DESIGN OF REPAIR OPERATORS FOR AUTOMATED PROGRAM REPAIR

SHIN HWEI TAN

NATIONAL UNIVERSITY OF SINGAPORE

2017

DESIGN OF REPAIR OPERATORS FOR AUTOMATED PROGRAM REPAIR

SHIN HWEI TAN

(M.S., University of Illinois at Urbana-Champaign)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:

Professor Abhik Roychoudhury

Examiners:

Professor David Samuel Rosenblum

Associate Professor Siau Cheng Khoo

Professor Martin Monperrus, KTH Royal Institute of Technology

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Shin Hwei Tan

30 October 2017

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my PhD advisor, Prof. Abhik Roychoudhury for consistently challenging me to excel in research. I would also like to thank my Master advisor, Prof. Darko Marinov for instilling on me a passion for research, which motivates me to pursue my PhD study. I am also thankful to my Master co-advisor, Prof. Lin Tan for her valuable advices and guidance.

I would like to give special thanks to my parents and my siblings for their unconditional love and support throughout my PhD journey.

I would also like to thank my thesis committee members, Prof. David S. Rosenblum, Prof. Siau Cheng Khoo and Prof. Martin Monperrus for their valuable comments and suggestions that help in improving the presentation and the quality of this thesis.

I would like to express my sincere gratitude to Sergey Mechtaev for being such kind friend who support me through several fruitful collaborations and useful suggestions.

I would also like to thank Prof. Jooyong Yi, Dr. Marcel Böhme, and Dr. Zhen Dong for their inspiring discussions and collaborations that leads to several works discussed in this thesis. I am also grateful to Dr. Konstantin Rubinov for sharing with me teaching responsibilities for the software testing module.

I would also like to thank Dr. Mukul R. Prasad and Dr. Hiroaki Yoshida for providing me the precious opportunity to work in Fujitsu Laboratories of America during my research internship. I am also grateful for the collaborations with Umair Z. Ahmed and Prof. Amey Karkare.

I would like to thank other fellow students, including Dr. Dawei Qi, Dr. Van-Thuan Pham, Gao Xiang, Manh-Dung Nguyen, and Yulis for their support and assistance throughout my study.

PAPERS APPEARED

- Shin Hwei Tan, Zhen Dong, Xiang Gao, Abhik Roychoudhury. Repairing Crashes in Android Apps. In the *Proceedings of International Conference on Software Engineering (ICSE), 2018, To Appear.*
- Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, Abhik Roychoudhury. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In the *Proceedings of International Conference on Software Engineering Companion (ICSE-C), p180–182.*
- Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, Abhik Roychoudhury. A correlation study between automated program repair and test-suite metrics. In *Empirical Software Engineering Journal (EmSE) 2017, p1–32.*
- Shin Hwei Tan, Hiroaki Yoshida, Mukul Prasad and Abhik Roychoudhury. Anti-patterns in Search-based Program Repair. In the *Proceedings of International Symposium on the Foundations of Software Engineering (FSE), 2016, p727–738*
- Shin Hwei Tan and Abhik Roychoudhury. *relifix*: Automated Repair of Software Regressions. In the *Proceedings of International Conference on Software Engineering (ICSE), 2015, p471–482*

Contents

Design of Repair Operators for
Automated Program Repair

Abstract

Software bug-fixing is a time-consuming software development activity. Recently, many automated program repair approaches are proposed to reduce the time and effort spent in fixing defects. These techniques rely on a fixed set of program transformations (*repair operators*) for generating patches automatically.

We present a comprehensive study on the effectiveness of repair operators in existing repair tools. We also introduce a new benchmark to enable the objective comparison between different repair tools. Moreover, we investigate the possibility of improving the effectiveness of automated repair techniques by enhancing the quality of test suite. Furthermore, we propose anti-patterns, a set of rules that illustrate problems in automatically generated patches. Our set of anti-patterns allows elimination of patches at the level of program transformations. In addition, we introduce a novel repair tool with repair operators drawn from code changes between different versions. Finally, we adopt the idea of program repair to fixing mobile applications by designing specialized repair operators for Android apps. As shown by extensive evaluations, our proposed approaches improve over existing automated program repair techniques.

List of Tables

List of Figures

Chapter 1

The Need for Automated Program Repair

A software bug is an error, failure, or a fault in a computer program that causes it to produce unexpected results. Software errors could lead to delay in software projects and increase of software maintenance cost. According to the US National Institute of Standards and Technology (NIST), software development costs, including the development and the distribution of software patches for fixing software bugs contribute to $60 billion US economic loss each year [151]. The catastrophic effect of software bugs may lead to loss of human lives. One well-known example of such disastrous failures includes software faults in a radiation therapy planning system where at least five people died due to radiation overdose [49].

The interval between the time when a software error is first reported and the time when the error is fixed has significant impact on the reliability of the software. Indeed, a case study of the Mozilla project reveals that a bug report could take more than 100 days to resolve [111]. When a software error is reported, software developers need to go through tedious tasks of locating and fixing the error. In recent years, many automated debugging techniques have been proposed to assist developers in finding the root cause of a software error [77, 125, 147, 148, 149]. These techniques demonstrated promising results in pinpointing the program locations that are relevant to the software error. Although developers could save the time spent in locating the error using these automated techniques, the process of understanding the cause of the error and the bug-fixing step are still primarily manual. Moreover, developers may need to spend additional time in debugging and bug-fixing when some of the issued patches lead to new errors.

To reduce time and effort spent in bug comprehension and bug-fixing, several automated program repair approaches were introduced [63, 67, 90, 94]. These automated repair techniques have various potential usages. First, automated repair approaches could be used to reduce the downtime of remote-control system (e.g., military drones) where the software failures could have disastrous impact (e.g., drone collision). This potential usage is related to the concept of *self-healing software systems* which is defined as software systems that have the ability to detect when the software stops functioning correctly and could restore it to the working condition without any human intervention [84]. According to the definition of self-healing software systems [84], the ability to automatically repair a software bug is essential in enabling the self-healing properties of a software system. Secondly, these automated program repair techniques could expand the capability of existing automated debugging techniques by providing debugging hints together with concrete patch suggestions. MintHint, an approach that automatically provides repair hints to increase developers' productivity demonstrated initial success for this potential usage [78]. However, in MintHint's experiments, repair hints have been studied only for small programs where repair hints took could take up to one hour to generate. Thirdly, although a program repair tool may be utilized in an interactive debugging session where repair hints need to be generated almost instantly, a more realistic scenario is to integrate program repair system together with nightly builds. In this scenario, developers could check in their code into the repository, run the regression test suite (*regression test* is test that verifies whether a software system performs correctly after making some changes to the software), and invoke the program repair system to fix *regressions* (software bugs that make a software stop functioning correctly after certain events, such as system patching). An approach proposed in this thesis, *relifix*, is designed specifically for this usage. Fourthly, even when the automatically generated patches do not completely fix a given defect, automated program repair systems may be still useful in extending the survival of software systems by temporarily circumventing the problem to buy developers more time in crafting the correct patch [127]. Lastly, instead of fixing traditional desktop applications, the concept of automated program repair could also be applied to other applications. Particularly, this thesis suggests the possibility of applying program repair techniques into mobile applications.

While the overall goals of automated program repair techniques are to improve developers' productivity and to enhance software reliability, it could enable many potential usages, including empowering self-healing systems, generating repair hints, integrating with nightly builds, extending the survival of software systems, and fixing applications of other domains.

Despite many possible usages of automated program repair techniques, there remain several challenges before automated program repair could be fully deployed and integrated into software development process.

First, most automated program repair approaches rely on given test cases T for validating the correctness of the generated patch. In the context of program repair systems, test cases serve as specifications that capture the intended behavior of the patched program. However, test cases tend to be incomplete as each test case only represents an input-output relationship of a given program.

As a result, the generated patches may be incomplete fixes and the patched program may also end up introducing new errors, because the patched program may fail tests outside T, which were previously passing [134]. The scenario where the patched program passes T but fail tests outside T is referred to as the problem of *overfitting* in automated program repair literature. Another work that studies the quality of automatically generated patches shows that the vast majority of patches produced by search-based program repair tools are semantically equivalent to functionality deletion [127, 134]. Apart from using test cases as a correctness criterion, several works go one step further by checking manually whether the automatically generated patch [94, 96, 108] is semantically equivalent to the corresponding human patch. Although the manual validation of automatically generated patches is more precise for determining the patch correctness, this may not be practical as developers may need to spend considerable amount of time filtering out a large number of invalid patches.

Second, the reliance of automated program repair techniques on test cases suggests that they may inherit problems that are common in the field of automated testing. In particular, automated testing techniques aim to produce a test suite that fulfills certain coverage criteria. To study the relationship between testing and program repair, in Chapter 4, this thesis investigates whether traditional coverage criteria used to assess the quality of a given test suite for automated testing could be adapted for automated program repair. Apart from assessing the quality of test suite, the ability to handle *flaky tests* (tests that exhibit both a passing and a failing result with the same program version) is another concern in automated testing. Similarly, flaky tests also pose unique challenges for automated program repair techniques. Specifically, since test cases for applications with graphical user interfaces (GUI) are often flaky, automated program repair techniques may misinterpret the test results by producing incorrect patches. To address this problem, this thesis introduces an automated program repair framework for Android applications (which contain GUI tests that fail due to crashes) in Chapter 7.

Third, automated program repair approaches are often not responsive enough to provide instant feedback to software developers as most of these approaches were designed to be invoked during nightly builds where software developers get no control over the bug-fixing process. In the worst case scenario where the desired patch falls out of the patch space of a repair system, program repair systems could take up to 12 hours [88] to traverse the entire patch spaces and yet produce neither working patch nor debugging clue to software developers.

Furthermore, most of the past evaluations of automated program repair tools [90, 95, 96, 108] are restricted to using one benchmark suite for C programs (GenProg ManyBugs benchmark [90]). This means that the current design of search space of these program repair tools may be derived based on the same benchmark. Meanwhile, the best result reported so far on the ManyBugs benchmark is finding correct fixes for only 17% of all the defects [96]. The low percentage of correct patches implies that the current design of the search space is not effective. The effectiveness of repair operators is important for program repair tools because this set of operators define the search space of these tools.

Goal of this thesis The overall goal of this thesis is to improve the current techniques in fixing software defects by addressing many of the aforementioned challenges. To achieve this goal, we investigate the current design of repair operators used in generating patches and propose several techniques for guiding the future design of repair operators in program repair tools.

This thesis makes the following contributions:

- We conduct the first study that investigates the current design of program repair tools and propose several selection strategies that can reduce the number of repair operators to increase the likelihood of obtaining a correct repair for automated program repair techniques. We also propose a new set of subject programs that allows extensive comparison among existing program repair tools.

- We perform the largest study that explores the relationship between traditional test suite metrics and the quality of automatically generated patches. In particular, our study is the first that investigates the correlation between mutation score and repair quality. We also propose a new metric, *capable-tests ratio* (the ratio of tests that kill one of mutants in a given test suite), that is found to be more strongly correlated with the quality of automated generated patches than mutation score.
- We introduce *anti-pattern*, a set of generic rules that forbid certain program transformations for each repair operator. We show that anti-patterns can be enforced on top of any search-based program repair tools. Enforcing anti-patterns allows search-based program repair tools to generate patches faster and patches with more pleasant properties (i.e., better fault localization and removes fewer functionalities).
- We propose *relifix*, a program repair tool that employs a specially designed set of repair operators. Compared to existing approaches that rely solely on the specification of the buggy program, our novel set of repair operators extracts change information from two program versions to reduce the likelihood of introducing overfitting patches. We show that relifix generates more patches and the generated patches are of better quality compared to an existing repair system.
- We introduce *Droix*, a program repair framework that automatically fixes crashes in Android applications. We design a novel set of repair operators based on the Android activity/fragment lifecycle. To compensate for the

inadequate test information for Android applications, Droix also enforces several properties extracted from Android activity/fragment lifecycle management rules. Our experiments show that Droix could generate fixes that are syntactically equivalent to human patches. As Droix aims to minimize the number of violations of several properties, Droix could even generate a patch that is better than the human patch for one defect.

Implications for future research. This thesis demonstrates through extensive studies and concrete techniques that (1) the design of repair operators plays an important role in ensuring the quality of the automatically generated patches and the number of generated patches; (2) the benchmark used to evaluate automated program repair techniques should contain a wide variety of defects fixed using different repair operators to allow comprehensive comparison between various repair approaches; (3) each repair operator has some program transformations that should be forbidden due to the problem of inadequate test coverage; (4) although existing program repair techniques are limited in fixing certain defect types, automatically generated patches could help in providing debugging clues to developers by pinpointing potential fix locations.

The remainder of this thesis is organized as follows. In Chapter 2, we provide an overview of the existing techniques in fault localization, in automated testing, and in automated program repair. Next, we present our study on the set of repair operators in automated program repair tools in Chapter 3. Chapter 4 introduces our correlation study that investigates the relationship between traditional test-suite metrics and the quality of automatically generated repairs. Chapter 5 describes our proposed set of rules that forbid certain program transformation for each repair operator. Chapter 6 presents our new program repair approach that employs a set of repair operators based on program changes information. Chapter 7 introduces a novel set of repair operators for an emerging application of program repair — repairing crashes in Android applications. We conclude this thesis in Chapter 8 by summarizing all the presented techniques and improvement.

Chapter 2

Prior Works on Automated Program Repair

Fault localization is a crucial initial step in automated bug fixing approaches as these approaches rely on existing fault localization techniques for finding the locations in which the patch will be applied. Another essential step for automated bug-fixing approaches involves running test cases for validating each generated candidate patch. Due to the importance of fault localization and testing in automated program repair, this chapter begins by presenting a literature review on fault localization, followed by introducing testing techniques that are relevant to automated program repair. Finally, this chapter ends by presenting existing automated program repair techniques.

2.1 Fault Localization

This section is organized as follows. Subsection 2.1.1 gives a brief overview of spectrum-based fault localization techniques. Subsection 2.1.2 discusses fault localization that uses information from multiple program versions.

2.1.1 Spectrum-based Fault Localization

Spectrum-based fault localization is a technique that analyzes the differences in *program spectra* (a program spectrum is an execution profile that provides the signature of a program's behavior [73]). Spectrum-based fault localization techniques aim to identify the program elements that are relevant to the detection of errors (these program elements are also referred to as "suspicious"). Several metrics have been proposed for measuring the suspiciousness of program statements, including the Tarantula formula and the Ochiai formula.

Tarantula computes a ranked list of statements based on the intuition that statements executed exclusively by failing test cases are more likely to be faulty than statements executed by passing test cases [77]. Tarantula assigns a suspiciousness score for each statement in the program by computing the following equation:

$$\text{suspiciousness}(s) = \frac{\frac{\text{failed}(s)}{\text{total failed}}}{\frac{\text{passed}(s)}{\text{total passed}} + \frac{\text{failed}(s)}{\text{total failed}}}$$

where `passed`(s) represents the number of passing test cases that visited statement s while `failed`(s) represents the number of failing test cases that visited statement s. The suspiciousness score for each statement denotes the likelihood of a statement being the faulty statement that causes the test failures. To provide better visualization, the Tarantula approach also assigns a color to each statement based on its suspiciousness score.

The Ochiai formula is another equation used in computing the suspicious score for fault localization [23]. The following equation shows the Ochiai formula:

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

Experiments in previous studies show that the Ochiai formula used in the molecular biology domain is more effective than the Tarantula formula in ranking program spectra.

2.1.2 Fault Localization based on Multiple Program Versions

To better analyze defects that occur due to program changes, several fault localization approaches incorporate information about multiple program versions, including Darwin [125] that uses a reference program, an approach proposed by Banner er al. [42] that uses golden implementation, and delta debugging [148] that uses code changes between program versions.

Darwin is a fault localization approach for debugging evolving programs [125]. It takes as inputs a reference program P, a modified program P' together with an input t that fails on the modified program and produces as output a bug report that pinpoints an observable error. It first concretely executes the test input t in both programs to generate program traces, then it feeds the produced traces into the symbolic execution engine to produce path condition f for P and path condition f' for P', then uses a STA solver to check for the satisfiability of $f \wedge \neg f'$ and $f' \wedge \neg f$. The solutions for the two conjunctive formulas are included in the bug report as the potential root cause for the test failure. Darwin has been successfully applied to several real world applications, such as the libPNG library.

Banerjee et al. [42]pointed out the difficulties in using Darwin for debugging two programs that may yield logical equivalent path conditions and proposed using golden implementation as the reference model for solving this problem. Their approach employs dynamic slicing with respect to the error of interest and performs weakest precondition computation along the dynamic slices. They have conducted experiments using different versions of libPNG and demonstrated promising results in deriving constraints that capture the root cause of a buffer overrun error.

Delta debugging is a debugging method that isolates failure-inducing circumstances that are responsible for test failures using a divide-and-conquer algorithm. There are two variants of the delta debugging approach. One variant of the approach uses the set of code changes as the failure-inducing differences while another variant uses differences between passing and failing test cases as the "delta". The first variant of the delta debugging approach takes as input a set of changes between two program versions and it produces a minimal subset of changes that correspond to the failure by performing binary search [148]. The second variant simplifies the failing test input through successive testing and generates a minimal subset of a failure-inducing input by recursively partitioning test cases into two subsets of similar sizes [149]. Yu et al. conducted an extensive evaluation of the effectiveness of delta debugging by applying the approach on real regression bugs [147]. Their results revealed that delta debugging may isolate superfluous changes that are irrelevant to the failure. In particular, three out of 15 regression bugs investigated contain superfluous changes after isolation mainly due to refactoring that occurs between the two program versions. They also discovered that delta debugging could produce wrong isolation with changes that are irrelevant to the failure. They further explained that the wrong isolation may be due to the choice of test function used and the inclination of delta debugging in isolating single changes first.

2.2 Testing and Automated Program Repair

Automated program repair approaches that leverage test cases for checking whether a candidate patch fixes a given error are referred to as *test-driven program repair approaches*. Test-driven program repair approaches take as input a buggy program and a test-suite in which at least one test fails in the given buggy program. Tests that fail in the buggy program are called *negative tests*, whereas those that pass are called *positive tests*. The goal of test-driven repair tools is to find an edit of the buggy program that passes all the tests in the provided test-suite. This section describes several testing techniques including mutation testing and regression testing that are associated with test-driven program repair approaches.

Typically, a test-suite used for a test-driven program repair tool consists of a small number of negative tests and a relatively large number of positive tests. While negative tests set a goal of repairing the given buggy program, positive tests serve as anti-goals; they filter out hazardous repair candidates, that is, those that pass negative tests but fail one of positive tests. Still, due to the incomplete nature of tests (not all desirable behaviors of software are tested), a test-driven repair tool runs the risk of generating repairs causing regressions.

2.2.1 Mutation Testing and Program Repair

Mutation testing is a systematic method to measure the fault detection ability of a test-suite [118]. In mutation testing, a given program is modified (mutated) by applying mutation operators at several program locations. These modified programs (mutants) represents typical programming errors that should be detected by the given test suite. A mutant m is considered *killed* when the test result of m for at least one test in the provided test-suite is different from the test result of the original program for the same test. The fault detection ability of a test-suite TS is measured as the *mutation score* of TS, which is defined by:

$$\text{Mutation Score}(s) = \frac{\text{number of killed mutants}}{\text{total number of mutants}}$$

Namely, mutation score measures the ratio of the number of killed mutants over the total number of non-equivalent (i.e., semantically different) mutants.

Automated program repair has similarities to mutation testing. It can be viewed that automated program repair "mutates" the original program, this time in an attempt to find a repair. As in mutation testing, mutants that fail to pass all tests in the provided test-suite are considered buggy (hence, incorrect repairs). This conceptual similarity between mutation testing and automated program repair suggests the plausibility of using the mutation score to measure the quality of a test-suite not only for mutation testing but also for automated program repair. Just as a higher mutation score is associated with a better fault-detection ability in mutation testing, it appears plausible to associate a higher mutation score with a better ability to guide a reliable repair.

2.2.2 Regression Testing

Regression is a software bug that makes a feature stop functioning correctly after a certain event (e.g., bug-fixing). Regression testing is a technique that aims to detect regression when software evolves [148]. As the size of regression test suite may grow as software evolves, it may be too time and resource consuming to execute the entire test suite. To reduce the cost of regression testing, prior regression testing techniques focus on test suite selection (technique that selects a subset of a given test suite) [72, 130], test suite minimization (technique that reduces the test suite size) [101, 119, 137] and test suite prioritization (technique that find an ordering of the given test suite for re-execution) [92, 131].

2.3 Automatic Program Repair Techniques

Several automatic program repair techniques have been proposed to generate patches for buggy programs. Subsection 2.3.1 gives a brief overview of search-based automated repair techniques. Subsection 2.3.2 describes automated repair technique that uses semantic techniques (i.e., program synthesis). Subsection 2.3.3 discusses automated repair techniques that leverages Eiffel contracts. Subsection 2.3.4 discusses various automated repair that incorporates domain-specific information.

2.3.1 Search-Based Repair

```
Input: P: Program
Input: T: Test Suite
Input: RepOp: Repair operator
Input: Fit: Fitness Function $P \rightarrow \mathbb{R}$
Input: PopSize: Population size
Output: Patch: a program variant P' that passes $\forall t \in T$
1 $Pop \leftarrow initialPopulation(PopSize)$;
2 repeat
3     $Pop \leftarrow apply(Pop, RepOp)$;
4     $Pop \leftarrow select(Pop, PopSize, Fit)$;
5 until $\exists Patch \in Pop$ that passes $\forall t \in T$ ;
6 return $Patch$
```

Algorithm 1: Search-based Program Repair Algorithm

Algorithm 1 depicts the patch generation algorithm for search-based program repair approaches [82, 90, 126]. A search-based program repair approach *Rep* takes as input a fitness function, a set of repair operators *RepOp*, and population size. After creating an initial population of size *PopSize* (Line 1), *Rep* repeatly performs two steps: generating new program variants by using its predefined set of operators (Line 3) and selecting top program variants based on the given fitness function (Line 4). This iteration process terminates when any program variant passes all given test cases are found (Line 5) or when all patches have been evaluated.

Genetic Program repair (GenProg [90]) is a search-based repair approach that uses genetic programming to search for a program variant that passes a set of test cases. Genetic programming is an algorithm motivated by biological evolution to discover programs that perform a specific task. GenProg takes as input a buggy program and a set of test cases, including a test case that encodes the buggy behavior of the program. It first performs fault localization at the statement-level, then it utilizes its mutation and crossover operators (Line 3 in

Algorithm 1: *RepOp* is based on mutation and crossover) to generate program variant. Each generated program variant are evaluated using a fitness function that computes how close the given program variant is to the final repair that passes all test cases (Line 4 in Algorithm 1: selection based on fitness function). Before generating the final repair, it uses structural differencing and delta debugging to exclude irrelevant codes that are introduced along with the repair. While the initial version of GenProg [142] uses genetic programming algorithm to modify the buggy program, the latest version of GenProg [141] (AE) employs a deterministic algorithm together with program equivalence relations to generate patches. Another search-based repair approach, RSRepair [126] differs from GenProg [90] in the algorithm used for selecting program variants (Line 4: GenProg uses genetic algorithm whereas RSRepair uses random search).

Arturo and Ya [34] proposed another framework that uses genetic programming for simultaneously evolving the buggy program for passing the current set of test cases and the tests used for finding bugs in the evolved program. The approach takes as input a buggy program and formal specification that are used for generating new test cases. The authors hypothesize that the co-evolution of the program and the test case could lead to the evolution of a program that satisfies the formal specification. They illustrate their idea of co-evolution through a case study on the bubble-sort implementation. Arturo later [33] presented a Java prototype called JEFF that automatically repair faulty Java programs with three search algorithms, including random search that randomly mutates the buggy program, a variant of Hill Climbing (HC) that iteratively searches for the best solution among neighbors (in the context of repair, the neighbors are different mutation operators), an evolutionary algorithm (i.e., genetic programming) [33] for evolving the buggy program and chooses program variant based on a fitness function. Their experiments show that genetic programming performs significantly better than random search and Hill Climbing.

In mutation testing, program mutation makes syntactical changes to a program and measures the quality of a given test suite by checking its effectiveness in detecting program modifications. Debroy and Wong suggested combining mutation and fault localization to automatically produce fixes for faulty programs [62]. Their approach uses Tarantula [77] for fault localization and then uses "mutant operators" (repair operators) to generate fixes for buggy programs. They include two classes of repair operators (*RepOp* in Algorithm 1): (1) replacement of arithmetic, relational, logical, increment/decrement, assignment operator by another operator and (2) decision negation in an if/while statement. Their evaluations on programs in Siemens suite and Apache Ant demonstrate that 18.5% of the faulty versions can be automatically fixed.

