4	36%	10%	8	73%	20%
BEARS										
PRIDE	24	3	13%	1%	0	0	0	3	13%	1%
RESTORE	10	1	10%	0%	0	0%	0%	0	0%	0%



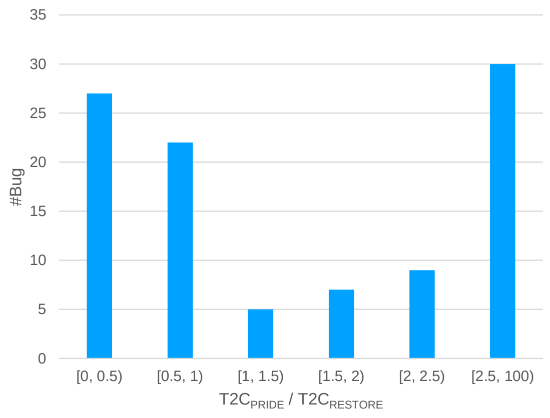
(a) #V



(b) C



(c) T2V



(d) T2C

Figure 5.3: Comparison of PRIDE and RESTORE on various measures.

the number of bugs that the tool could fix with a valid fix (#V), and the number of bugs that the tool could fix with a correct fix (#C) as well as the precision (#C/#V) and recall (#C/#BUG) of the result list (P and R) when fixes at any (ANY), only the first (FIRST), or the top-10 (TOP-10) positions are considered. Here, #BUG denotes the total number of bugs considered when evaluating an APR tool. Question marks in the table denote data that are unavailable for a tool.

A recent study [15] evaluated 11 repair tools for Java on different benchmarks—including INTROCLASSJAVA, QUIXBUGS, and BEARS. Since we compare the effectiveness of repair tools mainly based on their capability to propose correct fixes, while the study only considered the number of valid fixes, we refrain from including the repair tools evaluated in that study in Table 5.3.

PRIDE produced valid fixes to more bugs than any other tool, which suggests that PRIDE explores a larger fix space than other tools do—effective ranking of candidate fixes is therefore more critical for fixing with PRIDE to be successful. On DEFECTS4J bugs, when fixes in all rank positions are considered, PRIDE's recall is 1.8% higher than the first runner-up Hercules (i.e., 7 more bugs correctly fixed) and 2.3% higher than the second runner-up CoCoNuT (i.e., 9 more bugs correctly fixed). When only fixes in the top-10 positions are considered, PRIDE's recall drops to the third place among the tools, following Hercules and CoCoNuT. Note that 15 of the bugs correctly fixed by Hercules are multi-hunk bugs, while CoCoNuT needs the perfect line information about bugs as the input. Given that the fixes PRIDE generates are simple and inspecting 10 valid fixes may be acceptable for users, we believe the overall recall of PRIDE is comparable with state-of-the-art APR tools.

No matter how many fixes in various positions are considered, the precision PRIDE achieved, however, is relatively low when compared with most other tools listed in the table. One thing that the tools with high precision do, while PRIDE does not do, is applying knowledge learned from human-written fixes in code repositories to rank generated fixes. To enhance the precision of the repair results, we will investigate how knowledge learned from previous fixes and from various fixing processes could be incorporated and applied to drive more effective program repair.

PRIDE achieved even higher recall and precision on benchmarks INTROCLASSJAVA and QUIXBUGS than on DEFECTS4J, but it only managed to propose valid and correct fixes to 24 and 3 bugs from BEARS. We leave the understanding of BEARS bugs' unique characteristics and the development of new APR techniques to effectively repair BEARS bugs for future work.

PRIDE produced correct fixes for more bugs than any other APR tools for Java. PRIDE is less precise than some other tools on DEFECTS4J bugs.

RQ3: Feature Usefulness

Table 5.4 lists, for each feature (FID), the numbers of DEFECTS4J bugs that the tool PRIDE-FID could fix with a correct fix (#C) when fixes at any (ANY), only the first (FIRST), or the top-10 (TOP-10) positions are considered.

Table 5.4: Usefulness of individual features.

#C	FID																
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
ANY	53	51	52	51	50	52	53	49	52	51	52	52	50	50	51	51	52
FIRST	24	20	30	21	26	27	27	26	26	24	30	27	22	24	29	27	30
TOP-10	43	37	41	40	39	37	40	37	39	39	44	39	37	39	40	39	43

On the one hand, it is clear from the table that excluding any individual feature from the fix ranking model will cause the effectiveness of PRIDE to drop, which emphasizes the usefulness of the features. On the other hand, the differences caused by the removal of any individual feature in the measure values were not big, which suggests none of the features is dominant in ranking fixes, and therefore the learning-to-rank model is able to provide relatively stable ranking power even when one feature is excluded.

All features in PRIDE are useful for achieving the maximum effectiveness, while no feature is dominant in ranking fixes.

RQ4: Generality

In our experiments, SIMFIX produced correct fixes to 33 DEFECTS4J bugs; SIMFIX∗ produced correct fixes to 34 bugs, including the 33 bugs SIMFIX correctly fixed and an extra one, but with the correct fixes ranked between positions 2 and 4 for three bugs. The scatter chart in Figure 5.4 shows how the repair time until the

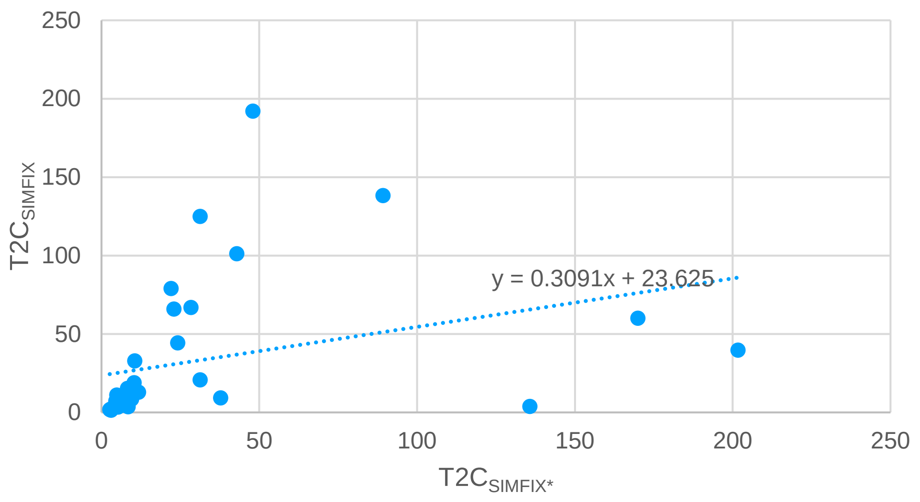


Figure 5.4: Comparison of SimFix∗ and SimFix on T2C.

first correct fix is found compares between Pride and Restore on the 33 faults they both correctly fixed. We build a simple linear regression equation to model the relation between the times with the two tools. The slope and the intercept of the regression line being around 0.3 and 23 suggests that SimFix∗ achieves a speedup of 3.3 (1/0.3) in finding a correct fix and that the amount of time SimFix∗ needs for finding a correct fix is often small.

Such result aligns well with our observation in Section 5.3.2. That is, although Pride can help program repair become more effective and efficient most of the time, it is essentially driven by heuristics and does not always produce the best results. Next, we will first conduct a larger-scale experiment to more thoroughly evaluate the strengths and weaknesses of Pride, and then try to devise new techniques to enable Pride to work in synergy with existing strategies, rather than to use Pride to simply replace existing strategies.

SimFix∗ *correctly fixed one more bug than* SimFix. *The correct fixes proposed by* SimFix∗ *were not ranked at the first positions on 3 faults, but* SimFix∗ *achieved 3.3 speedup in producing correct fixes.*

5.4 Summary

We presented the Pride technique for effective program repair. Compared with existing techniques, Pride is able to learn not only from what has happened in past fixing processes but also from what is happening in the current fixing session, which, we believe, makes Pride more effective and adaptive. Such capability stems from the rich set of static and dynamic features that Pride utilizes to characterize fixes and the ranking model that incorporate rich knowledge about fixes. Pride was able to propose correct fixes to 137 bugs from 4 popular benchmarks.