Automatic program repair using genetic programming [90] may suffer from the problem of generating nonsensical patches due to the random changes introduced by mutation operations. To overcome this problem, Kim et al. proposed Pattern-based Automated program Repair (PAR) that utilizes common fix patterns learned from manual inspection of human patches [82] (*RepOp* in Algorithm 1 is the set of fix patterns) . Examples of these fix patterns include adding null check and inserting object initialization. Similar to GenProg, PAR uses evolutionary algorithm to generate patches by applying extracted fix templates to produce program variants, evaluating the variants through a fitness function and selecting the best variant based on a fitness function. Their user study demonstrated that patches generated by PAR are more "acceptable" than those produced by GenProg.

Another search-based approach — SPR searches for patches that do not involve conditional statements using similar search-based algorithm as GenProg [94]. While it repetitively generates repair candidates until a repair is found as in GenProg, it also applies the following approach for conditional statement: given a potentially buggy conditional statement that is executed more than once during test execution, SPR searches for a sequence of boolean values that the conditional statement should have to pass the test. Once such sequence is successfully found, an expression that can produce the inferred sequence during the execution is synthesized. Prophet extends SPR with a ranking function and machine-learning algorithm to prioritize a repair similar to human patches, when there are multiple repairs found [96]. Prophet and SPR share the same set of repair operators.

2.3.2 Semantic-Based Repair

Semantic-based program repair approaches first construct repair constraints that should be satisfied by the patched program, then they use program synthesis engine for generating fixes that satisfy the repair constraints.

SemFix is the first semantic-based repair technique that employs program synthesis to search for the correct repair for a given buggy program [117]. It takes as input a faulty program and a test suite with at least one failing test and it follows a three-steps approach listed below:

Fault localization Performs statistical fault localization to generate a ranked list of suspicious statements

Specification inference Replaces the expression in the most suspicious statement by a symbolic expression x, then executes the program symbolically starting from the suspicious statement. This execution yields a set of formulas that can be used as a specification of the program semantics for all the paths that starts from the statement of interest.

Program Synthesis Uses component-based synthesis to find an expression that satisfies the inferred specification obtained from previous step.

As SemFix only modifies conditional statements and the RHS of assignments, its repair operators include ite, constant replacement, arithmetic, logical and relational operator replacement. Their results demonstrate that SemFix perform significantly better than GenProg in terms of number of generated repairs (repair rate). They also concluded that SemFix has higher repair rate in almost all types of bugs, especially arithmetic bugs where the repair expressions do not exist within the buggy programs. This observation illustrates the inherent limitation of GenProg since it assumes that the repair expression is present in other part of the buggy programs.

DirectFix [107] generates *minimal* repairs to obtain human-readable and comprehensible repairs. The idea is to encode the problem as a partial maximum satisfiability problem over Satisfiability Modulo Theory (SMT) formulas (partial maxSAT) and solve it using a suitably modified SMT solver. Angelix [108] solves the scalability problems of DirectFix by using a lightweight repair constraint. Both DirectFix and Angelix reuse the same set of repair operators as SemFix.

NOPOL is a semantic-based repair approach that fixes bugs in conditional statements [143]. Given a buggy Java program, NOPOL first uses angelic fault localization to identify the value that makes test cases pass (this value is called the *angelic value*). Then, it collects runtime information on existing program variables through code instrumentation. Finally, it synthesizes a new conditional expression that matches the angelic values using a SMT solver. Apart from modifying expressions in conditional statement as in SemFix, NOPOL introduces additional repair operators, including inserting preconditions, adding null checks and inserting method calls.

2.3.3 Contract-Based Repair

Instead of relying on test cases as specification for guiding the repair process, contract-based repair approaches leverage program contract. Autofix-E is a contract-based program repair technique that uses contracts for ensuring the correctness of the proposed fixes [140]. Autofix-E first automatically generates test cases using a random testing tool named AutoTest [56], then it uses the following information to synthesize potential fixes:

Object behavioral model It derives the object behavioral model by analyzing how different routines in the objects affect different queries in all passing tests. The object behavioral model gives the relevant routines to call in order to change the object state to avoid the failure.

Fault Profile It generates the fault profile by analyzing the differences between the executions of the passing test cases and the executions of the failing test cases. The fault profile provides a hint of the possible causes of the failure.

Using both the fault profile and the object behavioral model, Autofix-E then generates fixes with a set of "fix schemas" (repair operators). Most of these repair operators involve insertion of conditional statements. If several valid fixes are generated, it uses dynamic and static information to further rank the fixes. Their evaluation on two Eiffel libraries illustrates that Autofix-E could repair 16 out of 42 faults discovered by AutoTest.

Since Autofix-E relies on boolean queries to derives object states, it can only fix faults that involve boolean expression. To extend the applicability of Autofix-E to more general expressions, the authors proposed another approach called Autofix-E2 [123]. Autofix-E2 follows a similar workflow as its predecessor, but it extracts a set of expressions from the program text instead of a set of object states. It uses both static and dynamic information to assign a suspiciousness score to each component $\langle l, p, v \rangle$ where predicate p evaluates to value v at program location l. Autofix-E2 uses the same set of repair operators as Autofix-E but it instantiates the repair operators with p and an action derived from p. Their experiment shows that Autofix-E2 is able to produce significantly more fixes compared to Autofix-E.

2.3.4 Repair that Incorporates Domain Specific Knowledge

There are several program repair techniques that utilizes domain specific information. In particular, PACHIKA relies on differences between passing and failing runs to automatically infer object behavioral model from Java program and produce fixes by either inserting or deleting method calls [61]. BugFix is a tool that incorporates information gathered from several debugging session in order to increase precision for producing bug-fix suggestions [74]. R2Fix closes the loop between bug report submission and patch generation by automatically classifying the type of bug discussed in bug report and extracting pattern parameters to generate fixes based on a several predefined fix patterns [93]. PHPRepair fixes malformed HTML generation errors by encoding the string output for each test case execution as a constraint over variables corresponding to constant prints in the program and uses constraint solver to generate string modification [132]. Martinez and Monperrus mine semantic code modifications (repair operators) from human patches and attach a probability distribution to the mined repair actions [104]

Chapter 3

Selections of Repair Operators in Automated Repair

Several automated repair tools (e.g., GenProg [141], PAR [82], SemFix [117], DirectFix [107], Angelix [108], SPR [94], and Prophet [96]) have been introduced to save the time and effort spent in bug-fixing. Although these repair techniques demonstrate promising results [107, 126, 138, 141], prior evaluations of these techniques have several limitations.

First, prior evaluations of program repair tools ignore the impact of defect classes to the effectiveness of program repair tools [90, 95, 96, 108]. The effectiveness of program repair techniques relies heavily on the type of faults that they are designed to fix. As automated program repair techniques use a set of *repair operators* (edit operators that are used to generate patches) for fixing faults, the quality of the repair operators is critical to the effectiveness of repair techniques. Although the importance of studying the set of repairable defect classes are emphasized in several program repair works [89, 112], most studies only measure effectiveness in terms of the probability of generating fixes among evaluated defects (we call this the *repair rate*) and the probability of obtaining correct patches (we call this the *repair precision*) where the correctness is validated by performing: (1) *semantic equivalence check* that compares automatically generated patches with human ones [95], or (2) validation to check if the automatically generated patches pass the *held-out test suite* (a set of tests that are not revealed to the repair system) [134].

Moreover, almost all past evaluations of automated program repair tools [95, 96, 108, 141] for C programs use GenProg ManyBugs benchmark [141] which means that the current design of repair operators of these program repair tools may be derived based on the same benchmark. Moreover, the best result reported so far on the ManyBugs benchmark is finding correct fixes for only 18/105=17% of all the defects [96]. This raises several questions regarding the effectiveness of program repair tools: (1) how well these tools perform outside of the evaluated benchmarks and (2) how well these tools fix classes of defects that they are designed to address.

More importantly, prior research mainly focuses on the improvement of the effectiveness of the underlying algorithms [96, 108, 141] but neglects the effectiveness of individual repair operators which is one of the most essential components of program repair tools. In fact, the effectiveness of individual repair operators is either not measured in prior studies [107, 82, 108, 141] or has only been measured in terms of repair rate [91]. Without thorough investigation of the effectiveness of each repair operator in current program repair tools and comprehensive comparisons of these repair operators across different tools, the reasons behind the ineffectiveness of repair tools could not be assessed.

These three problems of prior evaluations of program repair tools limit the insight into their effectiveness and pose challenges to the objective evaluation of future tools. Consider a developer of a new program repair tool who would like to decide whether to add new repair operators to fix new types of defects or to employ a set of repair operators commonly used by existing program repair tools. While adding new repair operators seem to be a viable solution, prior study reveals that increasing the search spaces by supporting more repair operators may cause repair systems to find less correct patches [95].

We present the first large-scale extensive evaluation of the effectiveness of program repair tools. Specifically, we measure the impact of defect classes on the effectiveness of different program repair tools. Our study evaluates several repair techniques, including a search-based tool (GenProg), a tool that uses machine-learning (Prophet) and a semantics-based tool (Angelix). To allow comprehensive comparisons between program repair tools, we present a new benchmark called *Codeflaws*. The Codeflaws benchmark provides more objective evaluation of program repair tools across different defect classes. We employ the defect classifications in Codeflaws to study the impact of defect classes on the effectiveness of repair operators in program repair tools. Furthermore, we evaluate the effectiveness of individual repair operators by measuring both the repair rate and the correctness of patches generated using each repair operator. Indeed, prior research suggests that biased selection of mutation operators for evolutionary program repair techniques leads to an increase of the number of generated repairs and a decrease of repair time [91]. While the study provides one of the first steps towards understanding the role of repair operators in evolutionary program repair techniques, the effect of the *reliability of repair operators* (i.e., the probability that a repair generated using a particular repair operator corresponds to a correct repair) in existing automatic program repair techniques has not been previously studied. Our study goes beyond prior work, analyzing the effect of repair operators in several automated program repair techniques, and investigating not only the number of generated repairs but also the reliability of repair operators in existing automated program repair tools. In addition, we also propose a novel metric for evaluating repair operators based on the number of repairable defect classes (we call this the *expressiveness* of repair operators). Our metric evaluates the fault fixing capability of repair operators taking into accounts different types of defects. On top of that, we propose three selection strategies to choose a small number of repair operators in repair tools to balance the trade-offs between repair rate and repair precision.

3.1 Common Set of Repair Operators

To better analyze the set of repair operators supported by existing program repair tools, we performed a detailed analysis on the common syntactic features of all repair operators supported by each program repair tool. Specifically, we manually analyzed the implementation and definition of each repair operator supported by three state-of-the-art repair tools (e.g., GenProg, Prophet and Angelix). As different extensions of GenProg (e.g., AE and RSRepair) share the same set of operators as GenProg, the same analysis is applicable to other GenProg's variants. Similarly, SPR and Prophet share the same set of operators. Meanwhile, the set of repair operators in Angelix is similar to the operators supported in its predecessor, SemFix.

Our analysis attempted to answer two questions:

Q1: What is the set of repair operators supported by a specific program repair tool?

Q2: Do program repair tools share any common repair operators?

The aim was to find a set of repair operators that are supported by either some or all existing program repair tools.

Table 3.1 shows the results of our analysis. Specifically, our analysis identified a set of 17 repair operators, shown in each row of Table 3.1, seven of which are supported by GenProg, 10 of which are supported by Prophet/SPR (SPR uses same repair operators as Prophet), and, nine of which supported by Angelix. To allow a broader understanding of each repair operator, we further classify the 17

Table 3.1: Our set of Repair Operators and Example of each Repair Operator

AST Type	Repair Category	Repair Operator	Example	GP	Prophet	Angelix
Statement	Control flow	(SDIF) Delete control flow statement (e.g., if, while, return, break)	- break;	✓	✗	✗
		(SIIF) Insert control flow statement (e.g., if, while, return, break)	+ if(l)	✓	✓	✓
		(SIIR) Insert conditional return/exit statement	+ if(a) return 0;	✓	✓	✗
	Assignment	(SDLA) Delete assignment	- answer+=((i-1)*dif);	✓	✗	✗
		(SISA) Insert assignment	+ t=0;	✓	✓	✗
	Function call	(SDFN) Delete function call	- printf("%s %s\n",s1,s2);	✓	✗	✗
		(SISF) Insert function call	+ scanf("%d", &n);	✓	✓	✗
		(SIME) Insert Initialization	+ memset (str,'+',6);	✗	✓	✗
Operator	Logical	(ORRN) Replace relational/logical operator	- if(sum>n) + if(sum>=n)	✗	✗	✓
		(OILN) Insert && (tighten condition) or || (loosen condition)	- if(t%2==0) + if(t%2==0 && t!=2)	✗	✓	✓
	Arithmetic	(OAAN) Replace arithmetic operator	- v2-=d; + v2+=d;	✗	✗	✓
	Function call	(OFFN) Alternative function call	- max(a, b, c); + min(a, b, c);	✗	✓	✗
Operan**D**	Constant	(DCCR) Replace constant with variable/constant	- for(i=n+1;i<=9000;i++) + for(i=n+1;i<=10000;i++)	✗	✓	✓
	Variable	(DRVV) Replace a read variable with a variable/constant	- for (i=0;i<l;i++) + for (i=0;i<m;i++)	✗	✓	✓
	Expression	(DRVA) Replace subexpression	- i=1 + i=i+1	✗	✓	✓

Higher order	Statements	(HIMS) Insert multiple statements	+ if(a) a=a+1;
			+ if(i>2) i++;
				✗	✗	✓

	Expressions	(HEXP) Replace multiple operators and operands	- if(sum>n) + if(sum>=n) - n=n+1 + n=n+2	✗	✗	✓

repair operators into 11 repair categories, and four AST types. A *repair category* denotes the general syntactic structure that is modified for constructing a repair. An *AST Type* denotes the AST node involved for each modification, which includes single program modification of statement, operator, operand and higher order modifications that contain combinations of several single program modifications. The last three columns of Table 3.1 shows the set of repair operators supported by GenProg (i.e., GP), SPR and Angelix, where a ✓ for a particular row in the table means that the repair operator in that row is supported, whereas a ✗ means that the repair operator in that row is not supported. In each example patch in column four in Table 3.1 and in patches presented throughout the thesis, code with a leading "-" denotes statements deleted by the patch, while code with a leading "+" marks statements added. Code without any leading symbol denotes unchanged statements.

3.2 Effectiveness of Repair Operators

Studies [82, 91, 94, 108, 134, 141] that evaluated automated program repair tools used mainly two metrics for the measurement of effectiveness of repair tools—*total number of generated repairs* and *total number of correct repairs* (with respect to either human patch or the held-out test suite). We measure the effectiveness of repair operators based on: (1) repairability, (2) reliability, and (3) expressiveness.

Compared to previous studies that measure the total number of generated repairs in all evaluated defects, we measure the total number of repairs produced by individual repair operator. Specifically, we measure the repairability of a repair operator to calculate its ability to generate repairs.

Definition 1 *A patch is **plausible** if it passes all tests in the repair test suite. A plausible fix is **correct** if it passes all tests in the held-out test suite.*

Definition 2 ***Repairability** of a repair operator op is the ratio of defects repaired using this operator over all defects:*

$$Repairability(op) = \frac{\#plausible\ fixes\ by\ op}{\#defects}$$

Previous measurement of patch correctness are performed using: (1) *semantic equivalence check* that manually compares generated patches with human patches [95], and (2) patch validation via test executions by checking whether generated patches pass the held-out test suite [134]. We choose to measure correctness by using the held-out test suite because this measurement could be computed automatically.

Definition 3 ***Reliability** of a repair operator op is the average ratio of correct* *repairs over all repairs generated by this operator:*

$$Reliability(op) = \frac{\sum_{d \in defects} \frac{\#correct\ fixes\ by\ op\ for\ d}{\#plausible\ fixes\ by\ op\ for\ d}}{\#fixed\ defects}$$

Previous research has demonstrated the problem of overfitting in automatically generated patches [134]. Note that reliability is dual to overfitting: higher reliability of repair operators reduces the likelihood of producing patches that overfit the provided test suite.

Definition 4 *Two defects are in the same* ***repair class*** *if they are repaired by human developers using the same repair operator. A repair class is* ***expressible*** *by a repair operator if it contains a defect correctly repaired by this operator. The* ***expressiveness*** *of a repair operator op is the ratio of repair classes expressible by this repair operator over all repair classes.*

$$Expressiveness(op) = \frac{\#repair\ classes\ expressible\ by\ op}{\#repair\ classes}$$

We introduce the *expressiveness* of repair operators, which measures the ability of a repair operator to fix defects with different repair classes. A repair operator that is expressive could be used for fixing a greater variety of repair classes. We measure the expressiveness of repair operators because the effectiveness of repair operator can vary depending on the type of defects under repair. It is important to quantify the variation and evaluate if repair operator is capable of fixing various types of defects. While both the reliability and expressiveness are measured by calculating the number of correct repairs, the reliability of repair operator measures only the likelihood of generating a correct repair, whereas the expressiveness measures the ability to fix different defect types. Figure 3-1 shows 4 different repair classes (DCCR, SDIF, SISA and SMOV) that are expressible

```
// Human Patch using
// DCCR (constant replacement)
-  if((x*y)>1) {
+  if((x*y)>0) {
//Angelix Patch using ORRN
-  if((x*y)>1) {
+  if(((x * y) >= 1)) {
```

```
// Human Patch using
// SDIF (deletion of if-statement)
-  if(sum==0)
-  printf("-1");
//Angelix Patch using ORRN
-  if(sum==0)
+  if((sum < 0))
      printf("-1");
```

```
// Human Patch using
// SISA (insertion of assignment)
+ dif3 = mx3 -min3;
  x = n - min1 - min2 -min3;
  if( x == 0) { printf(...); }
  else {...
    if( x <= dif2 ) printf(...);
    else {
     x = x - dif2;
     if( x <= dif3 )
        printf("%d %d %d", mx1, mx2,
            min3+x);
//Angelix Patch using ORRN
-    if( x <= dif3 )
+    if((x != dif3))
        printf("%d %d %d", mx1, mx2,
            min3+x);
```

```
// Human Patch using
// SMOV (move statements)
if(n==1){ printf("%d\n",ara[0]); }
      else{...
      }}
-    } //this bracket is moved
      if(count==n){
          printf("%d\n",min); }
      else{
          printf("-1\n"); }
+    }
//Angelix Patch using ORRN
-  if(n==1){ printf("%d\n",ara[0]);}
+  if((n < 1)){ printf("%d\n",ara[0]);}
```

Figure 3-1: Human patches across different repair classes and how Angelix can express these classes using only the ORRN repair operator.

by the ORRN (replacement of relational/logical operator) operator used in Angelix. Particularly, the ORRN operator could stimulate the DCCR (replacement of constant) operator and the SDIF (deletion of if-statement) operator by replacing an existing relational operator with another similar relational operator. Interestingly, the ORRN operator could also express more complex repair classes that are more than operator replacement, such as the SISA (insertion of assignment) operator and SMOV (move statements) used in human patches. This example illustrates that expressiveness is a measure of the diversity of defects that could be simulated by a repair operator rather than the ability to generate equivalent mutants.

3.3 Evaluation

3.3.1 Codeflaws Benchmark

The Codeflaws benchmark consists of 7436 programs in the Codeforces[1] online database. Table 3.2 lists the information about the subject programs in Codeflaws. Each *programming contest* consists of multiple *programming problems* (3–5 problems) with various difficulty levels. Each *program* represents one user submission for a specific problem to Codeforces. These programs are submittted by 1653 users with diverse level of expertise, ranging from "Newbie" to "Legendary Grandmaster"[2]. Each *defect* is represented by a rejected submission and an accepted submission. To obtain more detailed information about each submission, users of the benchmark could refer to the original submission link: http://codeforces.com/contest/<contest-num>/submission/<submission-num>. In terms of the quality of the held-out test suite, Table 3.2 shows that the held-out test suite has relatively high code coverage (the average statement coverage is 98.6%, wheareas the average branch coverage is 95.8%). Moreover, there are several differences between Codeflaws and other benchmarks: (1) the held-out test suite are manually crafted by the designer of each programming problem for automatically grading of many significantly different solutions, (2) the correctness of each patched program could be checked against the complete held-out test suite in the Codeforces platform (test cases with long output could only be partially captured in our crawled held-out test suite); and (3) the oracle for each test checks for the precise output of a given program (instead of merely checking the exit status of a program, it validates whether the content and the format of the output match with the expected output).

To our best knowledge, in automatic program repair evaluation, the Codeflaws

[1]http://codeforces.com/
[2]http://codeforces.com/blog/entry/20638

Table 3.2: Subject programs in Codeflaws

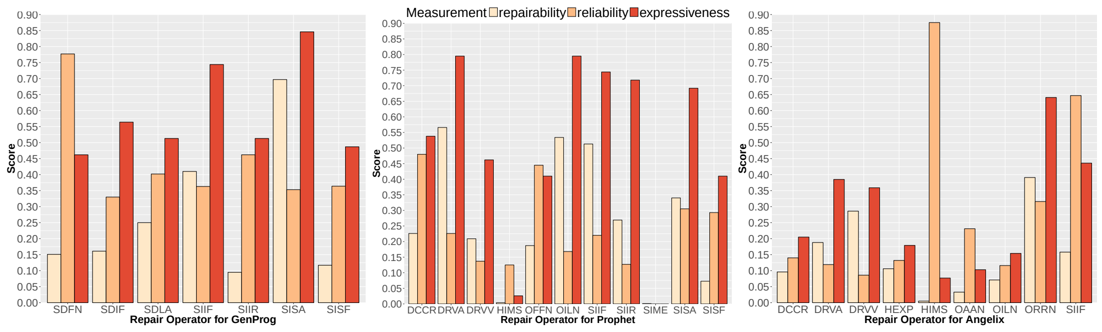
Measurement	Total/Range	Average
No. of Programming Contests	548	-
No. of Programming Problems	1284	-
No. of Programs	7436	-
No. of Defects	3902	-
Size of Repair Test Suite	2–8	3
Statement coverage of Repair Test Suite (%)	6.45–100	96.9
Branch coverage of Repair Test Suite (%)	4.65–100	91.4
Size of Held-out Test Suite	5–350	40
Statement coverage of Held-out Test Suite (%)	9.76–100	98.6
Branch coverage of Held-out Test Suite (%)	10.0–100	95.8
Source Lines of Codes	1–322	36

Table 3.3: Subject programs in IntroClass

Program	Description	LOC	Defects		Tests	
			bb	wb	bb	wb
checksum	Checksum of a string	13	29	49	6	10
digits	Digits of a number	15	91	172	6	10
grade	Grade from score	19	226	224	9	9
median	Median of 3 numbers	24	168	152	7	6
smallest	Minimum of 4 numbers	20	155	118	8	8
syllables	Count vowels	23	109	130	10	10
total		114	778	845	42	53

benchmark has the largest number of defects obtained from the largest number of subject programs to date. We think that the diversity in subject programs and defects would help us to get a comprehensive comparison of different repair techniques.

We compare the effectiveness of repair operator on GenProg, Prophet, and Angelix using two sets of benchmarks: (1) the Codeflaws benchmarks and (2) the IntroClass benchmarks [89]. We use the Codeflaws benchmarks because (1) it contains diverse types of real defects that allows us to have comprehensive study of the impact of defect types on the effectiveness of repair operators; (2) it contains many defects that could be automatically repaired by repair tools, which is important for thorough investigation of the patches generated using different repair operators; and (3) it contains specially crafted held-out test



(a) Repair operator for GenProg (b) Repair operator for Prophet (c) Repair operator for Angelix

Figure 3-2: Individual repair operator score for repairability, reliability and expressiveness for GenProg, Prophet and Angelix on Codeflaws

suites that allows automatic evaluation of patch correctness. We use the IntroClass benchmarks because (1) it is designed for empirical evaluation of automated program repair tools with held-out test suites for evaluation of patch correctness and (2) it is commonly used in previous empirical study of automated program repair tools [76, 79]. We use Pearson's chi-squared test [122] for statistical significance.

Our study aims to investigate the following research questions.

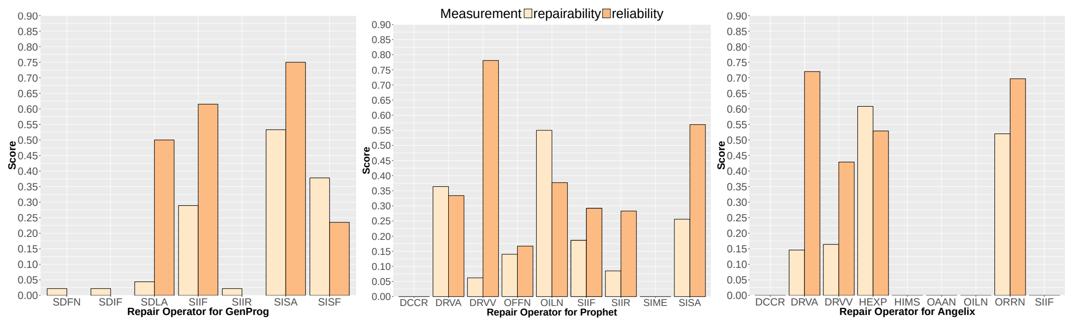
RQ1: Considering the entire search space of a given repair tool, what are the repairability, reliability and expressiveness of each supported repair operator in each program repair tools?

RQ2: Are there any redundant repair operators in each program repair tools?

RQ3: Considering the entire search space of a given repair tool, what are the repair rate versus repair accuracy trade-off for various repair operator selection strategies?

3.3.2 Experimental Setup

We evaluate the effectiveness of three automated program repair tools in 3902 defects from the Codeflaws benchmark and in 778 defects from the IntroClass benchmark [89].



(a) Repair operator for GenProg (b) Repair operator for Prophet (c) Repair operator for Angelix

**As the classifications of repair classes and the correct human fix for each defect are not available for the IntroClass benchmark, we exclude the expressiveness of operators.*

Figure 3-3: Individual repair operator score for repairability and reliability for GenProg, Prophet and Angelix on IntroClass

3.3.3 RQ1: Effectiveness of each Repair Operator

Figure 3-2 shows the individual operator measurement for three program repair tools: GenProg, Prophet and Angelix for Codeflaws benchmark, while Figure 3-3 depicts the same measurement for IntroClass benchmark. Each bar in Figure 3-2 and Figure 3-3 represents the individual score for repairability, reliability and expressiveness for each repair operator in the x-axis.

According to Figure 3-2a, SISA (insert assignment) operator outperforms other operators in terms of repairability, reliability and expressiveness. While SDFN (delete function call) operator is higher than SISA in reliability, SDFN operator has relatively low repairability. Similarly, Figure 3-3a shows that SISA operator also outperforms other operators in IntroClass.

Figure 3-2b shows that DRVA (variable replacement) operator performs better than other operators in terms of repairability, reliability and expressiveness. While OILN (tighten/loosen condition) operator is relatively high in repairability and expressiveness, it has relatively low reliability. This indicates that repairs generated using OILN operator tend to overfit the provided repair test suite. Meanwhile, the top two most effective repair operators for IntroClass (Figure 3-3b) are also DRVA and OILN.

Figure 3-2c illustrates that ORRN (replace relational/logical) operator in Angelix outperforms other operators in Codeflaws. ORRN operator is also most effective for IntroClass (Figure 3-2c). Based on Table 3.1, ORRN operator is uniquely supported by Angelix. Although HIMS (multiple insertions) operator is also supported uniquely by Angelix, this operator is low in repairability and expressiveness.

When we compare scores for individual repair operators across different repair tools, we observe that SISA (insert assignment) operator is high in repairability and expressiveness in both GenProg and Prophet. This indicates that GenProg and Prophet generate similar patches using SISA operator, and they share similar design of this operator. In contrast, SIIF operator is more reliable in GenProg than in Prophet but has higher repairability in Prophet. This difference is because (1) Prophet invokes a specialized search algorithm only for repair operators that involve condition synthesis; (2) Prophet enumerates a larger set of patches to synthesize a condition for SIIF operator, whereas GenProg copies an if-statement from other parts of the same program for SIIF operator. Meanwhile, instead of being restricted by the fixed templates as in Prophet, Angelix synthesizes conditions using its synthesis engine for SIIF operator. In all these cases, we observe how the repairability, reliability and expressiveness of each repair operator reflect the differences in the design of each operator.

3.3.4 RQ2: Redundancy of Repair Operators

We consider an operator as *redundant* if this operator has a negligible impact on the number of correct patches and the number of repaired defects.

As shown in Figure 3-2b, SDME operator generate repair for less than 0.001% of the total evaluated defects in Codeflaws and these repairs are all incorrect repairs. Therefore, we consider SDME operator as redundant for

Prophet in Codeflaws benchmark. Similarly, as SDME operator is not used to generate any patches in IntroClass benchmark (Figure 3-3b), we consider SDME operator as redundant for Prophet in IntroClass benchmark. Moreover, DCCR operator in Prophet is redundant in IntroClass benchmark as well.

Figure 3-2c demonstrates that HIMS operator has relatively low repairability for Angelix despite its high reliability. Figure 3-3c illustrates that HIMS operator is not used in any patches generated by Angelix. Hence, we consider HIMS operator as redundant for Angelix in IntroClass benchmark. Moreover, DCCR, OAAN, OILN and SIIF operators are also redundant for Angelix in IntroClass benchmark.

There is no redundant operator for GenProg in Codeflaws benchmark. Meanwhile, Figure 3-3a shows that SDFN, SDIF and SIIR are redundant in GenProg for IntroClass benchmark.

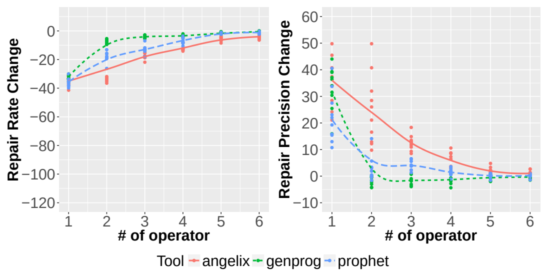
Overall, the redundancy of repair operators are higher in IntroClass benchmark than in Codeflaws benchmark. This suggests that a repair tool could use a smaller set of repair operators to fix defects in IntroClass benchmark.

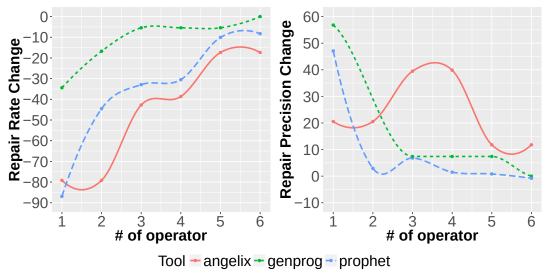
3.3.5 RQ3: Selection Strategies

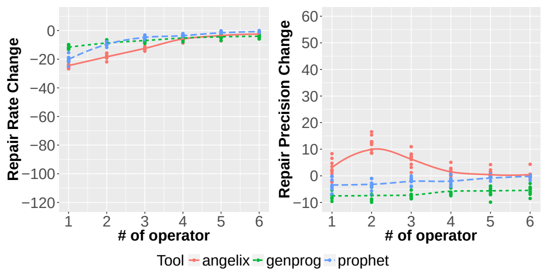
Although an individual operator could achieve relatively high repairability, it is often not able to address all classes of defects. We propose three selection strategies to choose a subset of repair operators that are effective. Our goal is to select a subset of repair operators that could achieve high repair precision without sacrificing repair rate.

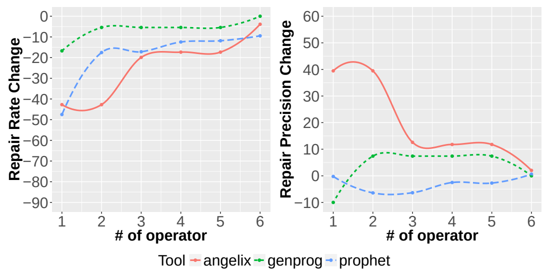
The three selection strategies include:

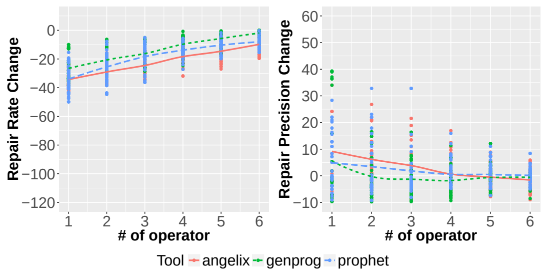
Highest Score (best-k): Given k=1, ..., 6, the best-k strategy chooses a subset of operators consists of k repair operators with the best score. We calculate the score of each operator based on the weighted sum of its repairability, reliability and expressiveness:


(a) △ Repair Rate vs △ Repair Precision for *best-k* on Codeflaws


(b) △ Repair Rate vs △ Repair Precision for *best-k* on IntroClass


(c) △ Repair Rate vs △ Repair Precision for *most-exp* on Codeflaws


(d) △ Repair Rate vs △ Repair Precision for *most-exp* on IntroClass


(e) △ Repair Rate vs △ Repair Precision for *random* on Codeflaws

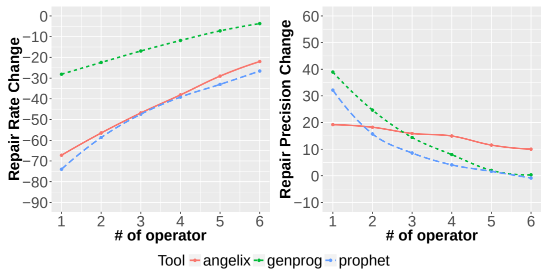

(f) △ Repair Rate vs △ Repair Precision for *random* on IntroClass

Figure 3-4: Overall Repair Rate Change vs Overall Repair Precision Change with various strategies

Definition 5 *The score of a repair operator op is:*

$$\begin{aligned} Score(op) = {} & \alpha * Repairability(op) + \\ & \beta * Reliability(op) + \\ & \gamma * Expressiveness(op) \end{aligned} \tag{3.1}$$

Most expressive (most-exp): Given k=1, ..., 6, the most-exp strategy chooses a subset of operators with k repair operators that have the highest expressiveness score for a particular repair tool.

Random Selection (random): Given k=1, ..., 6, the random selection strategy chooses a subset of operators consists of k repair operators based on random selection among the set of all operators for a particular repair tool.

We measure the effectiveness of different selection strategies by computing the overall repair rate change and the overall repair precision change of the chosen subset. For the best-k strategy, we tune the values for α,β, and γ by varying these values in the Codeflaws benchmark. We use $\alpha = 0.2$, $\beta = 0.45$ and $\gamma = 0.35$ for our experiment to give higher priority to reliability.

Definition 6 *Given a set S,* ***repair rate*** *is the ratio of plausible fixes among all evaluated defects.* ***Repair rate change*** *($\triangle$RepairRate) between the selected subset (sub) of operators and the original set of operators (orig) is the difference in the probability of obtaining plausible fixes.* ***Repair precision*** *is the average ratio of the number correct fixes for a defect over all fixes for that defect. The* ***repair precision change*** *($\triangle$RepairPrecision) between sub) and orig) is the change in the average probability of obtaining correct fixes.*

$$RepairRate_S = \frac{\#plausible\ fixes\ for\ S}{\#defects}$$

$$\triangle RepairRate(\%) = \frac{RepairRate_{sub} - RepairRate_{orig}}{RepairRate_{orig}}$$

$$RepairPrecision_S = \frac{\sum\limits_{d \in defects} \frac{\#correct\ fixes\ for\ S\ for\ d}{\#plausible\ fixes\ for\ S\ for\ d}}{\#fixed\ defects\ forS}$$

$$\triangle RepairPrecision(\%) = \frac{RepairPrecision_{sub} - RepairPrecision_{orig}}{RepairPrecision_{orig}}$$

We distinguish repairability/repair rate and reliability/repair precision because repairability and reliability are properties of a repair operator, while repair rate and repair precision measure the effect of a set of operators over the entire patch spaces. Note that a positive value for the repair rate change and the repair precision change indicates an increment for a measurement, whereas a negative value denotes a decrease of the measurement.

The *best-k* strategy assumes that repair operators that are effective individually could form a combined subset that has high overall effectiveness. As the *best-k* strategy consider all measurements of effectiveness, it is more likely to achieve a balance for the trade-offs between repair rate and repair precision. Although the *most-exp* strategy is the same as the *best-k* strategy that ignores repairability and reliability, we evaluate the *most-exp* strategy to determine the effect of defect types on the trade-offs between repair rate changes versus repair precision changes. The *random* strategy serves as the baseline for comparing other strategies. To study the general impact of applying various selection strategies across different benchmarks, we compute all different subsets using Codeflaws benchmark and compare their effects on both benchmarks.

Figure 3-4 shows the effectiveness of different strategies. The x-axis (# of operator) of all the subfigures in Figure 3-4 represents the size of the subset (k), whereas the y-axis denotes the relative changes of different measurement (repair rate changes and repair precision changes). Each color in all the figures represents a repair tool (Angelix, GenProg and Prophet). The individual dots in Figure 3-4 indicates the results for each fold in the 10-fold cross validation, while the different lines shows the overall trend line as computed by loess regression, a smoother based on polynomial regression [57].

Figure 3-4a shows the effect of selecting a subset of repair operators using *best-k* strategy on Codeflaws. The first graph in Figure 3-4a shows that adding more operators (increasing value of k) with *best-k* strategy increases the overall repair rate. Meanwhile, the second graph in Figure 3-4a shows that adding more operators causes a decrease in the overall repair precision. Figure 3-4a also shows that choosing less than three repair operators leads to an increase in overall repair precision and a decrease in overall repair rate. However, adding more than three operators is less beneficial since the increase in repair precision cannot compensate for the decrease in repair rate. Figure 3-4b demonstrates that a similar trend can be observed in IntroClass.

Figure 3-4c illustrates the overall effectiveness with *most-exp* strategy on Codeflaws, whereas Figure 3-4d demonstrates the change of effectiveness in IntroClass. It can be seen from Figure 3-4c that using the two most expressive repair operators in Angelix causes a decrease in the repair rate (around -15%) but induces a similar magnitude of increment in repair precision (+15%). Similarly, Figure 3-4d shows that selecting less than three repair operators in IntroClass is sufficient for Angelix to increase repair precision without sacrificing repair rate.

One notable difference between the results on the benchmarks is that all repair tools perform differently in both benchmarks when applying *best-k* strategy and *most-exp* strategy. This difference is due to (1) the diversity of defects is different in both benchmarks with Codeflaws containing more diverse defects than IntroClass and (2) many repair operators are redundant in IntroClass. As the *most-exp* strategy selects repair operators based on the number of repair classes, it is more resilient to different defect types across various benchmarks.

Figure 3-4e shows the effectiveness changes for the *random* strategy. It can be seen from Figure 3-4e and Figure 3-4f that both the repair rate changes and the repair precision changes fluctuate. More importantly, these figures demonstrate that the *random* strategy may lead to a decrease in all aspects (repair rate and repair precision). *This suggests that developers of future repair tools should be careful in their selection of the set of operators as random selection may lead to decrease in overall effectiveness.*

3.4 Discussion

Number of supported repair operators in repair tools. Figure 3-2 shows that the reliability for individual repair operator is significantly low (p-value $<$ 0.05) in Prophet than in GenProg. This is because Prophet supports more repair operators than other program repair tools. By supporting 10 repair operators (Table 3.1), the design of Prophet strike to achieve higher repair rate which may cause a decrease in overall repair precision (as indicated by the low reliability of individual repair operator in Prophet). This observation confirms with our hypothesis that using reduced set of repair operators could achieve a balance between changes in repair rate and changes in repair precision.

Higher Order Repair Operators. Our results demonstrate that higher order repair operators (HIMS and HEXP) contributes as much to the overall effectiveness of repair tools as other repair operators that involve single-line modifications. Particularly, Figure 3-2c reveals that the HIMS (multiple insertions) operator has significantly higher reliability (p-value$<$0.05) than other repair operators in Codeflaws benchmark, while the HEXP (multiple replacements) operator is effective in IntroClass benchmark.

The importance of identifying redundant operators. Our empirical evaluation in Section 3.3.4 reveals the redundancy of repair operators in existing repair tools. We think that identifying redundant repair operators are important because (1) including these operators could pollute the patch spaces such that repair tools will need to spend more time in exploring the patches generated by these repair operators, (2) redundant operators may also lead to decrease in overall repair precision if these operators are prioritized before other more useful operators; and (3) redundant repair operators represent rare defect classes since these removing these repair operators do not have any impact on the overall repair rate and repair precision. For example, we observe that SDME (insertion of initialization) is redundant in both Codeflaws and IntroClass benchmarks. In fact, this redundancy is similar in the GenProg benchmark [141] where only two correct patches out of 105 evaluated defects (less than 2% of defects) could be generated using the SDME operators [94].

Diversity of defects in benchmarks. If we compare the redundancy of repair operators across the two benchmarks, the redundancy of repair operators are higher in IntroClass benchmark than in Codeflaws benchmark. It can be seen from Figure 3-3 that only a small subset of repair operators are needed from each repair tool to fix the defects in IntroClass benchmark. We think this difference is because programs in IntroClass benchmark has less diverse defects compared to programs in Codeflaws benchmark. *The high redundancy of repair operators and the lack of diversity in IntroClass benchmark provide concrete evidence that the selection of benchmarks are important for the thorough study of the effectiveness of program repair tools*

Effectiveness of different selection strategies. In Section 3.3.5, we introduce three selection strategies (*best-k*,*most-exp* and *random*) which behaves differently with varying subset sizes. It is obvious from Figure 3-4e and Figure 3-4f that random selection of repair operators may lead to decrease in overall effectiveness. As our empirical evaluations shows that the *most-exp* strategy performs better for Angelix in IntroClass benchmark and as the expressiveness score are easier to compute, we recommend using the *most-exp* strategy for benchmark with less diverse defects. Meanwhile, the *best-k* strategy takes into account all three aspects of the effectiveness of repair operators (repairability, reliability and expressiveness), this strategy allows developers of repair tools to design their set of repair operators according to the preferred trade-offs between changes in repair rate versus changes in repair precision.

3.5 Threats to Validity

We acknowledge the following threats to validity of our results:

Set of Repair Operators. We define the set of repair operator in Table 3.1 to facilitate comparisons between repair operators supported by different repair tools. As it is non-scalable to manually classify over 3000 defects, we categorize each repair operator automatically by modifying GumTree's AST differencing [66]. The classification is based on the mutant operators designed for the C programming language [26]. While it may be interesting to check whether the fix ingredients for the insertion of new code exists within the same program (assumption made by GenProg), our classification only consider the syntactic structure of each patch instead of the expressions used in the patch. The set of repair operator may be different from the repair operator stated by developers of repair tools. For example, although GenProg AE supports only two mutation operations (insertion and deletion of statements), we identify seven repair operators in Table 3.1 because our set of operators consist of program transformations at the AST-level. We believe that having a more fine-grained set of repair operators allow more thorough investigation of their effectiveness.

Our Measurement of Effectiveness. We measure effectiveness of repair operator in terms of repairability, reliability and expressiveness. Although our measurement of effectiveness takes into account more dimensions of effectiveness than previous study [91], there may be other dimensions not studied in this chapter that affect the effectiveness of repair operators.

Effectiveness of Program Repair Tools. As we only empirically assess the effectiveness of three publicly available tools (GenProg, Prophet and Angelix), our results are restricted to these three tools. Nevertheless, the measurement that we derived for computing the effectiveness of repair operators and the selection strategies that we proposed could be used as guidelines for evaluation of future program repair tools.

Generality of our results beyond the evaluated benchmarks. As we only compare the effectiveness of repair tools in the Codeflaws benchmark and the IntroClass benchmark, the result may be different for other benchmarks. Nevertheless, our evaluated benchmarks are diverse and contain large number of defects.

Repair operator selection strategies. Our proposed selection strategies show the trade-off between repair rate and repair accuracy. While the dynamic selection of repair operators may help to balance this trade-off, we leave the derivation of other selection strategies as future work.

3.6 Chapter Summary

This chapter present the first study that investigates the effectiveness of repair operators supported by current program repair tools. Our empirical evaluation takes into account the impact of different defect types to the effectiveness of repair tools and identifies repair operators that are effective in each repair tool. Specifically, the most effective repair operator in GenProg is the SISA (insert assignment) operator, while the most effective operator in Prophet is DRVA (variable replacement); and the most effective operator in Angelix is ORRN (replace relational/logical). Our study also reveals that there exists redundancy in the set of repair operators used by existing program repair tools. Future developers of repair tools should be aware of the redundancy in their set of repair operators as these redundant operators could be assigned lower priority to avoid polluting the patch spaces.

Moreover, we proposed and compared three selection strategies (*best-k*,*most-exp* and *random*) which allow developers of future repair tools to make explicit design decision according to their preferred trade-offs between repair rate and repair accuracy. Our study indicates that random selection of the set of repair operators may cause decrease in the overall effectiveness of repair tools. Our empirical assessment of the effectiveness of repair operators in existing program repair tools aim to provide guidance for future design of repair operators.

Chapter 4

A Correlation Study between Automated Program Repair and Test Suite Metrics

Mutation testing is a technique that evaluates the quality of a given test suite by applying small changes to a program and checking if the test suite could detect the introduced changes [118]. The concept of automated program repair techniques is innately related to the idea of mutation testing in several aspects. First, as most automated repair techniques rely on test cases for ranking and validating each automatically generated patch, mutation testing could be used to assess the quality of the test suite used by program repair systems. Second, although *mutation operators* in mutation testing are program transformations designed specifically for simulating artificial defects, these operators have been successfully applied for generating fixes automatically [62]. This chapter focus on the first aspect — investigating the relationship between the test suite quality and the quality of the automatically generated patches where several metrics (including metrics in mutation testing) are used to assess test suite quality.

4.1 Introduction

In recent years, existing automated program repair approaches have showed initial success in generating fixes for real-world large-scale software such as the PHP interpreter and the well-known Heartbleed vulnerability in OpenSSL [82, 87, 108, 117, 126, 141, 143]. Meanwhile, the research focus is gradually shifting from the feasibility of automated program repair to the quality of generated patches [38, 95, 127, 134]. In particular, these latest research results raise a question about how to generate a *correct* patch—a patch that not only passes all tests available to a repair system, but also indeed fixes the bug. Most of the automated program repair approaches use a test-suite as a proxy of software specification, since formal specification is hardly used in the industry. While the fact that software tests are widely available is advantageous, a test-suite is merely incomplete specification that may make a generated repair incomplete. In general, there is no guarantee that an automatically generated repair will not cause new regressions. This problem of automated program repair is akin to the problem of software testing in that even if all available tests pass, there is generally no guarantee that no other new tests will fail the software under test. Despite this limitation, it is possible to improve software quality by improving the quality of a test-suite. Likewise, *is it possible to control the quality of automatically generated repair by controlling the quality of a test-suite?* This is our key high-level research question we aim to answer in this chapter. Apart from this main research question, we also investigate how test-suite metrics affect repairability (repair success rate) and repair time.

To answer our main research question, we conduct large-scale experiments about the correlation between test-suite quality and automated program repair. Our subject programs contain four large-scale real-world programs such as a PHP interpreter and a TIFF image processing library, in addition to a well-known benchmark, SIR [65]. In comparison, previous studies were conducted with small student programs or SIR subjects. As a result, we can provide stronger empirical evidences about the correlation between the quality of test-suites and the quality of automatically generated repairs than previous studies. Also, we for the first time compare various test-suite metrics such as statement coverage, branch coverage, test-suite size, and mutation score, focusing on their degrees of correlation (i.e., correlation coefficients) with repair quality. As a result, we can answer the question whether the traditional test suite metrics proposed for the purpose of software testing are also useful in the context of automated program repair, and which test-suite metric is most effective.

In terms of the quality of automatically generated repairs, we focus on the *reliability of a generated repair*, that is, whether regressions occur in a repair. Judging whether a repair is correct is often subjective and difficult to be automated in the absence of formal specifications. Previous studies investigate the reliability of repairs instead, because checking whether a generated repair causes regressions can be performed in an automated way [38, 80, 83, 134]. That is, once a repair is generated, this repair can be tested with a test-universe (held-out test-suite) which contains tests that were not given to the program repair system. If a failing test is found in the test-universe, the repair is considered incorrect as it causes regressions. As in previous studies, we also similarly investigate how often regressions occur to measure the quality of a repair. Meanwhile, we obtain automatically generated repairs by running GenProg [90, 141]. In total, we collected 3818 repairs from 142 buggy versions of 10 different programs of various sizes (173–1046K LoC), using 14600 randomly sampled test suites. We sample test suites from the whole test cases available in our subjects. While we retrieve the main results from GenProg-generated repairs, we also conduct smaller scale experiments with another repair tool

SemFix [117] to see whether our main results extend beyond GenProg. GenProg and SemFix are first search-based and constraint-based repair tools, respectively. Search-based repair tools navigate a set of repair candidates through a search algorithm until a repair is found, while constraint-based repair tools first construct repair constraints that should be satisfied by a repair and symbolically search for a repair satisfying the repair constraint using a theorem prover. While our experiments may not generalize to all other repair tools, GenProg, the repair system we mainly use in our study, has been used in many previous studies on automated program repair [83, 87, 88, 127, 134, 141]. Our experimental results obtained from GenProg complement the results from earlier studies.

Our results show that in general, the traditional metrics of test-suites, that is, statement coverage, branch coverage, test-suite size, and mutation score, are *negatively correlated* with the likelihood that a generated repair causes a regression. In other words, as the traditional metrics of a test-suite increase, generated repair tend to cause regressions less often. Our result implies that the traditional test suite metrics proposed for software testing can also be used for automated program repair. Among the test-suite metrics we investigate, statement coverage is shown to be most strongly correlated with regression ratio. A practical implication is that to reduce regression ratio, increasing statement coverage is likely to be more effective than improving the other test-suite metrics such as branch coverage. However, it should be noted that the highest correlation of statement coverage does not necessarily imply that a statement coverage-adequate test-suite is better than a branch coverage-adequate test-suite.