Chapter 6

Conclusion and Future Work

This chapter summarizes our contributions to automatic program repair as presented in this thesis and outlines potential directions for future research. Section 6.1 recapitulates the major contributions, and Section 6.2 describes future work.

6.1 Main Contributions

In this thesis, we presented three techniques, namely Jaid, Restore, and Pride, to advance the state-of-the-art in search-based automated repair of Java programs.

Jaid builds rich state-based abstractions of faulty programs automatically from plain Java code, and the state abstractions buttress effective fault localization and fix generation and ranking processes. Grounding the repair process on state abstractions mitigates the overfitting problem and improves the patch quality. In our evaluation study involving 693 bugs from the Defects4J, IntroClassJava, and QuixBugs benchmark suites, Jaid produced correct fixes for 113 bugs with a precision of nearly 60%. Jaid is the first APR technique that achieves high levels of precision without relying on additional input other than tests and faulty code.

To enhance the precision of fault localization in APR, we then devise *retrospective fault localization* and implement Restore that integrates the fault localization into the standard generate-and-validate process based on Jaid. By executing a form of mutation-based testing using byproducts of automated repair, retrospective fault localization delivers accurate fault localization information while curtailing the otherwise demanding costs of running mutation-based testing. Our experiments showed that RESTORE is a state-of-the-art program repair tool that can search a large fix space with correctly fixing 41 faults from the 359 bugs of DEFECTS4J benchmark, 8 that no other tool can fix. At the meanwhile, it drastically improved performance that speedup over 3 times, and cut in half candidates that have to be checked.

To further streamline the processes in automated program repair, we propose the PRIDE technique to learn from not only past fixing processes but also the current fixing session. With the rich set of static and dynamic features characterizing fixes and ranking models incorporating fix related knowledge, the learning capability makes PRIDE more effective and adaptive. In our evaluation experiment, PRIDE was able to propose correct fixes to 137 bugs from 4 popular benchmarks, significantly outperforming RESTORE and the other existing APR techniques for Java programs. The experimental results suggest that the learned knowledge about candidate fixes helps to improve the effectiveness and efficiency of program repair.

6.2 Future Work

Our work on JAID, RESTORE, and PRIDE demonstrates the power of state abstraction in facilitating search-based automatic program repair. In the future, we plan to investigate how state abstraction can be utilized to help further reduce the costs for applying APR in practice. For instance, by categorizing equivalent fix ingredients w.r.t. the corresponding Boolean expressions, we should be able to easily identify groups of candidate fixes with the same semantics and then reduce the fixing time by validating just one representative from each group of those candidate fixes.

As discussed in previous chapters, the effectiveness and efficiency of search-based automated program repair can benefit from integrating additional information about plausible repairs. Mining knowledge from software repositories and developer written fixes has already been successfully used in other tools [34, 89] but not yet in ours. Hence, one interesting thing that we plan to do in near future is to investigate how to extract useful information from other sources and factor those information into our approaches in an effective way.

Besides, in view that reference code snippets implementing the same or similar functionalities as the buggy code can provide great guidance as to how the buggy code should be rectified to remove the bug, we will also investigate how natural language processing techniques may be applied to automatically summarize the functionalities of code snippets and how reference code snippets can be effectively identified based on those summaries.

Moreover, our state based APR techniques could naturally combine with dynamic symbolic execution to further increase the efficiency of exploring search space. For example, we could regard the negation of suspicious snapshot as a constrain and use a constrain solver to obtain the concrete value of variables satisfying the constrain. The new variable would cause new states, then maybe we can further explore how to use old program states to help us obtain the correct new states.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, November 2009.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188, New York, NY, USA, 2016.