In summary, our main contributions are:

- We for the first time conduct a correlation study of automated program repair with various test-suite metrics such as statement coverage, branch coverage, test-suite size, and mutation score. According to our study, traditional test-suite metrics proposed for software testing are negatively correlated with the likelihood that a generated repair causes regressions. Therefore, improving a test-suite based on traditional test-suite metrics is beneficial both for software testing and automated program repair. Among test-suite metrics we investigate, statement coverage is shown to be most strongly correlated.

- We conduct the largest experiments to date about the correlation between test-suite quality and the effectiveness of automated program repair (in particular, the reliability of repairs). Our subject programs contain four large-scale real-world programs. Our experimental results provide strong empirical evidences that the repair quality problem is indeed quite severe (the average regression ratio of 3818 repairs we obtained from GenProg is 40%), and traditional test suite metrics can be used to control the quality of automatically generated repairs.

Apart from our main contributions, we also report other noteworthy results in this chapter. We for the first time investigate the correlation between mutation score and repair quality (regression ratio of repairs). Despite the conceptual similarity of automated program repair to mutation testing, the correlation of mutation score with repair quality is not observed to be stronger than the correlation of coverage-based metrics. Our new mutation-based metric, capable-tests ratio, is observed to be more strongly correlated with the reliability of repairs in real-world subjects than mutation score. We also investigate how test-suite metrics affect repairability (repair success rate) and repair time. While we could not find a correlation pattern consistent across all subjects, positive correlations between repairability and test-suite metrics (as test-suite metrics increase, repairability increases) and negative correlations between repair time and test-suite metrics (as the test-suite metrics increase, repair time decreases) were observed in some subjects.

4.2 Background

In this section, we first provide the background for the correlation coefficients used for our correlation study (Subsection 4.2.1). Next, in Subsection 4.2.2, we describe the relationship between mutation testing and automated program repair as this relationship motivates our study on how mutation scores of test suite used for repair affect automatically generated patches.

4.2.1 Correlation Coefficient

A correlation coefficient measures the statistical relationship between two variables. Pearson Product-Moment Correlation Coefficient is a widely-used statistical formula that measures the strength of the linear relationship between two variables [122]. Kendall rank correlation coefficient measures statistical associations based on the rankings of the data [81].

To check whether the correlation coefficient is statistically significant, statistical hypothesis testing is performed. The null hypothesis for the Pearson's correlation coefficient and Kendall rank correlation coefficient is:

Null hypothesis H_0 : $\rho = 0$ (the correlation coefficient is 0)

Alternative hypothesis H_1 : $\rho \neq 0$ (the correlation coefficient is not 0)

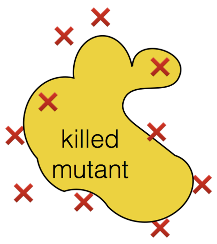
In statistical hypothesis testing, the p-value or probability value is often used to determine whether to reject the null hypothesis. If the p-value is less than or equal to the significance level α, we reject the null hypothesis in favor of the alternative. The commonly used significance level is $\alpha = 0.05$. The p-value for Pearson's correlation coefficient is computed using the t-distribution: $t=\frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$ where r denotes the correlation coefficient and n denotes the number of observations. For Pearson's coefficient, a small p-value indicates that the null hypothesis is false, and we can conclude that the correlation coefficient is different from zero and that a linear relationship between the two variables exists. Meanwhile, for the Kendall's correlation coefficient, a variation of the correlation coefficient, the Kendall's tau-b (τ_b) coefficient is used to handle data samples with ties rank. The Kendall's tau-b coefficient is computed using the formula: $\tau_B = \frac{\#\ of\ concordant\ pairs - \#\ of\ discordant\ pairs}{\sqrt{N_1} \cdot \sqrt{N_2}}$ where N_1 denotes the number of data pairs not tied in a target feature and N_2 denotes the number of data pairs not tied in another target feature. For Kendall's coefficient, we test the null hypothesis that $\tau_B = 0$. If the p-value is smaller than α, then the null hypothesis is false, and we can conclude that the ordinal association between the two variables exists.

4.2.2 On Duality between Mutation Testing and Automated Program Repair

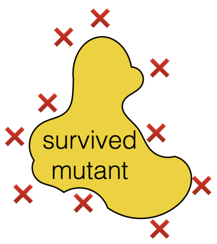
There is not only similarity between automated program repair but also duality between mutation testing and automated program repair. As pointed out in [141], "our confidence in mutant testing increases with the set of *non-redundant mutants* considered, but our confidence in the quality of a program repair gains increases with the set of *non-redundant tests*." Note that mutation score measures the non-redundancy of killed mutants, not the non-redundancy of tests capable of killing mutants. We introduce a new metric called *capable-tests ratio* in the next section that measures the non-redundancy of capable tests.

4.3 Research Questions

The key high-level research question of this study is whether it is possible to control the quality of automatically generated repair by controlling the quality of a test-suite. To address this question quantitatively, we investigate the *correlation* between the quality of a test-suite and the quality of an automatically generated repair. If a positive correlation is found, this means that using a high-quality test



(a) A killed mutant



(b) A survived mutant

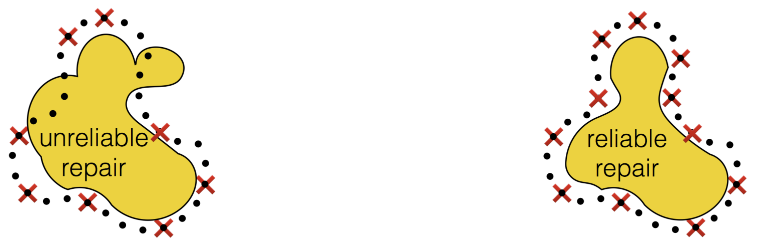
Figure 4-1: The cross marks represent the tests in a test-suite. Overlapping with a cross mark means that the mutant fails the corresponding test.

suite is likely to lead to a high-quality repair. Thus, our first research question is:

Research Question 1: Is there a positive correlation between the quality of a test-suite and the quality of an automatically generated repair?

However, this research question should be refined, because it does not state how to measure the quality of a test-suite and the quality of a generated repair. We first describe how we measure them, before refining the research question.

Measuring Test-Suite Quality with Traditional Metrics and Capable-Tests Ratio. We measure the quality of a test-suite using five kinds of test-suite metrics: (1) statement coverage, (2) branch coverage, (3) test-suite size, (4) mutation score, and (5) capable-tests ratio. All metrics except for the last one are traditional test-suite metrics. We introduce the last metric — capable-tests ratio — to complement a potential shortcoming of the mutation score which will be described shortly. Figure 4-1 pictorially describes mutation testing. The tests in a provided test-suite (represented with cross marks in the figure) guard the program against regression-causing changes (represented with a mutant that overlaps with a cross mark). We hypothesize that the more the provided test-suite `TS` contains tests that kill one of mutants (we call such tests *capable tests*), the more likely `TS` can prevent regression-causing repairs. However, mutation score does not measure the percentage of capable tests in a test-suite; it only shows the percentage of killed mutants, and adding or removing tests killing no mutant does



(a) An unreliable repair (b) A reliable repair

Figure 4-2: The dots represent the tests in the test-universe, while the cross marks represent the tests in a test-suite used to guide automated program repair. Crossing over a dot or a cross mark means that the repair fails the corresponding test.

not change the mutation score of the test-suite. To complement this shortcoming of the mutation score, we introduce a new metric *capable-tests ratio* defined as following:

$$\text{Capable-tests Ratio}(\texttt{TS}) = \frac{\text{number of capable tests in } \texttt{TS}}{\text{total number of tests in } \texttt{TS}}$$

Namely, capable-tests ratio of a test-suite TS is the ratio of the number of capable tests in TS, that is the number of tests that kill at least one mutant, over the total number of tests in TS.

Measuring Repair Quality. Meanwhile, we measure the quality of repair from the perspective of reliability. We deem a repair R to be reliable if there is no regression detected when testing R with its test-universe.[1] Note that the test-universe is the superset of a test-suite used to drive automated repair. In Figure 4-2, both repairs (the shaded areas) are valid because they pass all tests in a given test-suite, represented with the cross marks in the figure. However, the one in Figure 4-2a is unreliable because it fails some tests in the test-universe (the dots).

We evaluate the reliability of repairs through *regression ratio*, which is computed as the number of regression-causing repairs over the total number of

[1]Only positive tests are considered; an output change for negative tests is not a regression.

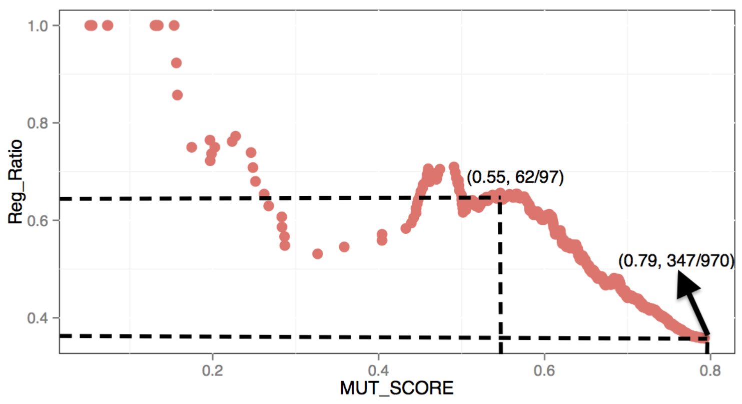


Figure 4-3: A scatter plot that illustrates the correlation between the mutation score (the MUT_SCORE axis) and the regression ratio (the Reg_Ratio axis). Coordinate (0.55, 62/97) of the plot denotes the following: (1) among test-suites with which repairs are successfully generated, there are 97 test-suites whose mutation scores are less than 0.55; (2) out of these 97 cases, a regression is detected in 62 cases. Similarly, coordinate (0.79, 347/970) indicates that there are 970 test-suites whose mutation scores are not greater than 0.79, and regressions are detected in 347 cases.

repairs obtained with the test-suites under investigation. Note that for each pair of a test-suite `TS` and a repair `R` generated with `TS`, we record whether regression is observed in `R` when tested against the test-universe. Our primary goal in this paper is to examine the correlations between the reliability of repairs and various test-suite metrics. We compute regression ratio at each metric score as follows. First, we collect repairs, each of which is generated with a test-suite whose metric is not greater than the score under investigation; for example, to compute the regression ratio at mutation score 0.5, we collect repairs generated with test-suites whose mutation scores are not greater than 0.5. Subsequently, we proceed to count how many of these repairs cause regressions when tested with the test-universe. Formally, the following formula is used to calculate the regression ratio at metric score `s`.

$$Reg\_Ratio(s) = \frac{|\{\mathtt{TS}|repaired(\mathtt{TS}) \wedge metric \leq s \wedge regression(\mathtt{TU})\}|}{|\{\mathtt{TS}|repaired(\mathtt{TS}) \wedge metric \leq s\}|}$$

where TS and TU represent a test-suite and a test-universe, respectively (note that TS $\subseteq$ TU). In the formula, predicate repaired(TS) denotes that a repair is successfully generated within the timeout when TS is used to guide automated program repair. Another predicate regression(TU) means that a regression error is observed when testing the obtained repair with TU (i.e., regression(TU) is true if there is a test $t \in$ TU $\setminus$ TS for which the obtained repair fails). By tracking the regression ratio at different metric scores, we can retrieve the correlation between the regression ratio and a test-suite metric. Note that the lower the regression ratio is, the higher reliability of repairs.

Figure 4-3 shows how the regression ratio (the Reg_Ratio axis) changes as the mutation score (the MUT_SCORE axis) changes in one of our subjects, tcas. For example, among test-suites with which repairs are successfully generated, there are 97 test-suites whose mutation score is not greater than 0.55, and a regression is detected in 62 cases out of those 97 cases. By increasing the mutation score threshold to 0.79, we can consider 873 more test-suites, in the majority of which a regression is not detected, as evidenced by a lower regression ratio there (347/970).

Now that we described how we measure the quality of a test-suite and a repair, we now refine our Research Question 1 as follows:

Research Question 1 (Refined): Is there a *negative correlation* between the *metrics* of a test-suite and the *regression ratio* of automatically generated repairs? In other words, are generated repairs *less likely* to cause regressions, as test-suite metrics increase?

Apart from showing a general tendency about how test-suite metrics are associated with the quality of repairs, correlation analysis can also be used to illustrate which test-suite metric is most strongly associated with the quality of repairs, by comparing the correlation coefficients of different test-suite metrics. We thus ask the following research question.

Research Question 2: Which test-suite metric is most strongly correlated with the regression ratio of automatically generated repairs?

Answering this research question can be practically important. Imagine the scenario where a test-suite available for automated program repair is neither statement coverage-adequate nor branch coverage-adequate. In this scenario, in terms of increasing the likelihood of obtaining a regression-free repair, would it be more beneficial to improve statement coverage or branch coverage? It would be more cost-effective to improve a test-suite metric which is more strongly negatively correlated with the regression ratio of repairs.

Meanwhile, a higher-quality test suite may make it more difficult for a program repair tool to generate a repair as the repair needs to satisfy stricter constraints imposed by the test suite. We therefore evaluate whether the repairability of automated program repair is negatively correlated with test-suite metrics.

Research Question 3: Is there a *negative* correlation between the *metrics* of a test-suite and the *repairability* of automated program repair? In other words, would repairability be sacrificed in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

Similar to our measurement of regression ratio, we compute repairability at each metric score. First, we collect test-suites, each of which has a metric not greater than the score under investigation. For example, to compute the repairability at mutation score 0.5, we collect test-suites whose mutation scores are not greater than 0.5. Then, we proceed by calculating the number of these test-suites that leads to a repair within the time budget. Formally, we use the following formula to compute repairability at metric score `s`.

$$\mathit{Repairability}(s) = \frac{|\{\mathtt{TS} | \mathit{metric} \leq s \wedge \mathit{repaired}(\mathtt{TS})\}|}{|\{\mathtt{TS} | \mathit{metric} \leq s\}|}$$

Table 4.1: Subjects of our experiments

Subject	LOC	Versions	Test-Universe Size	Test-Suites	Test-Suite Size
tcas	173	41	1608	4100	1–100
tot_info	565	23	1052	2300	1–100
print_tokens	726	7	4130	700	1–100
print_tokens2	570	10	4115	1000	1–100
schedule	412	9	2650	900	1–100
schedule2	374	9	2710	900	1–100
php	1046K	21	200	2100	1–100
libtiff	77K	11	78	1100	1–78
grep	9.4K	5	1582	900	1–100
findutils	18K	6	82	600	1–82
Total		142	18207	14600	

where predicate `repaired(TS)` denotes that a repair is successfully generated within one hour when test-suite `TS` is used as input for an automated program repair tool.

Apart from repairability, the quality of a test suite may also affect *repair time* (i.e., the time taken to generate a repair). We thus ask the following similar research question.

Research Question 4: Is there a *negative* correlation between the *metrics* of a test-suite and repair time? In other words, would more time be spent in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

4.4 Experimental Methodology

We perform our evaluation on 10 C programs obtained from various benchmarks, including the Software-artifact Infrastructure Repository (SIR) [65] benchmark, the GenProg benchmark [87] and CoREBench [48]. We evaluate these subject programs on GenProg and SemFix.

4.4.1 Subjects, Test-Universes and Test-Suites

Table 4.1 shows the information for the 10 C programs evaluated in our experiments. The first six subjects in Table 4.1 are obtained from the SIR benchmark, whereas the four subjects at the bottom are non-SIR subjects. The column "LOC" shows the lines of code, while the column "Versions" denotes the number of buggy versions for each subject program. Meanwhile, the column "Test-Universe Size" indicates the number of tests in the test universe of each subject, whereas column "Test-Suites" denotes the number of test suites constructed by sampling the test universe, and the column "Test-Suite Size" shows the number of tests in the constructed test-suites. As shown in the "LOC" column in Table 4.1, subject programs used in our experiments are of various sizes (ranging from 173 LOC to 1046K LOC).

Our subjects consist of six well-known Siemens programs collected from SIR benchmark [65], two real-world programs (`php` and `libtiff`) previously used to evaluate GenProg [87, 141],[2] and another two real-world programs (`grep` and `findutils`) obtained from CoREBench [48].[3] We choose these real-world subjects because they have large number of tests and multiple buggy versions, and contains more versions that could be automated fixed. In particular, in `php` and `libtiff`, GenProg is able to generate repairs in many buggy versions in previous study [87]. Similarly, buggy versions of `grep` and `findutils` could be repaired by GenProg in our pilot experiment. The "Versions" column in Table 4.1 shows that our subjects contain a total of 142 buggy versions, among which the six SIR subjects have 99 buggy versions and the four non-SIR subjects have 43 buggy versions.

Each of our subjects has relatively large number of tests, which is our test-universe. The test-universes of our large subjects consist of developer-written tests. Assuming that these tests are well-maintained, the tests in the test-universe are likely to be different from each other. In other words, the risk that some tests are identical to each other—which is undesirable because then, the training test-suite

[2]We used the original GenProg benchmark. At the time of writing this paper, the benchmark was updated after a few problems in the test scripts of php and libtiff are reported in [127].
[3]The grep subject in CoREBench contains real errors unlike the grep versions in SIR that contains seeded errors.

and the held-out test-suite can have the same test—is low. Meanwhile, the test-universes of our small subjects are extracted from the SIR benchmark [65], which are generally considered high-quality and were used in many prior studies [32, 78].

The "Test-Universe Size" column of Table 3.3 shows the total number of tests in the test-universe of each subject.[4] We construct a large number of test-suites by randomly selecting tests from these test-universes (without replacement) at each test-suite-construction iteration.

The "Test-Suites" column shows how many test-suites we constructed for each subject. For each buggy version of a subject, we constructed 100 test-suites such that it contains at least one failing test case. One exception is that we constructed 180 test-suites in `grep`. We constructed more test suites in `grep` to compensate for the fact that `grep` contains the smallest number of buggy versions (five versions), which means that less number of repairs could be obtained from these buggy versions. The size of each test-suite is chosen uniformly at random between 1 and 100, except for in `libtiff` and `findutils` where the maximum size is the size of the test-universe, that is, 78 and 82, respectively. Note that in our study, we compare experimental results across test-suite metrics, not across subjects. In each subject, we compute diverse test-suite metrics for each constructed test-suite. In total, we prepare 14600 test-suites.

We acknowledge that our simple random test-suite construction method in itself does not distinguish the effect of test-suite size and the effect of other test-suite metrics such as coverage (coverage tends to increase as the size of the test-suite increases). An alternative more controlled test-suite construction method is to construct a set of test-suites of identical size with different levels of coverage, and similarly a set of test-suites of identical coverage with different sizes, although prior study [115] reported that it is difficult to obtain such a more ideal set of test-suites.

[4]While php contains 8471 tests, we randomly selected 200 tests out of them due to the long running time of the php tests.

To compensate for the shortcoming of our test-suite construction method, we perform ANCOVA (analysis of covariance) and separate out the effect of coverage, similar to the prior work [115].

4.4.2 Automated Repair Algorithm

Our main experiment focuses on generating repairs using GenProg [87, 141] as it is often used in previous studies [83, 134]. We fed GenProg with the 14600 test-suites that we prepared. When running GenProg, we use almost the same configuration parameters as those that were used in an earlier GenProg experiment [87]. One noteworthy difference is that we use the deterministic repair algorithm of GenProg [141] to minimize randomness during experiments. All experiments were performed by distributing the load on 10 machines, each of which has two Intel Xeon E5520 2.2GHz processors and 24GB of main memory. To obtain a large number of repairs, which is essential for our study, we use a relatively short timeout of one hour. In total, we obtained 3818 repairs.

In addition to running GenProg, we also conducted smaller scale supplementary experiments with another repair tool (SemFix [117]) to investigate whether our results extend beyond GenProg. We chose SemFix because the repair algorithm of SemFix is fundamentally different from that of GenProg. Essentially, SemFix extracts from the runs of a test-suite a set of constraints in the form of logical formulas, and subsequently solves these constraints to obtain a repair. This deductive style of repair of SemFix is in contrast to GenProg's generate-and-validate approach; GenProg repeats the iteration of generating a repair candidate and validating it until a repair is found.

In our experiments with SemFix, we used the same test-suites as used for our main experiments with GenProg. We collected repairs using SemFix from the same SIR subjects as used in our main experiments except for `tot_info` which does not work with the current version of SemFix.[5] Meanwhile, testing our non-SIR subjects requires running non-trivial test-scripts written in scripting languages. To deal with these non-SIR subjects with the current version of SemFix, it is necessary to transform these test-scripts into corresponding C program statements. This is because SemFix extracts logical formulas through a symbolic-execution tool, KLEE [51] that currently cannot handle scripting languages. Still in an attempt to deal with a large subject at least partially, we manually transformed the test-scripts of 4 versions of `libtiff` (i.e., 01209c9, 3af26048, d13be72c, and 0661f81).

4.4.3 Measuring Test-Suites Metrics

It is well known that computing mutation score typically takes a long time due to the high volume of mutants. Each and every mutant should be tested with the test suite under investigation, resulting in running the same test repeatedly for different mutants. For large programs, obtaining mutation score is particularly challenging because there are too many mutants to be tested within a reasonable time budget. To alleviate the problem, it is customary to sample parts of the mutants, and compute the mutation score only with the sampled mutants.

We measure the mutation score and the capable-tests ratio of each test-suite using Proteum [99].[6] To deal with enormous size of mutants generated from the 4 large subjects (`php`, `libtiff`, `grep`, `findutils`), we randomly sampled 1 – 3% of the total mutants, using the options Proteum provides. Although we do not distinguish equivalent mutants, note that the same mutant samples of program P are used across all test-suites for P in our experiments. Thus, the mutation scores of these test-suites are affected at the same rate by equivalent mutants that may exist, making the correlations between mutation scores of these test-suites and the reliability of repairs unaffected accordingly. Meanwhile, to measure the statement

[5]tot_info includes non-linear arithmetic expressions which are not currently supported by the underlying SMT solver SemFix uses.

[6]We extended its parser to handle the large subjects (php, libtiff, grep, and findutils).

Table 4.2: GenProg experiments: statistics for repairs and regressions

Subjects	Test-Suites	Repairs	Repair Ratio	Regressing	Regression Ratio
tcas	4100	972	24 %	348	36 %
tot_info	2300	137	6 %	17	12 %
print_tokens	700	28	4 %	0	0 %
print_tokens2	1000	235	24 %	9	4 %
schedule	900	37	4 %	37	100 %
schedule2	900	108	12 %	53	49 %
php	2100	1666	79 %	915	55 %
libtiff	1100	313	28 %	42	13 %
grep	900	128	14 %	41	32 %
findutils	600	194	32 %	83	43 %
Total	14600	3818	26 %	1545	40 %

and branch coverage of our test-suites, we use `gcov`.[7] When running GenProg or SemFix, it is necessary to mark the source file(s) allowed to be repaired. Our measurements of mutation score and statement/branch coverage are performed on these marked files.

4.5 Experimental Results

In this section, we outline the results from our experiments with the repair tools GenProg and SemFix. We first present the results from our main experiments performed with GenProg. The results from SemFix is presented in Section 4.5.5.

4.5.1 Basic Statistics – Repair Ratio and Regression Ratio

Table 4.2 illustrates the basic statistics for our experiments, including how often repairs are generated (repair ratio) and how often regressions are observed (regression ratio). The "Test-Suites" column in Table 4.2 shows the number of test-suites in each subject, and the "Repairs" column the total number of obtained repairs for each subject. In total, we obtained 3818 repairs out of 14600 trials, resulting in average repair ratio of 26%. The *repair ratio* of each subject is defined as the ratio of the total number of obtained repairs (available in the

[7]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Table 4.3: GenProg experiments: correlations between the regression ratio and various test-suite metrics

Subject	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	-0.92	78%	100%	95%	-0.84	39%	95%	90%	-0.87	2	100	60	-0.93	0.05	0.79	0.67	-0.03	0.42	1.00	0.65
print_tokens2	-1*	7%	8%	7%	0.65	3%	4%	3%	-0.88	2	100	59	-0.80	0.24	0.83	0.79	0.51	0.33	1.00	0.56
tot_info	-0.89	76%	97%	95%	-0.88	67%	92%	89%	-0.84	3	99	56	-0.86	0.51	0.88	0.80	0.91	0.19	1.00	0.49
schedule2	-0.4	98%	99%	99%	0.12*	81%	94%	90%	0.48	16	100	67	0.83	0.67	0.73	0.70	0.41	0.23	0.90	0.38
php	-0.65	0%	89%	22%	-0.75	0%	50%	13%	-0.88	1	100	50	-0.06	0.00	1.00	0.45	-0.7	0.00	1.00	0.3
libtiff	-0.7	9%	31%	20%	-0.73	6%	23%	15%	-0.77	1	71	19	-0.44	0.00	1.00	0.43	-0.47	0.00	1.00	0.82
grep	-0.92	36%	68%	51%	-0.85	22%	53%	36%	-0.62	4	93	18	-0.51	0.01	0.10	0.02	-0.81	0.73	1.00	0.95
findutils	-0.95	2%	33%	22%	-0.95	1%	22%	15%	-0.86	1	56	18	-0.78	0.00	0.54	0.18	-0.82	0.00	1.00	0.65
Average	-0.84	34%	70%	51%	-0.83	23%	56%	43%	-0.66	4	90	43	-0.64	0.13	0.84	0.55	-0.12	0.24	0.99	0.6

The "r" column shows the correlation coefficient between the regression ratio and the corresponding metric in Pearson's r. Negative correlation coefficients (highlighted cells) imply that regressions decrease as the metric increases. The "Average" row shows the average coefficients of each metric across all subjects.

"Repairs" column) over the total number of repair trials (available in the "Test-Suites" column). Note that the total number of repair trials is equivalent to the number of test-suites, because we initiate a separate repair session for each constructed test-suite.

For the obtained repairs, we investigate whether regressions are observed by running each repaired program against its test universe. If a repaired program fails any of the previously passing tests in the test universe, we consider that a regression occurs. The "Regressing" column in Table 4.2 denotes the number of repairs for which regressions are observed. For example, in `tcas`, 348 out of 972 repairs are observed to be regression-causing repairs. The "Regression Ratio" column of Table 4.2 represents the *regression ratio* in each subject, which is defined as the ratio of the number of regression-causing repairs (as shown in the "Regressions" column) over the total number of repairs (as shown in the "Repairs" column). For example, the regression ratio in `tcas` is 348/972, which is approximately 36%. The overall regression ratio ranges from 0% of `print_tokens` to 100% of `schedule`. The "Total" row shows that the average regression ratio of all subjects is 40%.

Ideally, an automated program repair tool should generate more repair (which is indicated by a high repair ratio), and the generated repair should be regression-free as much as possible (which can be indicated by a low regression ratio). However, Table 4.2 shows that the current repair tool has not achieved this ideal scenario. The overall repair ratio is as low as 26%, while the overall regression ratio is as high as 40%. Particularly, in subject `schedule`, the repair ratio is only 4% and all generated repairs cause regressions. Meanwhile, although the repair ratio is relatively high (79%) in `php`, the regression ratio is also quite high (55%).

4.5.2 Correlation Coefficients about Regression Ratio

Our first research question investigates the correlation between test-suite metrics and the regression ratio of repairs:

Research Question 1: Is there a *negative correlation* between the *metrics* of a test-suite and the *regression ratio* of automatically generated repairs? In other words, as test-suite metrics increase, are generated repairs *less likely* to cause regressions?

Table 4.3 shows the correlations between the regression ratio and various metrics of test-suites (i.e., statement coverage, branch coverage, test-suite size, mutation score, and capable-tests ratio). For each metric, the "r" column shows Pearson's product moment correlation coefficients (Pearson's r) [121] rounded to two decimal places. The correlation coefficients of `print_tokens` and `schedule` are not available because in our experiments, repairs for these subjects either always caused regressions (in the case of `schedule`) or always did not cause regressions (in the case of `print_tokens`). All correlation coefficients shown in Table 4.3 are statistically significant at the 0.05 level except those asterisked. In Table 4.3, we also show the minimum, maximum and mean values of each metric (under the "Min", "Max", and "Mean" columns, respectively) of our randomly constructed test-suites.[8] For example, the test-suites of `tcas` has an average of 95% statement coverage ranging between 78% and 100%. Note that we only retrieve these min/max/mean values from the test-suites that have successfully guided repairs, excluding test-suites where no repair is found within the time limit; for these excluded test-suites, the regression ratio of repairs is undefined.

Encouraging Results of Traditional Metrics. In Table 4.3, negative correlation coefficients are highlighted. A negative correlation coefficient of a

[8]The minimum statement/branch coverage of php is 0 because some tests do not execute the marked source files.

metric M implies that as the value of M increases, the regression ratio decreases, which means that the reliability of repairs increases. In general, negative correlations are observed across all traditional test-suite metrics that we investigated (i.e., statement coverage, branch coverage, test-suite size, and mutation score). In particular, statement coverage consistently shows negative correlations across all subjects. Similarly, the other traditional test-suite metrics also show negative correlations in most of the evaluated subjects. Our results suggest that traditional test-suite metrics can also be effectively used to control the regression rate of automatically generated repairs.

As the traditional test-suite metrics (statement coverage, branch coverage, test-suite size, and mutation score) increase, the regression ratio of automatically generated repairs generally decreases, showing the promise of using the traditional test-suite metrics to improve the regression ratio of automatically generated repairs.

Our findings also imply that the efforts to improve test-suites for the purpose of testing—which is already practiced in the industry—can also benefit automated program repair. Indeed, our main findings are consistently observed across real-world large-scale software and controlled small-scale subjects (SIR subjects).

Discouraging Results of Capable-Tests Ratio. Compared to our results on traditional test-suite metrics, the results from capable-tests ratio are discouraging. The expected negative correlations are observed only in large real-world subjects (`php`, `libtiff`, `grep`, and `findutils`). In all the small subjects except for `tcas`, positive correlations are observed. Capable-tests ratio does not seem as useful as the traditional metrics in controlling the quality of generated repairs.

Next, we compare correlation coefficients of different test-suite metrics to investigate our second research question:

Table 4.4: Average rankings of test-suite metrics

Metric	Statement Coverage	Branch Coverage	Test-Suite Size	Mutation Score	Capable-Tests Ratio
Avg. Ranking	1.75	2.5	2.75	4	3.875

Research Question 2: Which test-suite metric is most strongly correlated with the regression ratio of automatically generated repairs?

To investigate this research question, we rank test-suite metrics in each subject in ascending order of the correlation coefficients. The metric whose correlation coefficient is the smallest is ranked first in each subject. For example, in `tcas`, mutation score is ranked first, statement coverage is ranked second, test-suite size is ranked third, etc. Table 4.4 shows the average ranking of each test-suite metric. Statement coverage has the highest average ranking. Indeed, statement coverage is ranked first in 5 subjects (`print_tokens2`, `tot_info`, `schedule2`, `grep`, and `findutils`) out of total 8 subjects, and ranked second (`tcas`) in one subject. Moreover, only statement coverage consistently demonstrates a negative correlation across all subjects.

In our experiments, statement coverage is, on average, more strongly correlated with regression ratio than other test-suite metrics. Our results suggest that to reduce the regression ratio, increasing statement coverage is more promising than improving the other test-suite metrics.

Implication and Limitation of Correlation. It should be noted that the highest correlation of statement coverage does not necessarily imply that a 100% statement coverage-adequate test-suite is most effective in controlling the reliability of repairs. A correlation between A and B only shows how B tends to change as A changes, or vice versa. In fact, as executing a buggy statement may not be sufficient to reveal a bug, more sophisticated test-suite metrics such as branch coverage is more commonly advocated in software testing. Our findings only imply that improving statement coverage seems to be more beneficial than improving other metrics. In other words, a practical implication of our findings is that if the currently available test-suite is neither statement coverage-adequate nor branch coverage-adequate, improving the statement coverage of the test-suite would improve the reliability of repairs more effectively than improving branch coverage.

In Table 4.4, mutation score has the lowest average ranking. Mutation score is ranked even lower than another mutation-based metric, capable-tests ratio, although the average ranking gap between these two metrics is marginal. Mutation score is ranked last in 5 subjects (`schedule2`, `php`, `libtiff`, `grep`, and `findutils`) out of total 8 subjects. A possible reason for the low ranking of mutation score is that the mutants used in mutation testing are sampled *evenly from all possible mutants*, whereas in automated program repair, repair edits are performed *only on suspicious program locations* (the suspicious locations are identified through the fault localization step of automated program repair). When the mutant sampling rate is 100% as in the case of our small subjects, a mutant M can be sampled at a non-suspicious program location L in which a program repair tool does not generate a repair candidate. Even if M is killed, the increase of the mutation score has no direct bearing on improving the reliability of a repair in this case, because no repair candidate is generated at L. In other words, M is not likely to represent unreliable repairs. Meanwhile, when the mutant sampling rate is low as in the case of our large subjects, the chance that a mutant is sampled at suspicious program locations is also low. Another possible reason for the low ranking of mutation score is the discrepancy between mutation operators and repair operators. Given that the mutation scores can change depending on which mutation operators are used [146], using selective mutation operators—instead of all mutation operators—may change the correlation coefficients.

Comparison with Test-Suite Size. In Table 4.4, mutation score and capable-tests ratio show lower average ranking than test-suite size, whereas statement coverage and branch coverage show higher average rankings than test-suite size. This results suggest that increasing statement coverage or branch coverage is likely to reduce regression ratio more effectively than blindly adding arbitrary tests into the test suite. To further investigate whether (a) improving coverage indeed influences the reduction of regression ratio in the statistical sense or (b) regression ratio reduces merely because the test-suite size increases, we perform ANCOVA (analysis of covariance). When performing ANCOVA with statement coverage and test-suite size, the p-value of the statement coverage is less than 0.05 in all subjects except in `php` where the p-value is 0.057. This indicates that the impact of statement coverage on regression ratio is, in general, statistically significant. Meanwhile, the interaction effect between statement coverage and test-suite size is not as significant. The p-value of the interaction effect is statistically insignificant (> 0.05) in `tot_info`, `php`, `libtiff`, `grep`, and `findutils`. The results on branch coverage are similar. The p-value of the branch coverage is less than 0.05 in all subjects except in `print_tokens2` where the p-value is 0.075. The p-value of the interaction effect between branch coverage and test-suite size is also statistically insignificant (> 0.05) in `tot_info`, `php`, `libtiff`, `grep`, and `findutils`.

4.5.3 Correlation Coefficients about Repairability

Next, we investigate a research question regarding repairability:

Research Question 3: Is there a *negative* correlation between the *metrics* of a test-suite and the *repairability* of automated program repair? In other words, would repairability be sacrificed in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

Table 4.5: GenProg experiments: correlations between repairability (repair success rate) and various test-suite metrics

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	0.55	45%	100%	94%	0.88	14%	95%	85%	0.92	2	100	54	0.97	0.00	0.83	0.65	-0.03	0.00	1.00	0.68
print_tokens	0.13	53%	97%	90%	0.24*	39%	93%	82%	0.64	1	100	49	0.78*	0.35	0.85	0.80	0.72	0.37	1.00	0.62
print_tokens2	-0.82	7%	8%	7%	-0.4*	3%	4%	3%	0.87	1	100	52	0.26	0.02	0.84	0.79	0.38	0.02	1.00	0.60
tot_info	0.71	65%	98%	94%	0.87	60%	93%	88%	0.87	1	99	52	0.92	0.42	0.90	0.82	-0.19	0.17	1.00	0.50
schedule	0.33	30%	99%	96%	0.43	14%	95%	86%	0.93	1	100	49	0.71	0.00	0.89	0.81	0.42	0.00	1.00	0.47
schedule2	0.31	66%	99%	99%	0.71	49%	96%	88%	0.99	1	100	54	0.65	0.28	0.77	0.69	0.26	0.20	1.00	0.45
php	0.73	0%	89%	22%	0.76	0%	50%	13%	0*	1	100	50	-0.08	0.00	1.00	0.44	0.81	0.00	1.00	0.28
libtiff	-0.31	6%	31%	23%	-0.19	4%	23%	17%	-0.93	1	77	29	-0.33	0.00	1.00	0.47	0.83	0.00	1.00	0.81
grep	-0.99	36%	73%	60%	-0.99	22%	58%	45%	-0.91	4	100	43	-0.46	0.00	0.77	0.02	-0.69	0.00	1.00	0.58
findutils	-0.94	2%	36%	28%	-0.94	1%	24%	19%	-0.99	1	81	40	-0.66	0.00	0.61	0.20	-0.22	0.00	1.00	0.70
Average	-0.03	31%	73%	61%	0.19	20%	67%	55%	0.27	1	95	47	0.22	0.08	0.84	0.54	0.23	0.08	1.00	0.57

The topmost column shows the five test-suite metrics we investigated from statement coverage to capable-tests ratio. The "r" column of each test-suite metric shows the correlation coefficient between the repairability and the corresponding metric in Pearson's r. All shown coefficients are statistically significant at the 0.05 level except those asterisked. Negative correlation coefficients, shaded in the table, imply that less repairs are obtained as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites.

Table 4.5 shows the correlations between the repairability and various test-suites metrics. Pearson's correlation coefficients are shown in the table with negative coefficients being highlighted.[9] The overall correlation patterns are different between the small SIR subjects and the large real-world subjects. In the small subjects, positive correlations are observed more often than negative correlations across traditional test-suite metrics (statement/branch coverage, test-suite size, and mutation score); test-suite size and mutation score are positively correlated with repairability across all small subjects, and statement/branch coverage is also positively correlated in the majority of the small subjects. This implies that as test-suite metrics increase, it is more likely for a repair to be generated automatically in the small subjects. Meanwhile, the opposite pattern is observed in the large subjects. In the large subjects, negative correlations are observed across the same traditional test-suite metrics as the preceding; mutation score is negatively correlated with repairability across all large subjects, and the remaining traditional test-suite metrics (statement/branch coverage and test-suite size) are also negatively correlated except for in `php`.

Our experimental data indicates that a high quality test-suite helps program repair tool to find a fix in small programs. One possible explanation is that the use of a higher quality test-suite for statistical fault localization leads to more precise fault localization as reported in previous studies [36, 45], and having more precise information about the faulty locations is important in fixing a program. However, it can also be more difficult to satisfy the constraints given in more tests, which makes a negative impact on repairability. Depending on the situations in which repair takes place, test-suite may impact on repairability

[9]Min/Max/Mean values of the table are different from those of Table 4.3, because there we consider only test-suites from which repairs are generated, whereas in Table 4.5, we consider all test-suites.

Table 4.6: Mean and max time of successful repairs with a one-hour timeout

Subject	Repair Time Mean	Repair Time Max
tcas	2.7 m	14.1 m
print_tokens	6.9 m	48.3 m
print_tokens2	0.6 m	29 m
tot_info	1.4 m	4.9 m
schedule	3 m	24 m
schedule2	1 m	13 m

Subject	Repair Time Mean	Repair Time Max
php	7.2 m	56 m
libtiff	14.1 m	57.6 m
grep	24 m	59.6 m
findutils	11.6 m	59.5 m

positively or negatively. We conjecture that our inconclusive result on repairability may be due to interaction effects. For example, repairability may be affected significantly by failing-tests ratio (the proportion of failing tests in a test-suite), and the interaction between failing-tests ratio and test-suite metrics may cause the observed inconclusive result. We leave the investigation of this conjecture as future work.

Our experimental results are inconclusive about the correlation between test-suites and repairability. However, we note that increasing test-suite metric does not always decrease repairability. In some subjects, positive correlations were observed between test-suite metrics and repairability, indicating that as the test-suite metrics increase, repairability tends to increase.

4.5.4 Correlation Coefficients about Repair Time

Our last research question involves repair time:

Research Question 4: Is there a *negative* correlation between the *metrics* of a test-suite and repair time? In other words, would more time be spent in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

Table 4.6 shows the mean and the maximum time taken to generate repairs in each subject. Repair time of small subjects (shown in the left-hand side table) is generally smaller than the repair time of large subjects (shown in the right-hand

Table 4.7: GenProg experiments: correlations between repair time and various test-suite metrics

Subject	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	0.92	78%	100%	95%	0.77	39%	95%	90%	0.99	2	100	60	0.85	0.05	0.79	0.67	-0.88	0.42	1.00	0.65
print_tokens	-0.96	74%	95%	88%	-0.94	62%	92%	79%	-0.98	9	100	50	-0.79	0.76	0.86	0.82	0.78	0.46	0.94	0.62
print_tokens2	-1*	7%	8%	7%	0.04*	3%	4%	3%	-0.57	2	100	59	-0.37	0.24	0.83	0.79	0.75	0.33	1.00	0.56
tot_info	0.68	76%	97%	95%	0.73	67%	92%	89%	0.95	3	99	56	0.9	0.51	0.88	0.80	-0.84	0.19	1.00	0.49
schedule	-0.66*	95%	99%	97%	-0.75	83%	93%	89%	-0.45*	8	100	64	-0.89	0.74	0.86	0.84	-0.48	0.21	1.00	0.40
schedule2	0.84*	98%	99%	99%	0.92	81%	94%	90%	0.93	16	100	67	$0.94^{\dagger}$	0.67	0.73	0.70	0.24	0.23	0.90	0.38
php	-0.36	0%	89%	22%	-0.29	0%	50%	13%	-0.74	1	100	50	0.53	0.00	1.00	0.44	-0.78	0.00	1.00	0.30
libtiff	0.98	9%	31%	20%	0.99	6%	23%	15%	0.87	1	71	19	0.75	0.00	1.00	0.43	0.61	0.00	1.00	0.82
grep	0.85	36%	68%	51%	0.95	22%	53%	36%	0.76	4	93	18	$0.65^{\dagger}$	0.00	0.10	0.01	0.66	0.00	1.00	0.58
findutils	0.86	2%	33%	22%	0.87	1%	22%	15%	0.86	1	56	18	0.6	0.00	0.54	0.18	0.67	0.00	1.00	0.65
Average	0.42	39%	73%	56%	0.36	40%	68%	57%	0.34	4	91	44	0.2	0.29	0.84	0.62	0.07	0.18	0.98	0.54

The topmost column shows the five test-suite metrics we investigated from statement coverage to capable-tests ratio. The "r" column of each test-suite metric shows the correlation coefficient between the repairability and the corresponding metric in Pearson's r. All shown coefficients are statistically significant at the 0.05 level except those asterisked. Negative correlation coefficients (highlighted), imply that less time tends to be taken to obtain a repair, as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites.

side table). Table 4.7 illustrates the correlation between repair time and test-suite metrics. Similar to the case of repairability, no conclusive pattern is observed.

Our experimental results are inconclusive about the correlation between test-suites and repair time. However, we note that increasing test-suite metric does not always increase repair time. In some subjects, negative correlations were observed between test-suite metrics and repair time, indicating that as the test-suite metrics increase, repair time tends to decrease.

4.5.5 Generalizing the Results

To mitigate external threats to our results, we perform the following. First, we replace Pearson's correlation coefficients shown earlier with Kendall's rank correlation coefficients, and check if similar results are observed (Section 4.5.5). Second, we replace GenProg, an automated program repair tool used in our experiments, with another program repair tool, SemFix [117], and check if similar results are observed (Section 4.5.5).

Different Correlation Coefficient: Kendall Rank Correlation Coefficient

To investigate how our results are affected by the use of different kinds of correlation coefficients, Table 4.8 shows the correlation coefficients between regression ratio and test-suites in Kendall's rank correlation coefficients. We use Kendall's τ_b to handle tied ranks [81]. Despite the changes of correlation coefficients, the overall results remains similar. Similar to our previous analysis, we find that:

- Negative correlations are generally observed across all test-suite metrics.
- Statement coverage is, on average, most strongly correlated with regression ratio. The average ranking of test-suite metrics is ordered as follows: statement coverage (2.25) $\leq$ test-suite size (2.25) $\leq$ branch coverage (3) $\leq$ mutation score (3.25) $\leq$ capable-tests ratio (4.25), where the numbers in parentheses show the

average ranking of the corresponding metrics.

- Coverage-based metrics generally show stronger correlation with regression ratio than mutation-based metrics.
- In large real-world subjects, capable-tests ratio is shown to be negatively correlated with regression ratio.

Different Repair Algorithm: SemFix

Apart from our main experiments with GenProg, we also conduct supplementary experiments with another repair tool, SemFix. The objective of this additional SemFix experiments is to investigate if our findings obtained from the GenProg experiments also hold when a different repair algorithm is used. We emphasize that comparing the performance of GenProg and SemFix is not the purpose of this study. In fact, comparing correlation coefficients between the two tools may not meaningful because one tool may show a stronger correlation with the test-suite quality than the other tool, while it may still generate regression-causing repairs more frequently.

Table 4.9 shows the correlations between the regression ratio observed in the SemFix experiments and various test-suite metrics. We used the same test-suites as used in our GenProg experiments. In general, the results are similar to those obtained from the GenProg experiments. As in the GenProg experiment, negative correlations between regression ratio and the traditional test-suite metrics (statement coverage, branch coverage, test-suite size, and mutation score) are observed in the majority of cases. In particular, statement coverage and test-suite size are negatively correlated with regression ratio in all subjects. Mutation score shows negative correlations except in `schedule2`, similar to the GenProg experiment. While branch coverage shows positive correlations in two subjects (`print_tokens2` and `schedule2`), these results are not statistically significant ($p > 0.05$).

Table 4.8: Correlations between the regression ratio and various test-suite metrics (Kendall's τ_b)

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean
tcas	-0.89	78%	100%	95%	-0.64	39%	95%	90%	-0.82	2	100	60	-0.96	0.05	0.79	0.67	-0.12	0.42	1.00	0.65
print_tokens2	-1*	7%	8%	7%	0.98	3%	4%	3%	-0.97	2	100	59	-0.99	0.24	0.83	0.79	0.25	0.33	1.00	0.56
tot_info	-0.86	76%	97%	95%	-0.92	67%	92%	89%	-0.99	3	99	56	-0.82	0.51	0.88	0.80	0.49	0.19	1.00	0.49
schedule2	-0.71	98%	99%	99%	-0.08*	81%	94%	90%	0.51	16	100	67	0.82	0.67	0.73	0.70	0.47	0.23	0.90	0.38
php	-0.44	0%	89%	22%	-0.36	0%	50%	13%	-0.83	1	100	50	0.27	0.00	1.00	0.45	-0.74	0.00	1.00	0.3
libtiff	-0.95	9%	31%	20%	-0.94	6%	23%	15%	-0.99	1	71	19	-0.58	0.00	1.00	0.43	-0.45	0.00	1.00	0.82
grep	-0.98	36%	68%	51%	-0.99	22%	53%	36%	-0.97	4	93	18	-0.93	0.01	0.10	0.02	-0.71	0.73	1.00	0.95
findutils	-0.87	2%	33%	22%	-0.88	1%	22%	15%	-0.89	1	56	18	-0.9	0.00	0.54	0.18	-0.73	0.00	1.00	0.65
Average	-0.83	34%	70%	51%	-0.79	23%	56%	43%	-0.74	4	90	43	-0.66	0.13	0.84	0.55	-0.19	0.24	0.99	0.6

The topmost column shows the five test-suite metrics we investigate. The "τ_b" column of each test-suite metric shows the correlation coefficient between the regression ratio and the corresponding metric in Kendall's τ_b. Negative correlation coefficients, shaded in the table, imply that regressions are less observed as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites. All correlation coefficients shown in the table are statistically significant at the 0.05 level except those asterisked.

Table 4.9: SemFix experiments: correlations between the regression ratio and various test-suite metrics

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	-0.9	78%	100%	95%	-0.62	39%	95%	90%	-0.79	2	100	59	-0.73	0.05	0.79	0.67	-0.01	0.42	1.00	0.646
print_tokens2	-1*	7%	8%	7%	0.46*	3%	4%	3%	-0.69	4	100	60	-0.97	0.49	0.83	0.80	0.66	0.36	1.00	0.554
schedule2	-0.35*	98%	99%	99%	0.64*	82%	93%	90%	-0.3*	16	100	68	0.86	0.67	0.73	0.70	-0.53	0.23	0.69	0.374
libtiff	-0.82	6%	31%	23%	-0.81	4%	23%	17%	-0.87	1	74	27	-0.37	0.00	1.00	0.45	-0.83	0.00	1.00	0.824
Average	-0.77	47%	59%	56%	-0.08	32%	54%	50%	-0.66	5.75	93.5	53	-0.30	0.30	0.84	0.66	-0.18	0.25	0.92	0.6

The topmost column shows the five test-suite metrics we investigate. The "r" column of each test-suite metric shows the correlation coefficient between the regression ratio and the corresponding metric in Pearson's r. Negative correlation coefficients, shaded in the table, imply that regressions are less observed as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites. All correlation coefficients shown in the table are statistically significant at the 0.05 level except those asterisked.

Table 4.10: Average rankings of test-suite metrics (SemFix)

Metric	Statement Coverage	Branch Coverage	Test-Suite Size	Mutation Score	Capable-Tests Ratio
Avg. Ranking	1.5	4	2.75	3.75	3

Similar to the GenProg experiment, we compute the average ranking of each test-suite metric. Table 4.10 shows the average ranking of test-suite metrics for SemFix. Recall that in each subject, the metric whose correlation coefficient is the smallest is ranked first. Statement coverage again is ranked highest as in our GenProg experiment. All other metrics are, on average, ranked lower than test-suite size.

Our experimental results from SemFix generally coincide with our finding from the GenProg experiment, despite the differences in repair algorithms and fault localization techniques. Similar to our experiments with GenProg, the traditional test-suite metrics are, overall, negatively correlated with regression ratio. Specifically, statement coverage remains to be most strongly correlated with regression ratio.

4.6 Threats to Validity

External: Subjects, Test Universes, Mutants, and Repair Tools. Our findings may not generalize to other subjects, although our subjects consist of various software projects of different sizes, extracted from diverse sources (SIR, GenProg, and CoREBench) that contain seeded bugs (SIR), actual bugs (GenProg), and actual regression bugs (CoREBench). Similarly, our test universes may not be representative of the whole test case population, which is theoretically infinite. In general, the larger a test universe is, the more likely regressions are observed when testing a repaired program. To mitigate this threat, we selected subjects that have a large number of test cases.