[5] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 137–146, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Christopher J. C. Burges. From RankNet to LambdaRank to LambdaMART: An overview. Technical report, Microsoft Research, 2010.

[7] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 121–130. ACM, 2011.

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

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

[10] Liushan Chen, Yu Pei, and Carlo Alberto Furia. Contract-based program repair without the contracts: An extended study. *IEEE Transactions on Software Engineering*, 2020.

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[12] D.E. Critchlow. *Metric Methods for Analyzing Partially Ranked Data.* 3Island Press, 1986.

[13] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74, Washington, DC, USA, 2010. IEEE Computer Society.

[14] Thomas Durieux, Benjamin Danglot, Zhongxing Yu, Matias Martinez, Simon Urli, and Martin Monperrus. The patches of the nopol automatic repair system on the bugs of defects4j version 1.1. 0. 2017.

[15] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 302–313, 2019.

[16] Thomas Durieux and Martin Monperrus. Introclassjava: A benchmark of 297 small and buggy java programs. 2016.

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Trans. Software Eng.*, 45(1):34–67, 2019.

[19] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep learning for bug-localization in student programs, 2019.

[20] Jiawei Han and Micheline Kamber. *Data mining : concepts and techniques.* Kaufmann, San Francisco [u.a.], 2005.

[21] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[22] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 464–475, Washington, DC, USA, 2015. IEEE Computer Society.

[23] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 12–23, 2018.

[24] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transaction Software Engineering*, 37(5):649–678, September 2011.

[25] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309, 2018.

[26] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[27] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

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

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

[30] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[31] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590, New York, NY, USA, 2015.

[32] 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 *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 376–379, 2017.

[33] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 593–604, 2017.

[34] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[35] 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 *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[36] 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*, 41(12):1236–1256, 2015.

[37] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):92:1–92:30, October 2017.

[38] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[39] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, 2017.

[40] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE, 2019.

[41] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA, 2019. ACM.

[42] Tie-Yan Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, March 2009.

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

[44] Fan Long and Martin Rinard. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015.

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

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

[47] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[48] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478, Feb 2019.

[49] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

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

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

[52] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 129–139, New York, NY, USA, 2018. ACM.

[53] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 448–458.

[54] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[55] Martin Monperrus. Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 51:1–24, 2017.

[56] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 153–162, Washington, DC, USA, 2014. IEEE Computer Society.

[57] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3):11:1–11:32, August 2011.

[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013.

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

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

[61] 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*, 40(5):427–449, 2014.

[62] Yu Pei, Yi Wei, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Code-based automated program fixing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 392–395. IEEE, 2011.

[63] Jeff H. Perkins, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, Martin Rinard, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, and Stelios Sidiroglou. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, 2009.

[64] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.

[65] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, New York, NY, USA, 2014.

[66] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, New York, NY, USA, 2013. ACM.

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

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

[69] Christian Robert. Statistical rethinking, 2015.

[70] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 648–659, 2017.

[71] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 13–24, Piscataway, NJ, USA, 2019. IEEE Press.

[72] SerVal-DTF. Serval-dtf/tbar. `https://github.com/SerVal-DTF/TBar/tree/master/Results`.

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

[74] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018.

[75] Armando Solar-Lezama. Program sketching. *Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

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

[77] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *International Symposium on Search Based Software Engineering*, pages 99–114, Paderborn, August 2017. York.

[78] Shangwen Wang, Ming Wen, Xiaoguang Mao, and Deheng Yang. Attention please: Consider mockito when evaluating newly proposed automated program repair techniques. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 260–266. 2019.

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

[80] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

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

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

[83] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

[84] W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

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

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

[87] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):573–597, 2009.

[88] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 660–670, 2017.

[89] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. ACM, August 2017.

[90] Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo Alberto Furia. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering*, 2020.

[91] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, 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*, 43(1):34–55, 2016.

[92] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–10. IEEE, 2019.

[93] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

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

[95] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 272–281. ACM, 2006.