Similarly, the use of one hour timeout also threatens the external validity of our experimental results. Results may vary if longer timeout is used in the experiments. The external validity of our mutants are also similarly threatened, because in large subjects, we randomly sampled 1–3% of mutants to be able to handle the large size of the mutant population (for SIR subjects whose sizes are smaller, we used the whole mutant population). The relatively weak correlation between mutation score and regression ratio as compared to other test-suite metrics may be due to the differences between mutation testing and automated program repair. Repair candidates generated from an automated program repair tool are not necessarily identical with or similar to mutants generated from a mutation testing tool. Moreover, an automated program repair tool modifies only suspicious program locations, whereas mutation testing does not consider the suspiciousness of program locations when sampling mutants. Our results obtained with randomly sampled mutants is, despite its limitations, still interesting from practical point of view, because mutant sampling is a common approach taken in mutation testing to deal with a large number of mutants practically. One way to mitigate the threats posed by sampled mutants is to change the sampling rate and check if similar results are observed. We leave this investigation as future work.

Lastly, the selection of repair tool may affect the experimental results, and different results may be obtained when a different repair tool is used. To mitigate this threat, we also conducted additional experiments with SemFix, and observed that the overall results are similar to the results from our main experiments. Note that SemFix uses a fundamentally different repair approach from GenProg used for our main experiments.

Internal: Correctness of Tools. Our findings are based on the raw data generated by various tools, i.e, GCOV, GenProg, SemFix, and PROTEUM, where the latter three tools are research prototypes. We also modified PROTEUM because the original PROTEUM cannot handle any of our non-SIR subjects. To mitigate this threat, our modification to PROTEUM is minimally restricted to its parser.

4.7 Conclusion

Many automated program repair tools use a test-suite as the specification of the software under repair. As test cases are merely incomplete specifications, automated program repair tools may end up generating a repair that fails new tests that were not available at the time of repair, causing regressions. Indeed, our experimental results show that regressions often occur in automatically generated repairs. Our study is the largest to date that demonstrates how severe the regression problem of automatically generated repairs is. To address this problem, we investigate the possibility of using test-suite metrics proposed for software testing to control the regression ratio of automatically generated repairs. Our results indicates that traditional test-suite metrics are generally negatively correlated with the regression ratio of repairs, implying that traditional test-suite metrics can also be applied for automated program repair. Among all evaluated test-suite metrics, statement coverage is shown to be most strongly correlated. This implies that to reduce the regression ratio, increasing statement coverage is generally more promising than improving branch coverage or mutation score.

Chapter 5

Anti-patterns for Search-Based Program Repair

Chapter 3 demonstrates that some repair operators in automated program repair tools are ineffective and careful selection of repair operators could help in balancing the trade-off between repair rate and repair precision. However, eliminating a repair operator may reduce the repair rate significantly if it is needed to fix a particular classes of defects. Hence, finding the set of program transformations that should be removed due to their high likelihood of introducing overfitting patches allows more flexibility in balancing the trade-off between repair rate and repair precision. Instead of focusing on the selection of repair operators, this chapter introduces a novel way of solving the ineffectiveness of repair operators in repair tools by proposing a set of rules that restrict the allowed program transformations for each repair operator.

5.1 Introduction

A major challenge in automated program repairs arises from the "incomplete specification" of intended behavior. Indeed, any repair technique tries to patch errors so as to achieve the intended behavior. Yet, in reality, the intended behavior is incompletely specified, often through a set of test cases. Thus, repair methods attempt to patch a given buggy program, so that the patched program passes all tests in a given test-suite T. Unsurprisingly, this may not only lead to incomplete fixes but the patched program may also end up *introducing* new errors, because the patched program may fail tests outside T, which were previously passing [134].

Several recent research articles have pointed out the pitfalls of using test-suites as specification to drive program repair [127, 134]. Furthermore, if the test oracles of the tests in the test-suite are not strong enough, simple program modifications, such as deletion of program functionality, have been shown to be sufficient to repair programs [127]. The situation presents us with an unenviable dilemma — we want to avoid incomplete or incorrect fixes but it is not practical to assume the presence of formal specifications to drive program repair towards correct fixes.

In this chapter, we propose to use *anti-patterns* to help alleviate the problem of incorrect or incomplete fixes resulting from program repair. We present our technique in the context of *search-based program repair* systems. These repair tools seek to repair a buggy program (one failing at least one test in a given test-suite T) by searching among possible fixes by applying fix templates. A proposed fix is "validated" if it passes all the tests in the given test-suite T. One key problem faced in the resulting fixes is that they often boil down to program modifications like deletion of functionality — which, though sufficient to pass tests in given test-suite T, may fail tests outside T and can be unacceptable to developers in general.

Our main idea is simple — for any search-based repair technique which is searching for a plausible repair, we define a set of *anti-patterns* that essentially capture disallowed modifications to the buggy program. In other words, even if such a modification results in the modified program passing all tests in the given test-suite, we do not count them as repairs. Our set of *anti-patterns* is generic and does not vary across application domains.

Conceptually, our idea is different from the strategy of using human patch templates to guide program repair [82]. Generally speaking, the use of human patch templates is geared towards producing patches close to human patches — the underlying assumption being that by going close to human patches, we will avoid incorrect or incomplete fixes. However, this requires providing human patch templates, which is limited by a fixed set of templates, and hence the produced repair may overfit the provided set of templates. Furthermore, there is a strong assumption that by fitting patches to human patch templates, we have a greater chance of the fix being accepted by developers — an assumption that may or may not be true (*e.g.*, [112] argues that "fix acceptability may be an unanswerable question").

Instead of gearing our repair towards human patches by providing human patch templates, we ask ourselves the following research question — *is it possible to drive the repair search towards correct and complete fixes, simply by providing a generic set of* anti-patterns*?* Many of our *anti-patterns* are at the level of the control flow graph — forbidding certain manipulations to the control flow graph. A few of the *anti-patterns* involve assignments affecting branch outcomes and one anti-pattern forbids adding tautologies as branch conditions. Overall, our *anti-patterns* are generic.

Furthermore, and more importantly, we are not proposing a separate repair method based on *anti-patterns*. Our proposed set of *anti-patterns* can be integrated into any existing search-based repair tool, and we can then compare the repair produced after enforcing *anti-patterns* with the original repairs produced by the search-based repair tool. Indeed, we propose a small set of *anti-patterns* and we have integrated them into two existing search based program repair tools: GenProg [141] and SPR [94].

Any automated program repair system is driven by a correctness criterion (to which we repair to), and since formal specifications are usually absent, test-suites are used as correctness criteria. As a fully automated derivation of a formal correctness criterion is often impossible, our *anti-patterns* are not meant to solve the problem of deriving better correctness criteria. Instead, the value of our *anti-patterns* lies in their ability to provide more precise repair hints to developers [78], which is illustrated through our evaluation on patch quality. We evaluated our *anti-patterns* on 86 real bugs obtained from 12 subjects. Results from our experiments indicate that *anti-patterns* could lead search-based program repair tools to producing patches that localizes better by isolating either the correct line or the correct function. Moreover, *anti-patterns* could also reduce the destructive effect of search-based repair tools by producing patches that removes less functionality. Our *anti-patterns* also provide considerable amount of speedup in obtaining the final repair because our *anti-patterns* prune the repair search space. All experimental data are available at the following website: *https://anti-patterns.github.io/search-based-repair/*.

5.2 Prevalence of Anti-patterns

Although various search-based program repair techniques [126, 138, 141] show promising results in generating a large number of patches, prior studies show that most of these patches are often only plausible but incorrect [127]. Specifically, SPR generates 28 out of 40 (i.e, 70%) plausible but incorrect patches, while GenProg generates 50 out of 53 (i.e., 94.33%) plausible patches, for the GenProg benchmarks [87].

To better understand the nature of the plausible patches (see Definition 8 for definition of plausible patches), we performed a manual inspection on all the

Table 5.1: Prevalence of Anti-patterns in Plausible Patches

	Anti-delete CFG exit node				Anti-delete Control Stmt		Anti-delete Single-stmt CFG	Anti-delete Set-Before-If	Anti-delete Loop-Counter Update	Anti-append Early Exit			Anti-append Trivial Conditions	
	Delete exit	Delete return	Delete goto	Delete error	Delete if-stmt	Delete loop	Delete the sole stmt in if	Delete condition	Delete loop counter update	Insert early return	Insert early exit	Insert early goto	Insert Tautology	Insert Contradiction
GenProg	4.00	8.00	2.00	14.00	28.00	6.00	4.00	4.00	2.00	2.00	0	2.00	0	0
SPR	0	7.14	7.14	14.29	10.71	21.43	7.14	7.14	3.57	7.14	3.57	3.57	7.14	39.29
Average	2.00	7.57	4.57	14.14	19.36	13.71	5.57	5.57	2.79	4.57	1.79	2.79	3.57	19.65

All values in this table is calculated in terms of percentage across all automatically generated patches for a given repair tool.
Note that our calculation considers the fact that one plausible patch may exhibit several features.

Table 5.2: Our set of *anti-patterns* with examples that illustrate the usage of each anti-pattern

Anti-patterns	Example
A1: Anti-delete CFG exit node. This pattern disallows removal of return statements, exit calls, functions with the word "error" (i.e., ignoring letter case), and assertions.	**Ex1:** The example below shows a patch generated by GenProg for `libtiff-8f6338a-4c5a9ec`. The patch removes the erroneous exit call. `static void BadPPM(char* file) {` `  fprintf(stderr, "%s: Not a PPM file.\n", file);` `- exit(-2);` `}` Listing 5.1: Example patch for Anti-delete CFG exit node
A2: Anti-delete Control Statement This pattern disallows removal of control statements, e.g., if-statements, switch-statements, and loops.	**Ex2:** The example below shows a patch generated by GenProg for `php-307931-307934`. The patch removes the whole if-then-else statement that checks for the return value of a function call. `call_result = call_user_function_ex(...);` `-if(call_result==SUCCESS && retval!=NULL && ...) {` `-   if(SUCCESS == statbuf_from_array(...))` `-        ret = 0;` `- } else if(call_result == FAILURE) {` `-          php_error_docref(...);   }` Listing 5.2: Example for Anti-delete Control Statement
A3: Anti-delete Single-statement CFG This pattern disallows deletion of the statement within a CFG node that has only one statement.	**Ex3:** The example below presents a candidate patch generated by GenProg for `libtiff-90d136e4-4c66680f`. The patch removes the statement that assigns the return value of 1 which indicates a failure. `fail:{` `- ret = 1;` `}` Listing 5.3: Example for Anti-delete Single-statement CFG
A4: Anti-delete Set-Before-If This pattern disallows deletion of a variable definition if the variable in the definition is used in subsequent if-statement.	**Ex4:** The example shows a GenProg generated candidate patch for `libtiff-d13be72c-ccadf48a`. The patch removes the statement that stores the value of the expression `EstimateStripByteCounts(...)<0`. `- tmp=EstimateStripByteCounts(tif,dir,dircount)<0;` ` if(tmp!=0)` `   goto bad;` Listing 5.4: Example patch for Anti-delete Set-Before-If
A5:Anti-delete Loop-Counter Update Although more sophisticated techniques are needed to ensure termination in patches, we implement an approximation of this pattern by disallowing deletion of an assignment inside a loop if the variables used in the terminating condition intersects with the variables used in the LHS of the assignment.	**Ex5:** The example below shows a patch that deletes the increment statement within a loop. `while( x> 5)` `-   x++;` Listing 5.5: Example: Anti-delete Loop-Counter Update
A6: Anti-append Early Exit This pattern disallows insertion of return statement and goto statement at any location except for after the last statement in a CFG node.	**Ex6:** The example shows a SPR's patch for `php-308262-308315`. The patch adds a conditional return statement before a function call that throws an error. `+ if ((type != 0))` `+    return;` `zend_error((1<<3L),"Uninitialized string ...);` Listing 5.6: Example patch for Anti-append Early Exit
A7: Anti-append Trivial Conditions This expression-level pattern disallows insertion of trivial conditions. A condition is trivial iff (1) it is either true or false constant (e.g., if (0)), (2) it is evaluated to true or false by any assignment of the program variables (e.g., if(x || y || !y)), and (3) it is always evaluated to true or false by any values that program variables can take based on static analysis (e.g., if(x || y != 0) in which y is initialized).	**Ex7:** The example below shows SPR patch for `lighttpd-2661-2662`. The patch in Listing 5.7 appends `!(1)` to the existing condition, which is semantically equivalent to removing the branch. `-if ((fmap[j].key != format->ptr[i + 1]))` `+if ((fmap[j].key != format->ptr[i + 1]) && !(1))` `     continue;` Listing 5.7: Example patch with contradiction

Table 5.3: Problems in search-based program repairs and the corresponding *anti-patterns* aim to solve these problems

Problem	*Anti-patterns*
Weak Oracle. Instead of checking for the actual output of a program, developers may validate the outcome of a failing test by relying on the exit status or assertions of the program. Such statements serve as proxies for verifying the correctness of a program, and thus, they should not be manipulated by machine-generated patches. However, such restrictions are not imposed on automatically generated patches. In fact, patches that simply remove such statements may be more preferable for test-driven program repair techniques as they can be generated faster [127].	***A1: Anti-delete CFG exit node.***
Inadequate Test Coverage. If the program under test has low code coverage, test-driven program repair tools could incorrectly remove a logical block of statements as they are seen as redundant code to the test suite. This may lead to regressions in the patched program [134]	***A2: Anti-delete Control Statement*** ***A3: Anti-delete Single-statement CFG*** ***A4: Anti-delete Set-Before-If***
Mask Existing Vulnerabilities. A patched program may mask previously exposed vulnerability by removing certain branches through implicit data-flow.	***A4: Anti-delete Set-Before-If***
Non-termination. Program repair tools may incorrectly remove a loop update statement, causing infinite loop in the patched program. If no timeout is specified, search-based repair tools may spend the entire repair session to validate the patched program. Worst still, such patches could be mistakenly treated as a repair if the test only checks if an error is thrown within a time limit.	***A5:Anti-delete Loop-Counter Update***
Trivial Patch. An incorrectly patched program may bypass an important functionality or an error check through insertions of premature exit calls. The worst scenario happens when repair tools produce trivial patches that simply insert return-statements based on the value of the expected output of the failing test (e.g., a trivial patch that insert `if(test1) return expected-out;`)	***A6: Anti-append Early Exit***
Functionality Removal Repair tools like SPR may produce patches that are semantically equivalent to functionality removal by inserting tautological condition or contradiction. A tautology will cause the elimination of the check condition while a contradiction will cause the entire branch to be removed.	***A7: Anti-append Trivial Conditions***

machine-generated patches produced by SPR and GenProg (including plausible and correct patches) as well as on the correct developer-provided patches for these bugs. Specifically, we manually analyzed each patch and attempted to answer two questions:

Q1: What makes a given patch plausible? Why is it incorrect (i.e., does not capture the semantics of the developer-provided patch)?

Q2: Do the plausible patches, as a whole, share any common syntactic features that explain their "plausibility" as well as distinguish them from the pool of correct patches (human as well as machine generated)?

The aim was to find a compact set of syntactic features that are independent of the repair templates used by the tool. Table 5.1 shows the results of our manual inspection. Interestingly, our manual inspection identified a set of 14 simple features, shown in the second row of Table 5.1, one or more of which appear in *each* of the plausible patches produced by SPR and GenProg and almost none of the correct patches [1]. They correspond to various modifications to the control flow or the data flow of the program. Each column corresponds to a specific feature and denotes the percentage of plausible patches bearing that feature. For example, *Delete if-statement* (column 6) deletes an if statement and appears in 28% of plausible GenProg patches and 10.71% of plausible SPR patches. Similarly, *Insert Tautology* (column 14) inserts a trivial tautological condition into the program and appears in 7.14% of the plausible SPR patches but none of the plausible GenProg patches. We further generalize and consolidate these 14 features into 7 transformations, shown in the first row of Table 5.1, which we chose to further develop as *anti-patterns* in our approach. Table 5.2 lists the *anti-patterns* that we identified through our manual inspection [50].

[1]Except for one PHP defect and one Python defect.

```
// GenProg AE patch for findutils-84aef0ea-07b941b1
static boolean parse_noop (const struct parser_table* entry, char **argv, int *
    arg_ptr)
{
  (void) entry;
 - return parse_true(get_noop(), argv, arg_ptr);
}
```

Listing 5.8: Example patch generated by GenProg

```
// mGenProg patch for findutils-84aef0ea-07b941b1
static boolean insert_regex (char **argv, int *arg_ptr, const struct parser_table
     *entry, int regex_options)
{...
   if (error_message)
     error (1, 0, "%s", error_message);
   (*arg_ptr)++;
-   our_pred->est_success_rate = estimate_pattern_match_rate(argv[*arg_ptr], 1);
   return true;

// Developer patch for findutils-84aef0ea-07b941b1
insert_regex (char **argv, int *arg_ptr, const struct parser_table *entry, int
    regex_options)
{...
   if (error_message)
     error (1, 0, "%s", error_message);
-  (*arg_ptr)++;
   our_pred->est_success_rate = estimate_pattern_match_rate(argv[*arg_ptr], 1);
+  (*arg_ptr)++;
   return true;
```

Listing 5.9: Example patches generated by mGenProg and Findutils developer

5.3 How Anti-patterns may Help

We illustrate how *anti-patterns* can improve program repair by showing two examples of fixes in two projects. The first example shows the effect of applying *anti-patterns* for improvement in *fix localization* (the program location at which the human fix is applied), while the second example demonstrates the benefit of *anti-patterns* in reducing patch generation time.

Example 1: Improving fix localization. Listing 5.8 shows the GenProg patch for `findutils-84aef0ea-07b941b1` generated by GenProg. The GenProg patch deletes the only return statement in `parse_noop(...)`. This patch violates our *Anti-delete CFG exit node* pattern. In contrast, our version of GenProg modified with *anti-patterns*, called mGenProg, removes the statement that assigns the return value of `estimate_pattern_match_rate` to the field `our_pred->est_success_rate`. Meanwhile, Findutils developer moved the

```
// GenProg, mGenProg & Developer patch for php-309111-309159
if ((p = memchr(s, '?', (ue - s)))) {
    pp = strchr(s, '#');
    if (pp && pp < p) {
+   if (pp - s) {
+      ret->path = estrndup(s, (pp-s));
+      php_replace_controlchars_ex(ret->path, (pp - s));
+   }
            p = pp;
            goto label_parse;
    }
}
```

Listing 5.10: Example patch generated by GenProg, mGenProg and PHP developers

statement `(*arg_ptr)++` to the location after the problematic statement `our_pred->est_success_rate=estimate_pattern_match_rate(...);` that throws "Segmentation fault" error due to out-of-bound access of the `argv` array (Listing 5.9). We argue that in this example, our mGenProg patch is preferable to GenProg's because (1) mGenProg localizes the correct function compared to the GenProg patch, which is applied inside a completely different function `parse_noop()`; (2) mGenProg correctly pinpoints the function call that causes the error, while the GenProg patch completely removes the functionality encapsulated by the `parse_noop()` function. This example shows that *anti-patterns* can improve fix localization and eliminate nonsensical patches that remove functionality.

Example 2: Accelerating program repair. Listing 5.10 shows a patch that inserts a conditional statement that can be copied from other places within the same file. While both GenProg and mGenProg generate the patch in listing 5.10 that is in fact equivalent to the correct patch, mGenProg takes only 13.7 hours compared to 20.6 hours taken by GenProg (i.e., mGenProg achieves a 20.6/13.7 = 1.5x speedup). Thus, if the correct repair can be found within the repair space, our *anti-patterns* can serve as a search-space pruning mechanism that reduces the time taken to find the correct repair through filtering of invalid patches.

Table 5.3 shows the common problems in the patches generated by search-based program repair tools together with the *anti-patterns* that solve these problems.

5.4 Integrating Anti-patterns

We integrate our *anti-patterns* directly into two search-based repair tools (i.e., GenProg AE [141] and SPR [94]).

```
Input: P': Program
Input: M: Transformations functions
Output: isAnti: indicates if M violates any anti-patterns
isAnti ← $false$;
if $M.type == delete$ then
    if $isSingleCFGStmt(M.stmtk)$ then
        isAnti ← $true$;
    else if $isExitNode(M.stk) \,||\, isCondition(M.stk)$ then
        isAnti ← $true$;
    else if $isAssignment(M.stmtk)$ then
        isAnti ← $isSetBfIf(M.stk) \,||\, isSetInLoop(M.stk)$;
    end
else if $M.type == append$ then
    isAnti ← $isExitNode(M.stk) \wedge \neg isLastStmt(M)$;
end
return isAnti
```

Procedure `isAntipattern`

M.stk: the AST node type of M.
M.type: the edit type of M

Procedure 2 shows our *anti-patterns* filtering algorithm. The function *isSingleCFGStmt*(*E.stmtk*) corresponds to the *Anti-delete Single-Statement CFG* pattern. Similarly, *isSetBefIf*(*E.stk*) checks for the *Anti-delete Set-Before-If* pattern, while *isSetInLoop*(*E.stk*) corresponds to the *Anti-delete Loop-Counter Update* pattern. The function *isCondition*(*E.stk*) indicates whether an edit E involves a conditional statement, which is used in the *Anti-delete Control-Statement.* The function *isExitNode*(*E.stk*) checks if the statement in Edit E is a CFG exit node. Both *Anti-Delete CFG exit node* and *Anti-append Early Exit* use this function. The function *isLastStmt*(E) checks if a statement will be inserted as the last statement in a CFG block to fulfill the requirement for the *Anti-append Early Exit* pattern. As many search-based approaches [82, 94, 138, 141] are based on evolutionary algorithm [85] in which a population is reproduced, evaluated, and selected, we recommend integrating our *anti-patterns* filtering algorithm before the initial population is generated to reduce the time spent in evaluating each individual in a population.

Modification of SPR and GenProg. Algorithm 3 shows the pseudo-code of the mSPR repair generation algorithm. We implement our *anti-patterns* on two parts of mSPR: (1 – first box). For candidate repairs that do not require condition synthesis, we apply similar modifications to mSPR and mGenProg (refer to Procedure 2). (2 – second box). For candidate repairs that require condition synthesis in mSPR, we apply the *Anti-append Trivial Conditions* pattern to each synthesized condition. The function $isTrivialCondition(c)$ checks if the given condition c is a trivial condition (refer to Table 5.2 for definition of trivial conditions). As our modifications on GenProg is similar to the changes on mSPR for repairs that do not require condition synthesis, we leave out the details for mGenProg.

5.5 Experiments

We compare the effectiveness of *anti-patterns* on GenProg and SPR using two sets of benchmarks: (1) the CoREBench benchmarks [48] and (2) the GenProg benchmarks [87]. We use the CoREBench benchmarks for the evaluation set because it contains real errors in widely used C programs. Although our manual inspection for deriving anti-patterns in Section 5.2 was based on plausible patches from the GenProg benchmarks, this study used just *one* generated patch per buggy program. A recent study shows that the typical repair search space for these bugs contains up to thousands of plausible patches [95]. Thus, we feel it is still meaningful to study the impact of *anti-patterns* on the complete repair space of these bugs. Our evaluation studies the following research questions:

RQ1 How do *anti-patterns* affect the quality of patches generated by search-based

```
Input: P: Program
Input: positive and negative test cases NegT and PosT
Input: $M$: Transformation functions.
Output: the repaired program P' or $\emptyset$ if failed
1   for P' in M(P) do
        if $\neg isAntipattern(P', M)$ then
            $M' \leftarrow M \cup P'$;
        end
    end
2   for P' in M'(P) do
3       if P' contains abstc then
4           $C \leftarrow CondSynthesis(P', NegT, PosT)$;
5           for c in C do
                if $\neg isTrivialCondition(c)$ then
                    $C \leftarrow C/c$;
                    if Test(P'[c/abstc],NegT, PosT) then
                        return P'[c/abstc]
                    end
                end
            end
6       else if Test($P'$, $NegT$, $PosT$) then
7           return $P'$
8       end
9   end
```

Algorithm 3: mSPR Repair generation algorithm

$CondSynthesis(P, NegT, PosT)$: searches for a sequence of values in P that pass all tests in NegT and PosT. The output of this function is C — the set of all synthesized conditions in the repair space.
$P[c/abstc]$: the result of replacing every occurrence of *abstc* in P with the condition c.
$Test(P, NegT, PosT)$: check if the program P passes all tests.

Table 5.4: Subject Programs and Their Basics Statistics

Subjects	Description	kLoC	Tests
coreutils	File, Shell and Text manipulation Utility	83.1	4772
findutils	Directory Searching Utility	18.0	1054
grep	Pattern Matching Utility	9.4	1582
make	Program executable generation utilities	35.3	528
php	Programming Language	1046	8471
libtiff	Image Processing Library	77	78
python	Programming Language	407	35
gmp	Math Library	145	146
gzip	Data Compression Utility	491	12
wireshark	Network Packet Analyzer	2814	63
fbc	Compiler	97	773
lighthttpd	Web Server	62	295

program repair tools?

RQ2 How many nonsensical patches can our *anti-patterns* eliminate to reduce manual inspection costs?

RQ3 When our modified tools produce the same patch, what is the speedup that we achieve?

RQ4 How does the use of *anti-patterns* compare to an approach that simply prohibits deletion?

5.5.1 Experimental Setup

We evaluate the effects of *anti-patterns* on 49 defects from the CoREBench benchmarks and at least 37 defects from the GenProg benchmarks. We exclude some versions in our evaluation due to specific technical difficulties, such as benchmarks that require specific system configurations to be built. Specifically, we exclude 21 defects from the CoREBench benchmarks. For the GenProg benchmarks, we manage to reproduce the bugs for 42 defects in the original GenProg experiment and 37 defects in the original SPR experiments.

Table 5.4 lists information about these subjects. The first four rows of the table list the details for the four CoREBench subjects while the remaining rows show relevant statistics about the GenProg subjects. For each bug, we run GenProg, mGenProg, SPR and mSPR to produce repairs.

Many of our *anti-patterns* block functionality deletion, so it is natural to ask if the same effect could be achieved by simply disallowing deletion in repair. To answer RQ4, we implement a customized version of GenProg, called dGenProg, where we disallow the usage of the deletion mutation operator. We reuse the same parameters listed in previous work [87] for running GenProg. One significant difference is that we switch to the deterministic adaptive search algorithm (AE) [141] to control potential randomness. Each run of GenProg, mGenProg, dGenProg, SPR, and mSPR terminates either after all candidate repairs have been evaluated or when a patch is found (i.e., each tool runs to completion without timeout). All experiments for GenProg, mGenProg, and dGenProg were performed by distributing the load on 20 virtual machines with single-core Intel Xeon 2.40GHz processor and 19GB of memory. All experiments for SPR and mSPR were performed on a 12-core Intel Xeon E5-2695 2.40Ghz processor and 408GB of memory.

After collecting all the repairs, we manually inspect each of these patches and compare the quality of patches generated by GenProg versus mGenProg, mGenProg versus dGenProg, and SPR versus mSPR.

Definition 7 *We measure the quality of patches generated by search-based repair tools using the criteria defined below:*

(Q1) Same Patch. *A generated repair is considered "Same Patch" if both the original tool and the modified tool generate exactly the same repair.*

(Q2) Localizes Correct Line. *A generated repair is considered "Localizes Correct Line" if both the generated patch and the human patch generate repairs that modify the same line. For example, we categorize the mGenProg patch in Listing 5.9 as "Localizes Correct Line".*

(Q3) Localizes Correct Function but Incorrect Line. *A generated repair is considered a patch that localizes the correct function if both the generated patch* *and the human patch modify statements within the same function.*

(Q4) Removes Less Functionality. *A generated repair is considered a repair that removes less functionality if the repair removes or skips over (e.g., by inserting return) fewer lines of source code than the repair generated by the unmodified repair tool.*

(Q5) No Repair. *We label a benchmark as "No Repair" when the original tool generates a repair but the modified tool has iterated through the entire repair space and produce no final patch.*

We categorize the patch quality of each repair according to the order listed above (i.e, we first check if a patch is "Same Patch" and only categorize a patch as "Removes Less Functionality" if it does not satisfy other more preferable criteria (e.g, "Localizes Correct Function but Incorrect Line"). We eliminate the potential discrepancies on categorization by ensuring that each defined criteria can be measured through comparisons of the syntactic differences between two patches.

Each column in Tables 5.5, 5.6 and 5.9 corresponds to the criteria defined above. The "Others" column denotes the cases where the patch does not fulfill any of the defined criteria. Numbers in the last row in Tables 5.5, 5.6, and 5.9 are of the form $x+y = z$, where x represents the number of patches in the CoREBench benchmarks, y denotes the number of patches in the GenProg benchmarks, and z is the total number of patches in both benchmarks.

We also manually classify and compute the number of correct repairs and the number of plausible repairs.

Definition 8 *We use the definition below for our patch correctness analysis:*

Correct Repair. *A repair r is a* correct *repair if (1) r passes all test cases in the test suite and (2) r is semantically equivalent to the repair issued by developer.*

Plausible Repair. *A repair r is a* plausible *repair if (1) r passes all test cases in the test suite but (2) r is **not** semantically equivalent to the repair issued by* *developer.*

5.5.2 Evaluation on CoREBench benchmarks

The first four rows of Table 5.5, 5.6, 5.8, and 5.9 show the evaluation results for the CoREBench benchmarks.

Patch Quality (RQ1)

Table 5.5 shows that both GenProg and mGenProg produce the same patch for 10 defects in the CoREBench benchmarks. mGenProg could localize the correct line in 7 more defects than GenProg. mGenProg also generates patches that remove less functionality in 7 defects.

Table 5.6 shows that both SPR and mSPR produce the same patch in 17 defects. mSPR localizes the correct line in 2 more defects than SPR. For 7 defects, mSPR generates patches that removes less functionality.

Table 5.7 shows the overall patch correctness analysis results for GenProg, mGenProg, SPR, and mSPR for each subject. GenProg generates 34 plausible patches while mGenProg produces 33 plausible patches for the CoREBench benchmarks. Specifically, mGenProg does not produce any repair for `findutils-e8bd5a2c-66c536b` because the patch violates our *anti-patterns*. Both SPR and mSPR generate 34 plausible patches for the CoREBench benchmarks.

Improvement on fix localization. Our results show that *anti-patterns* could lead both mGenProg and mSPR to producing patches that localize either the correct line or the correct function. *Anti-pattern*-enhanced techniques may achieve this improvement because *anti-patterns* may filter all invalid repairs on a given location, forcing fixes to be generated at other locations. We claim that the ability to localize more precisely is important because when the repair tools fail to generate the correct repair, the next best thing is to check whether they can still generate hints that may lead developers to the repair faster.

Less functionality removal. Under the presence of weak oracles [127], search-based repair tools may generate patches that pass the test suite by removing untested functionality. Our results shows that *anti-patterns* help in producing patches that remove less functionality and thus reduce the potential destructive effects of generated patches.

Comparison between mGenProg and mSPR. Our *anti-patterns* integration achieves greater improvement of patch quality on GenProg compared to SPR. We think that this difference may be due to SPR being innately restricted by its set of transformation schemas, which contain transformations that are often used in human patches.

Predominance of Plausible Patches. Both GenProg and SPR do not generate any correct patch for the CoREBench benchmarks. One possible explanation is that the defects in the CoREBench have higher error complexity than other benchmarks. Thus, more substantial patches are required to fix the errors in these benchmarks [48]. These results also agree with our earlier observation (in Section 5.2) that there is a clear predominance of plausible but incorrect patches among all automatically generated patches.

Reducing Manual Inspection Cost (RQ2)

SPR may produce multiple patches in one repair session due to the use of batch compilation and its staged repair algorithm. Given several candidate repairs, developers need to manually inspect and verify each individual patch.

Figure 5-1 shows the total number of patches generated by SPR versus mSPR for the CoREBench subjects. As SPR and mSPR can produce multiple patches for a given bug, Figure 5-1 reports the total number of patches, while the data in Tables 5.5, 5.6 and 5.7 uses a single, best patch (according to the order in Definition 7) among all generated patches for a particular bug. Overall, SPR generates 87 patches while mSPR only generates 54 patches. Our patch analysis reveals that all 33 additional patches generated by SPR (not generated by mSPR) are indeed plausible but incorrect patches.

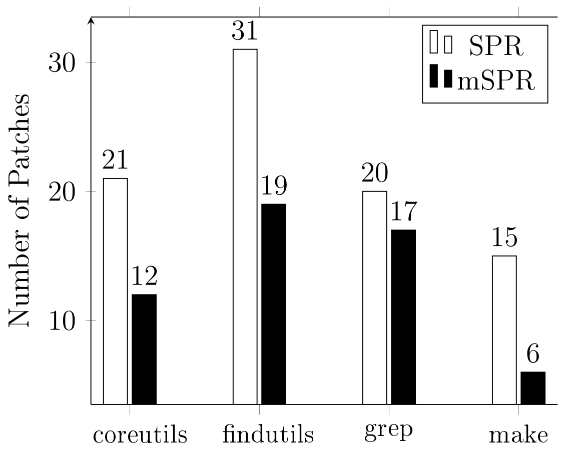


Figure 5-1: Number of Patches Found by SPR vs. mSPR

Discussion on Number of Plausible Patches. Though prior evaluation of search-based repair [141] focuses on measuring the number of successful repairs, our results show that mSPR actually produces less number of candidate repairs than SPR because some of the plausible patches produced by SPR are actually nonsensical patches that are eliminated by our *anti-patterns.* Producing less plausible patches could save the time spent on manual filtering of invalid patches, which would eventually be rejected by developers.

Speedup (RQ3)

The "Average Speedup (Same Patch)" column in Tables 5.5, 5.6, and 5.9 denotes the average speedup obtained when we only considered the subjects where both the original tool and the modified tool produce the same patch. We use the formula below for our speedup calculation (*Repair Time* is defined as the time taken for a repair to be generated):

$$Repair\ Time\ Speedup = \frac{Original\ Repair\ Time}{Modified\ Repair\ Time} \tag{5.1}$$

When GenProg and mGenProg produce the same patch, mGenProg obtain an average repair time speedup of 1.39x while mSPR obtain an average repair time speedup of 1.78x, for the CoREBench benchmarks.

Table 5.8 shows the overall reduction in the total number of repair candidates generated for mGenProg and mSPR. The last row is of the form x, y, z where x denotes average for CoREBench subjects, y denotes average for GenProg subjects and z denotes the average for all subjects. We calculate the "Repair Space Reduction" according to the formula below (where $TotC$ refers to the total number of repair candidates within the entire repair search space):

$$Repair\ Space\ Reduction = (1 - \frac{Modified\ TotC}{Orig\ TotC}) * 100 \tag{5.2}$$

On average, mGenProg achieves 41% repair space reduction compared to GenProg while mSPR obtains 27% repair space reduction compared to SPR for CoREBench subjects.

Discussion on Speedup. Tables 5.5 and 5.6 show that by enforcing *anti-patterns*, we produce patches faster due to repair space reduction shown in Table 5.8. One conceptual argument against the idea of *anti-patterns* may be that it might make the repair search unduly inefficient. These results show that it is not so. In fact, the anti-patterns skip "irrelevant" parts of the repair space (i.e., repairs that causes undesirable behavior, such as the deletion of the symptoms of a bug).

Comparison with dGenProg (RQ4)

Table 5.9 shows the results for dGenProg versus mGenProg for the CoREBench benchmarks. While both mGenProg and dGenProg produce 12 same patches, mGenProg localizes better compared to dGenProg in seven more subjects than dGenProg. *Although dGenProg explicitly prohibits deletions, our results show* *that mGenProg actually removes less functionality in five subjects compared to dGenProg.* Our analysis reveals that dGenProg may produce patches that skip over many source lines of code by introducing early return. For `make-73e7767f-d584d0c1`, mGenProg localizes the correct line while dGenProg do not produce any repair.

When mGenProg and dGenProg produce the same patch, mGenProg achieves an overall speedup of 1.20x over dGenProg in the CoREBench benchmarks.

Improvement over dGenProg. Our results on the CoREBench benchmarks show that GenProg with *anti-patterns* produce patches of better quality and faster than GenProg that simply prohibits deletions.

5.5.3 Evaluation on GenProg benchmarks

The 5-12th rows of Tables 5.5, 5.6, 5.8, and 5.9 show the experimental results for the GenProg benchmarks. Tables 5.5 and 5.6 illustrate that our *anti-patterns* achieve similar improvement on patch quality on the GenProg benchmarks. In particular, mGenProg localizes better than GenProg in seven more defects. mGenProg also removes less functionality in 12 defects. In contrast, mSPR removes less functionality than SPR in three defects on GenProg benchmarks.

Table 5.7 shows that GenProg produces 3 correct repairs and 39 plausible repairs while mGenProg produces two correct repairs and 37 plausible repairs for the GenProg benchmarks. mGenProg does not generate any repair for three subjects due to their violations of *anti-patterns*. Instead of producing correct repair as in GenProg, mGenProg only generates plausible repairs for `php-309892-309910` because the correct repair actually involves deletion of a if-statement, which violates our *Anti-delete Control Statement* pattern. In contrast, mSPR produces one more correct patch than SPR and 23 plausible repairs. Specifically, mSPR produces correct patch for `php-308262-308315` while SPR only generates plausible patch for this version. For `libtiff-086036`

`-1ba752`, mSPR does not generate any repair while SPR generates patch with trivial condition that disables a branch. As the correct repair for this libtiff defect requires modifications of multiple statements, our analysis reveals that the correct repair is indeed outside of SPR's repair space.

We also achieve similar reduction on repair time on the GenProg benchmarks, as in the CoREBench benchmarks.

***Restrictiveness of* anti-patterns.** Another conceptual argument against the idea of *anti-patterns* may be that *anti-patterns* will be overly restrictive and will rule out any repair in many cases whereas, if an existing search-based tool produces some repair, it still helps the developers to some extent. Our results on the GenProg benchmarks show that *anti-patterns* are not overly restrictive and, in the few cases where it ruled out any repair, indeed no valid repair existed.

Weak Proxies. Our experiments for SPR and mSPR use the updated proxies in previous work [127], which modifies the test harness and the developer test script for `php` and `libtiff`. In contrast, we reuse the weak proxies for our experiments on GenProg and mGenProg. We used the weak oracles for GenProg and the strong oracles for SPR because they are provided together with the original tool distribution. If we compare the row 5-6 of Tables 5.5 and 5.6 in which different set of proxies are used, we observe that having a stronger proxy does not help SPR substantially in terms of fix localization. Indeed, the improvement of mSPR over SPR in terms of localizing the correct line, is similar to the improvement of mGenProg over GenProg.

Discussion on Patch Correctness. Our results show that enforcing anti-patterns does not necessarily lead to patches that are exactly equivalent to the human patches. This is not entirely unexpected, because *we only mark a generated patch as correct, if it is near identical to the developer provided patch.* Our repair method is driven by a suite of test cases and aims to pass the test-suite while not inserting any of the anti-patterns. It frees the developers from providing different human patch patterns for different defect classes, exception types, vulnerabilities, etc. Nevertheless, mSPR still generates one more correct repair than SPR while mGenProg generates one plausible repair that removes a branch from the original program.

5.6 Threats To Validity

We identify the threats to validity of our experiments.

Set of *anti-patterns*. Our *anti-patterns* merely represent bug, tool, and language agnostic patterns that we found frequently occurred in bad patches and seldom in correct ones. Though our experimental results show that our proposed *anti-patterns* are effective in eliminating invalid patches, we do not claim that our proposed set is a "complete" set.

Search. We terminate the search for repairs in both GenProg and SPR after a repair has been found, due to limited resources. While both tools support full exploration that may generate similar patches as in our modified versions, such exploration may also lead to increase in the number of invalid patches and longer manual inspection time. As "we use the deterministic adaptive search algorithm (AE) to control potential randomness" (Section 5.5.1), we will re-evaluate the savings for the stochastic algorithm in future work.

Patch Correctness Analysis. While we tried to assess repair quality across multiple dimensions, our check for semantic equivalence is inherently incomplete and many fixes exist for a particular fault. Our conservative patch analysis classifies a patch as "correct" only when near identical to the human patch. Hence, the number of repairs reported as "correct" may be an underestimate because a plausible patch marked as not correct could very well be semantically equivalent to the developers' provided patch.

Generality of *anti-patterns*. As we only evaluate the effect of *anti-patterns* on CoREBench benchmarks [48] and the GenProg benchmarks [87], our *anti-patterns* may have different effects on other benchmarks. Nevertheless, our experimental results show that *anti-patterns* provide similar benefits at least in both these benchmarks.

5.7 Chapter Summary

In this chapter, we proposed integrating *anti-patterns* to search-based program repair. Our experimental results show that by enforcing *anti-patterns*, we produce patches with more pleasant properties, such as patches that delete less functionality, and localize better. Tools integrated with *anti-patterns* also could generate patches faster due to repair space reduction. A recent study [95] shows the abundance of plausible patches and sparsity of correct patches — thereby arguing for rich specifications (beyond test-suites) to guide the repair process. Our results indicate that our *anti-patterns*, while they are not correctness specifications, form one such set of specifications whose enforcement can improve patch quality.

While in this work we explicitly specified a set of anti-patterns as prohibited code transformations, in future, it is feasible to implicitly specify anti-patterns as selected “code smells”. Thus, during the repair search, any program modification that produces a program with a bad code smell could be effectively prohibited.

Our work opens the possibility of adapting the idea of *anti-patterns* to other search-based software engineering activities beyond program repair. For example, specific code *anti-patterns* identifying energy hot-spots may be employed for energy reduction.

In future, we are unlikely to have programming environments that automatically patch all errors without sufficient intervention or domain knowledge. Meanwhile, it might be possible to have programming environments, which attempt to patch programs so as to pass a given test-suite and point the developers to likely error locations and likely fixes. Our proposal of *anti-patterns* is a step in this direction.

Table 5.5: Overall Results on GenProg (AE) versus mGenProg (mAE)

		Different Patch										
		Localizes Better										
Subjects	Same Patch	Localizes Correct Line		Localizes Correct Function but Incorrect Line		Less Functionality Removal	No Repair		Others		Average Speedup (Same Patch)
		AE	mAE	AE	mAE		AE	mAE	AE	mAE	
coreutils	0	0	0	4	4	5	0	0	5	0	-
findutils	4	0	4	2	1	1	0	1	5	0	1.11
grep	4	0	2	3	2	1	0	0	2	0	1.30
make	2	0	1	3	2	0	0	0	0	0	1.77
php	10	1	1	0	2	6	0	0	8	0	2.08
libtiff	3	0	4	3	1	5	0	3	10	0	1.13
python	1	0	0	0	0	0	0	0	0	0	0.98
gmp	-	-	-	-	-	-	-	-	-	-	-
gzip	1	0	0	0	0	0	0	0	0	0	1.12
wireshark	0	0	3	0	0	0	0	0	3	0	-
fbc	-	-	-	-	-	-	-	-	-	-	-
lighthttpd	1	0	0	0	0	1	0	0	1	0	1.85
Total	10+16=26	0+1=1	7+8=15	12+3=15	9+3=12	7+12=19	0+0=0	1+3=4	12+22=34	0+0=0	1.39+1.43=1.42

Table 5.6: Overall Results on SPR versus mSPR

Subjects	Same Patch	Different Patch									Average Speedup (Same Patch)
		Localizes Better				Less Functionality Removal	No Repair		Others		
		Localizes Correct Line		Localizes Correct Function but Incorrect Line							
		SPR	mSPR	SPR	mSPR		SPR	mSPR	SPR	mSPR	
coreutils	6	0	0	2	2	3	0	0	3	0	1.56
findutils	6	1	2	1	0	1	0	0	1	0	1.62
grep	5	0	1	3	3	2	0	0	3	0	2.15
make	0	0	0	2	2	1	0	0	1	0	-
php	15	0	2	2	0	0	0	0	0	0	1.96
libtiff	2	1	1	1	0	1	0	1	1	0	2.10
python	2	0	0	1	1	0	0	0	0	0	1.50
gmp	2	0	0	0	0	0	0	0	0	0	1.42
gzip	1	0	1	0	0	0	0	0	1	0	1.08
wireshark	3	0	1	1	0	0	0	0	0	0	1.85
fbc	-	-	-	-	-	-	-	-	-	-	-
lighthttpd	0	0	2	1	0	2	0	0	3	0	-
Total	17+25=42	1+1=2	3+7=10	8+6=14	7+1=8	7+3=10	0+0=0	0+1=1	8+5=13	0+0=0	1.78+1.65=1.69

Table 5.7: Patch Correctness Analysis Result on mGenProg and mSPR

Subjects	GenProg		mGenProg		SPR		mSPR	
	Correct	Plausible	Correct	Plausible	Correct	Plausible	Correct	Plausible
coreutils	0	9	0	9	0	11	0	11
findutils	0	11	0	10	0	9	0	9
grep	0	9	0	9	0	11	0	11
make	0	5	0	5	0	3	0	3
php	2	17	1	18	8	9	9	8
libtiff	0	16	0	13	1	4	1	3
python	1	0	1	0	1	2	1	2
gmp	0	0	0	0	1	1	1	1
gzip	0	1	0	1	1	1	1	1
wireshark	0	3	0	3	0	4	0	4
fbc	0	0	0	0	0	0	0	0
lighthttpd	0	2	0	2	0	4	0	4
Total	0+3=3	34+39=73	0+2=2	33+37=70	0+12=12	34+25=59	0+13=13	34+23=57

Table 5.8: Subject Programs and Repair Space Reduction Results for mGenProg and mSPR

Subject	Repair Space Reduction(%)	
	mGenProg	mSPR
coreutils	43	22
findutils	47	19
grep	38	32
make	37	36
php	37	20
libtiff	43	61
python	31	26
gmp	-	9
gzip	43	31
wireshark	42	35
fbc	-	-
lighthttpd	41	15
Average	41, 40, 40	27, 28, 28

Table 5.9: Overall Results on mGenProg (mAE) versus dGenProg(dAE)

Subjects	Same Patch	Different Patch									Average Speedup (Same Patch)
		Localizes Better				Less Functionality Removal	No Repair		Others		
		Localizes Correct Line		Localizes Correct Function but Incorrect Line							
		dAE	mAE	dAE	mAE		dAE	mAE	dAE	mAE	
coreutils	4	0	0	1	1	3	0	0	3	0	1.80
findutils	3	0	5	1	1	0	0	0	5	0	1.03
grep	2	0	1	3	3	2	0	0	4	1	0.79
make	3	0	1	1	1	0	1	0	0	0	1.18
Total	12	0	7	6	6	5	1	0	12	1	1.20

Chapter 6

Relifix: Automated Repair of Software Regressions

As our study of repair operators in Chapter 3 illustrates the inefficiency of the current set of operators, we introduce a new program repair approach that employs a careful selection of repair operators in this chapter. Compared to anti-patterns (Chapter 5) that are designed to solve the problems found in automatically generated patches, the set of repair operators proposed in this chapter aims to solve the problem in human generated patches — the "software regression" problem.

6.1 Introduction

Software regression captures the scenario where failures occur in previously passing tests. As software evolves due to changes in software requirement and bug fixes, regression bugs may be introduced. Even worse still, fixing a regression bug is likely to introduce another regression bug due to low-quality patches and inadequate testing.

Prior studies on regression errors primarily focus on techniques for localizing and understanding of regressions. The delta debugging approach searches for failure-inducing circumstances contributing to test failures (i.e., the set of code changes and the state differences between passing and failing tests) using a divide-and-conquer algorithm [148]. Given a reference program, a buggy program, and an input that fails on the buggy program, the Darwin approach generates alternative input that fails on the buggy program, then compare the executions of the two inputs to pinpoint the root cause of the error [125]. Previous studies show promising results in locating the cause of regression errors. However, after locating the cause of regression errors, how do we utilize the availability of a previous working version to automatically repair such errors? This is addressed in the current chapter.

Fixing regression errors manually is time-consuming and error-prone. Recent study stated that some regression errors could take up to 8.5 years before they are detected and fixed by developers [48]. Recently, several automated program repair techniques have been introduced. Arcuri and Yao suggested adapting evolutionary algorithms for automatic program generation [34]. Weimer et al. utilized genetic programming for automated program repair [67, 90]. Wei et al. leverages software contract to automatically fix faulty Eiffel classes [140]. Nguyen et al. employed symbolic execution and component-based program synthesis for discovering the code required for fixing the buggy program [117]. Kim et al. proposed an automated patch generation approach (i.e., PAR) that utilizes common fix patterns learned from manual inspection of human patches [82]. Recent study shows that statements or expressions required for fixing exist in previous commits of the programs [43, 106]. However, existing automated program repair techniques have not fully exploited information from the software change history for automated repair of regressions. In this chapter, we verify the possibility of using syntactical information between program versions and test execution history to repair problematic changes that causes regressions.

The key challenge in repairing regression errors is to retain as much of the new functionalities introduced along with the new version as possible while reproducing the regression tests' behavior in the previous version.

Criteria 1 *We want our automated repair of regression error to follow the following criteria:*

C1: Introduces small changes *Retains as much of the code of the new version as possible as more changes may lead to more regression errors.*

C2: Produces readable code *Generates source code that developers can understand and verify easily.*

C3: Passes progression tests *Progression tests that pass in the new version and fail in the previous version must remain passing after the repair.*

C4: Passes previously failed regression tests *Regression tests that fail in the previous version and pass in the current version must be made passing in the new version.*

C5: Only change if no regression will be introduced *If changes caused other tests in the test suite to fail, then leave the source code unchanged. The repaired version should not introduce further regression error.*

We present a novel approach, called *relifix*, for automated repair of software regressions. In particular, our contributions can be summarized as follows:

New Domain: We focus on program fixing on a new domain, specifically on repairing software regression errors. This domain was not studied in prior work in automated program repair, but various researches on fault localization [148, 147, 125, 42] and regression testing [131, 130] showed that this domain is important and widely represented in software development activities.

New Perspective: We formulate the software regression repair problem as a problem of reconciling problematic changes. We hypothesize that the fixes for regression bugs can be crafted using code from both the previous version

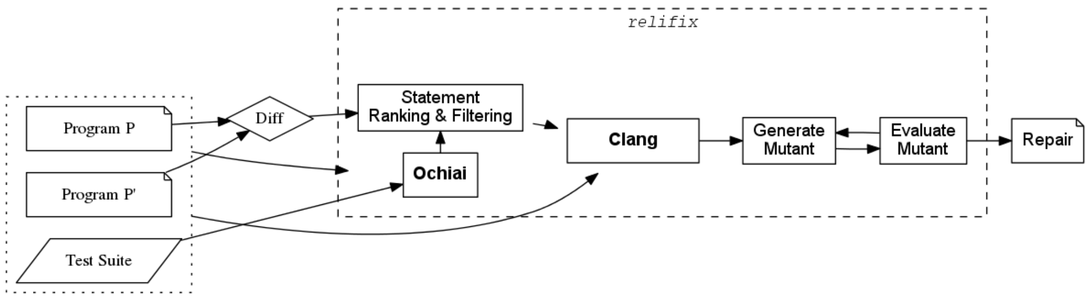


Figure 6-1: *relifix*'s Overall Workflow

(specifically, the preceeding version before the regression error occurs) and the current program version. We justify this formulation further in section 6.2. This formulation allows us to introduce fixes only to the changed lines.

Program Repair using previous version : Our approach leverages different program versions and code changes for guiding automated repair of regression bugs.

New contextual operators: We manually inspected 73 real regression bugs from the CoREBench benchmarks [48]. Our manual inspection produces a set of operators that uses information from two program versions, including changed statements and program location of the changed statements.

Evaluation: We applied *relifix* on eight open-source C projects (`Make`, `Find`, `Vim`, `Tar`, `Indent`, `Python` and `Perl`), which have well-developed and well-tested code. We compare the repairability of *relifix* with GenProg on 36 real regression errors. *relifix* successfully repaired 23 bugs, while GenProg only fixes five bugs. To compare the likelihood of both approaches in introducing new regressions, we also evaluated the regression rate of both approaches given the reduced test suite (test suite that contains the tests with different test behaviors in the two program versions). Our experimental results show that our approach is less likely to introduce new regressions compared to GenProg.

6.2 Repairing Regression as Reconciling Problematic Changes

When a developer fixes a regression error, he or she needs to execute the failing regression test, locate the cause of the test failure, and fix the current version of the program by referring to the previous version. This suggests that while repairing regression we probably try to find a fix that replicates the regression tests' behavior in the previous version. In fact, trivial fixes exist in the context of regression – execute the previous version for the failing regression test and run the current version for the remaining test cases. Such fixes are quick to issue, and they pass all tests in the test suite. However, they are costly to maintain and difficult to understand. Worse still, the number of program versions will double when another regression bug emerges. Thus, a good repair must be able to reproduce the behavior of the regression tests in the previous working version while maintaining the working functionality for the current version.

There are generally three types of software regressions.

Local A code modification breaks existing functionality in the changed program element.

Unmask A code modification unmasks an existing bug that had no effect to some test's behavior before the modification.

Remote A code modification introduces a bug in another unchanged program element.

Intuitively, if a functionality works in the previous version, the **Local** regression error can be fixed by rolling back to its old implementation in previous version. This intuition is supported by the *Revert to previous statement* operator derived from the Corebench benchmarks. Our hypothesis that fixes for regression errors exist in the immediate version before a regression error occurs, is in line with this intuition. Recent studies that speculate on the probabilities of locating fixes from multiple program versions [106, 43] further validates this hypothesis.

In contrast, if some source code in previous version had successfully hidden an existing bug, the **Unmask** regression error may be fixed by re-masking the problematic changes. In this case, fixing regression involves searching for a condition under which the problematic code modifications have no effect to the tests' behavior. This intuition is supported by the *Add condition* operator derived from the Corebench benchmarks. We hypothesize that the condition for hiding the problematic changes can be found among the program expressions in the current version. Some existing automated program repair techniques [90, 82] share similar hypothesis.

From these two observations, we formulate the software regression repair problem as a problem of reconciling problematic changes.

6.3 Experience about Real-life Regressions

We manually examined 73 benchmarks obtained from four subject programs to understand how software evolves during real-world regressions. We used the CoREBench benchmarks[1] for our manual investigation. This set of benchmarks is derived from regression errors that were systematically deduced from version control repositories and bug reports of four open source GNU projects (i.e., `Make`[2], `Grep`[3], `Findutils`[4],`Coreutils`[5]).

For each of the 73 benchmarks, we examined two set of code changes: (1) changes that occur between the version before the regression is introduced (i.e., version P1) and the version immediately after P1 (i.e., version P2); and (2) code changes that occur between the version before the regression is fixed (i.e., version

[1]`http://www.comp.nus.edu.sg/~release/corebench/`
[2]http://www.gnu.org/software/make/
[3]http://www.gnu.org/software/grep/
[4]http://www.gnu.org/software/findutils/
[5]http://www.gnu.org/software/coreutils/

Table 6.1: This table summarizes the number of code transformation operators that are used for fixing regression bugs.

Operator	Count
Add condition	27
Add statements	21
Use changed expression as input for other operator	13
Revert to previous statement	10
Replace with new expression	13
Remove incorrectly added statement	9
Change type	5
Add method	5
Add parameter	4
Add local variable	3
Swap changed statement with neighbouring statement	2
Negate added condition	1
Refactoring Analysis	1
Convert statement to condition variable statement	1
Add field	1
Total	116
Total Requires 2 operators	43

P3) and the version after the regression is fixed (i.e., version P4). We refer to each program version as P1, P2, P3 and P4 to denote the corresponding program version for the rest of the chapter. We then derived a set of general code transformations by comparing version P3 with P4. This set of code transformations form the operators that can be applied to repair the regression bugs. Table 6.1 shows the code transformation derived from the benchmarks, together with the number of benchmarks that uses the corresponding operator in regression bug fixing. The table is sorted with the most commonly used operators at the top of the table. The last row in the table shows that 43 repairs that involves two code transformation operators (the other 73-43 = 30 repairs involve only one code transformation operator). Overall, our manual inspection shows that information given by code changes between program versions are often included in human patches.

6.3.1 Contextual Operators that Use Information from Different Program Versions

Our manual inspection produces a set of operators that utilize information obtained from the previous version and from the code changes that occur between two consecutive program versions. We refer to this set of operators as *contextual operators* due to their references to different program versions. We next provide examples of our contextual operators.

Below are the details of each contextual operator:

Use changed expression as input for other operator This operator uses the program expressions that change (i.e., modify, add or remove) between two versions as input to other non-contextual operators (e.g., Add condition). The example below shows a patch in regression bug-fixing for `Coreutils`. The expression `max_range_endpoint < eol_range_start` was removed in the evolution from version P1 to version P2.

```
if(output_delimiter_specified && !complement && eol_range_start && ...
- && !is_printable_field(eol_range_start))
+ (max_range_endpoint < eol_range_start || !is_printable_field(eol_range_start)))
```

Listing 6.1: Example for use changed expression as input for other operator

Revert to previous statement This operator replaces newly added statements with the corresponding statements from old version, essentially reverting back some statements to the old version. The example in the following show a loop that was removed when `Make` evolves from version P1 to version P2. The developer added back the same loop to fix the regression in version P4.

```
- while(out > line && isblank((unsigned char)out[-1]))
- --out;
```

Listing 6.2: Revert to previous statement example

Remove incorrectly added statement This operator deletes program statements that were incorrectly added by the developer due to wrong bug fix.

Swap changed statement with neighboring statement This operator exchanges a changed statement with another consecutive statement. The changed statement serves as the pivot position for the exchange. Listing 6.3 shows the code modifications from version P3 to version P4. The `FindUtils` developer added `our_pred->est_success_rate=estimate_pattern_match_rate(...);` in version P2, and later changed the statement order relative to the statement`(*arg_ptr)++;`.

```
- (*arg_ptr)++;
our_pred->est_success_rate = estimate_pattern_match_rate(argv[*arg_ptr], 1);
+(*arg_ptr)++;
```

Listing 6.3: Example for Swap changed statement with neighbouring statement

Negate added condition This operator negates a branch condition that was previously added by the developer. For example, the `Grep` developer added the condition `included_patterns && !excluded_file_name(...)` in version P2. The bug is fixed by changing the added condition to `included_patterns && excluded_file_name(...)` in version P4.

Convert statement to condition variable statement This operator convert a statement with `Boolean` return type to a condition variable statement. Listing 6.4 presents the changes between version P3 and P4. The Coreutils developer forgets to check for the condition when the `set_acl` function returns 0. The fix requires converting the function call statement to a condition variable statement.

```
- set_acl(dst_name, dest_desc, 0666 &~cached_umask());
+ if(set_acl(dst_name, dest_desc, 0666 & ~cached_umask()) != 0)
```

Listing 6.4: Code changes between version P3 and P4 to illustrate convert statement to condition variable statement

6.4 Example

We illustrate how *relifix* can be used by showing three examples of fixes generated by *relifix* in three projects. The first two examples illustrate various operators involved in the fixes while the last example compares our generated fixes with the patches issued by developers. Consider first the `Vim` project [6], a popular editor that supports efficient text editing. A regression is introduced in version 7.2.50 of `Vim`, causing failures in two tests in `Vim`'s existing test suite. Listing 6.5 shows the repair generated by *relifix* with an application of the *Revert to previous statement* operator. This example demonstrates how *relifix* repairs a **Local** regression error. Note that there are approximately 18 *change hunks*[7] in the faulty source files between the two program versions, while the produced repair only modifies one hunk. Instead of reverting the entire source files to the previous version, the produced repair only reverts the faulty lines. This shows that our repair satisfies the criterion **C1**.

```
- fwrite(p, 1, (size_t)1, fd);
+ fwv &= fwrite(p, 1, (size_t)1, fd);
```

Listing 6.5: Example patch generated by *relifix* using "Revert to previous statement"

Consider next the GNU `Indent` project [8], a utility that formats C source files according to specific indent style. An **Unmask** regression occurs in version 2.2.10 of Indent, causing the buggy version to append too many newlines between variable declarations of a C source files. Listing 6.6 shows the repair generated by *relifix* using the "Add inverted condition" operator and "Use changed expression as input for other" operator. *relifix* first generates the condition `!(parser_state_tos->decl_on_line)` by negating an existing

[6]http://www.vim.org/

[7]A change hunk is a single sequence of contiguous source codes which has been modified from one version to another [102, 116, 120].

[8]http://www.gnu.org/software/indent/

```
// Patch that repairs the reduced test suite
+if (/* added */(!(parser_state_tos->decl_on_line)){
  ...
+}

// Patch that repairs all tests in the test suite
if (/* added */(!(parser_state_tos->decl_on_line)
+|| parser_state_tos->procname != "\0")}){
```

Listing 6.6: Example patches generated by *relifix* using two operators

`Boolean` expression. This intermediate patch passes the reduced test suite that contains one failing test, but it introduces new regression in other tests from the whole test suite. *relifix* then repairs the regression that it introduced by modifying the changed lines (i.e., the added condition). The final patch that passes all tests in the entire test suite is formed by appending another condition `parser_state_tos->procname != "\0"` (i.e., condition obtained by converting the assignment statement `parser_state_tos->procname = "\0"` to disequality) to the intermediate patch. This example illustrates the two-phase patch evaluation performed by *relifix*.

We next discuss one example that illustrates the differences between the patch generated by our approach and the patches issued by the developer. For this example, consider the GNU `Make` project[9], a tool that builds executables for a program from its source files. Two regression bugs (i.e., bug #12202 and bug #12267) are introduced with version 73e7767 of `Make`. Listing 6.7 shows the two patches generated in two different commits by the `Make` developer to fix the bugs. Listing 6.8 presents the code changes that causes the regression, while listing 6.9 shows the single patch generated by *relifix* that repairs both regression errors. To fix both regression bugs, *relifix* appends the condition `isintermed_ok` to one of the code change hunks. In this case, we consider that the code changes in listing 6.8 unmask a latent regression error and the added condition `isintermed_ok` has successfully masked both regression errors. While the fix generated by *relifix* is significantly different from the developer's patches, it may be preferable because

[9]http://www.gnu.org/software/make/

```
// Developer fixes for regression bug #12202
+ f->is_target = 1;
...
 +file->is_target = 1;

// Developer fixes for regression bug #12267
- register struct file *f = enter_file (imf->name);
+ register struct file *f = lookup_file (imf->name);
+ if (f != 0)
+   f->precious = 1;
+ else
+   f = enter_file (imf->name);
...
+if (!f->precious)
```

Listing 6.7: Example patches issued by the developer

```
//In file.c
+ f2->is_target = 1;
// In implicit.c
+ struct file *f;
...
- if (lookup_file (p) != 0
+ if (((f = lookup_file (p)) != 0 && f->is_target)
```

Listing 6.8: Example code change hunks between version 73e7767 and its preceeding versions

```
// Patches generated by relifix
- if (((f = lookup_file (p)) != 0 && f->is_target)
+ if (((f = lookup_file (p)) != 0 && (f->is_target || isintermed_ok))
```

Listing 6.9: Example patch generated by *relifix* using one operator

(1) it satisfies all criteria in Criteria 1 , and (2) it fixes both regression errors using only one patch.

6.5 Algorithm

Figure 6-1 shows the overall work-flow of our approach. Our *relifix* approach follows a three-step process. The first step takes as input the source code of the two program versions and the whole test suite with at least one failing test case that captures the regression error, and generates a ranked list of suspicious statements. The second step modifies the source code for the buggy version at the program location according to the list generated at previous step to produce a candidate repair. The last step builds the modified source code and re-executing the test suite to check if the generated repair passes all test cases. The main novelty in our work is in coming up with the contextual operators, and then applying them at the "right" places.

```
    Input: List of suspicious statements $RankList$
    Input: Set of test suite $T$, Reduced test suite $T_r \subseteq T$
    Input: List of contextual operators $O$
    Input: Set of program expression $E$
    Input: Period $P$ – the number of iterations for each location before considering next location
    Output: Program mutant that passes all test cases
 1  $iter \leftarrow 0$; $currO \leftarrow Shuffle(O)$; $currTS \leftarrow T_r$;
 2  $Tabu \leftarrow \{\}$; $currL \leftarrow 0$; $currC \leftarrow$ $original\ program$;
 3  while $repair\ not\ found$ do
 4      $currL \leftarrow$ $next\ top\ ranked\ location \in RankList$;
 5      $changedCount \leftarrow 0$;
 6      while $iter \leq P \wedge changedCount < size(CurrO) - 1$ do
 7          $op \leftarrow Dequeue(CurrO)$;
 8          if $op\ is\ parameterizable$ then
 9              /* select expression that are not in tabu */ repeat
                    $currE \leftarrow$ randomly chosen expr $\in E$;
                until $currE \notin Tabu$ ;
10              /* apply operator $op$ with $currE$ as parameter to candidate $currC$ at location $currL$ */
                $c \leftarrow currC.apply(op, currL, currE)$;
11          else
12              /* apply operator $op$ to candidate $currC$ at location $currL$ */
13              $c \leftarrow currC.apply(op, currL)$;
14          end
15          $Result \leftarrow Evaluate(c, currTS)$;
16          /* two-phrase mutant evaluation */
17          if $\forall r \in Result, r = passes$ then
18              $currTS \leftarrow T$;
19              $AResult \leftarrow Evaluate(c, currTS)$;
20              //check if repair is found
21              if $\forall a \in AResult, a = passes$ then
22                  $break$;
23              else
24                  /* $c$ causes new regressions, repair $c$ with the whole test suite */
25                  $currC \leftarrow c$;
26                  /* reset and re-shuffle O */ $currO \leftarrow$Shuffle(O);
27              end
28          else
29              /* check if the operator $op$ can be applied at location $currL$ and if candidate $c$ is
                compilable */
30              if $canBeApplied(op, currL) \wedge isCompiled(c)$ then
31                  /* reuse operator used in candidate c if it induces any change in the test execution
                    results for any test in the test suite */
                    if $op\ is\ parameterizable \wedge \exists r \in Result, diffResult(r)$ then
                        $Enqueue(CurrO, op)$;
                    end
32                  $changedCount \leftarrow changedCount + 1$;
33                  $iter \leftarrow iter + 1$;
34              else
35                  $Tabu \leftarrow Tabu \cup currE$;
36              end
37          end
38      end
39  end
```

Algorithm 4: *relifix* Mutant Generation and Evaluation Algorithm

6.5.1 Fault Localization

Our goal is to find a faulty program location that leads to the regression error. We first compute a suspiciousness score for each statement in the buggy program using the Ochiai formula [23] given below:

$$\text{suspiciousness}(s) = \frac{\text{failed}(s)^{10}}{\sqrt{\text{total failed}^{11} \times (\text{failed}(s) + \text{passed}(s)^{12})}}$$

We choose the Ochiai formula due to its effectiveness demonstrated by previous studies [24, 22]. To obtain the code changes between the two versions, we use the open-source GNU Diffutils [13]. Diffutils perform plain text comparisons to find the differences between two text files. After sorting the suspiciousness score for each statement, we remove the statements that do not lie within the set of modified statements from the list of suspicious statements. This step allows us to (1) reduce the inspection cost for the location to apply the fix and (2) increase the probability of applying our contextual operators. We share assumptions that are commonly used to evaluate testing and debugging techniques [65], that the error is among the changed statements. Note that this may lead to loss of residual latent error that are not manifested by the current test suite.

6.5.2 Mutant Generation and Evaluation

After *relifix* generates the list of suspicious statement, it uses our mutation generation component to generate a mutant. Our definition of program mutant is similar to prior work on mutation testing [64]: each mutant is defined as a program that are modified through some applications of mutation operators at some faulty locations.

[12]failed(s): Number of failing tests that executes statement s
[12]total failed: Total number of failing tests
[12]passed(s): Number of passing tests that executes statement s
[13]http://www.gnu.org/software/diffutils/

We collect a set of program expressions of type `Boolean` to be combined with all the parameterizable operators at later step. This set contains (1) all `Boolean` program expressions (that are within the program scope at the faulty location), and (2) expressions formed by converting the assignment operators in all the assignment statements (that are within the program scope at the faulty location) to the equality operators.

Algorithm 4 shows the pseudo-code of the *relifix* mutant generation and evaluation algorithm. Our *relifix* approach applies a randomly chosen contextual operator at a faulty location, evaluates each mutant against the current test suite, and iteratively repeats these steps until all the tests pass, no contextual operators apply, or the time limit is reached. Before applying each operator, we check whether the statement at the faulty location matches the given context for the operator and gather the required contextual information from both program versions. For example, if the faulty location has integer return type, the *Convert statement to conditional statement* operator cannot be applied.

We implement two optimizations for our random search algorithm: (1 – highlighted in first box) we store the index for the program expressions that do not compile in a *tabu* list, which helps us to avoid reusing program expressions that are not compilable. (2 – highlighted in second box) we enumerate the number of *well-formed mutants* (Definition 9). As each *well-formed mutant* indicates progress in generating the final repair, the operator involved in generating that mutant can be reused in generating the next set of mutants.

Definition 9 Well-formed mutants *are mutants that satisfy the following conditions:*

Compilable *Mutants generated should not generate any compilation errors*

Match Given Context *The program location and the structural type of the program element must match the context for the chosen operator used in* *generating the repair.*

Induce Change in Test Execution Results *Mutants generated should induce changes that affect the test behavior of some tests within the test suite.*

We modify the *clang-mutate* tool [14] to implement our mutant generation component. *clang-mutate* is built on top of the Clang [15] LibTooling library that offers utilities for parsing C programs and performing source-to-source transformations. Our mutant generation component satisfies the criterion **C2** as it modifies C source files directly to produce understandable code annotated with code comments (see Section 6.4 for examples of our generated code).

We implement all the contextual operators listed in section 6.3.1, including five non-parameterizable operators: (1) *Revert to previous statement*, (2) *Remove incorrectly added statement*, (3) *Swap changed statement with neighboring statement*, (4) *Negate added condition* , and (5) *Convert statement to condition variable statement.* We also implement four parameterizable operators (i.e, operators that needs to be with program expression), including (1) *Add condition to changed expression* (this operator combines the operator *Use changed expression as input for other operator* and the operator *Add condition*), (2) *Add condition* , and (3) *Add statement.* The first four parameterizable operators aim to find the condition for hiding an **Unmask** regression error.

Before applying contextual operators, we collect contextual information (program location, changed expression and type of changes) required to support the defined operations.

6.5.3 Test Case Prioritization and Reduced Test Suite

We evaluate each patch using a two-phase approach. We first execute the resulting patch against the reduced test suite. The reduced test suite (Figure 6-1) consists

[14]https://github.com/eschulte/clang-mutate
[15]http://clang.llvm.org/

Table 6.2: Subject Programs and Their Basic Statistics

Subject	Description	Size in kLOC	Bug Introducing Commit	Bug Report	PT Size/Test Suite Size
`Vim`	Text Editor	150	f80e67 [18]	[17]	1/74
			509890 [16]	[15]	2/73
			a3552c [20]	[19]	1/71
			220906 [14]	[13]	1/72
`CPython`	Programming language	407	b878df [4]	[3]	1/268
			5b0fda [2]	[1]	1/286
`Perl`	Programming language	271	dca606 [10]	[9]	1/159
			bb9ee97 [8]	[7]	1/159
`Indent`	Source code re-format utilities	15	2.2.10 [6]	[5]	1/159
`Tar`	Archives manipulation utilities	21	1.14 [12]	[11]	1/15
`Findutils`	Directory searching utilities	18	6 versions	10 bugs	[1,10]/1054
`Make`	Program executable generation utilities	35.3	12 versions	15 bugs	[1,2]/528

Table 6.3: Operators used in fixes generated by *relifix*

Subject	Operators Used	Number of Operators	Change Hunks
`Vim-f80e67 [18]`	Swap	1	1
`Vim-509890 [16]`	Revert	1	1
`Vim-a3552c [20]`	AddIf	1	1
`Vim-220906 [14]`	-	-	-
`Cpython-b878df [4]`	Revert	1	1
`Cpython-5b0fda [2]`	AddIf	1	1
`Perl-dca606 [10]`	Revert	1	3
`Perl-bb9ee97 [8]`	-	-	-
`Indent-2.2.10 [6]`	AddIf & AddOld	2	1
`Tar-1.14 [12]`	Revert	1	2
`Findutils`	4 AddIf, 3 Revert, 1 Insert	8/8	10/8
`Make`	3 AddIf, 3 AddNegated, 1 Revert	7/7	7/7
Total/Mean	10 AddIf, 8 Revert, 3 AddNegated, 1 AddOld, 1 Swap, 1 Insert	24/23=1.04	28/23=1.22

of tests with different execution results in both versions, namely the *progression tests* (tests that fail in the previous version but pass in current version) and the failing *regression tests* (tests that fail in the previous version but pass in current version). When a patch that passes both set of tests is found, we then check if it introduces any new regression by re-executes all tests in the test suite.

We prioritize test cases using the reduced test suite based on the assumption that test cases that evolve across the two versions are more likely to fail in future execution [131]. Our goal is to save the time spent in evaluating each patch against the entire test suite, and to allow more candidate mutants to be generated within the time limit.

6.6 Experimental Evaluation

We perform an evaluation on real regressions by comparing the effectiveness of our approach with GenProg [90]. To evaluate the effectiveness of our approach, we aim to address the following research questions:

Table 6.4: Overall Repairability (i.e., RP) and Regression Rate (i.e. RR) for *relifix* and GenProg on the new Subject Programs

Subject	*relifix*		*relifix*	GenProg		GenProg	*rGenProg*		*rGenProg*
	Reduced test suite		Whole test suite	Reduced test suite		Whole test suite	Reduced test suite		Whole test suite
	RP	RR		RP	RR		RP	RR	
`Vim-f80e67 [18]`	1	0	1	0	0	0	0	0	0
`Vim-509890 [16]`	1	0	1	0	0	0	0	0	0
`Vim-a3552c [20]`	1	0	1	0	0	0	0	0	0
`Vim-220906 [14]`	0	0	0	0	0	0	0	0	0
`Cpython-b878df [4]`	1	0	1	0	0	0	0	0	0
`Cpython-5b0fda [2]`	1	0	1	0	0	0	0	0	0
`Perl-dca606 [10]`	1	0	1	0	0	0	0	0	0
`Perl-bb9ee97 [8]`	0	0	0	0	0	0	0	0	0
`Indent-2.2.10 [6]`	1	1	1	0	0	0	0	0	0
`Tar-1.14 [12]`	1	0	1	0	0	0	0	0	0
`Findutils`	8/10	0/8	8/10	2/10	2/2	5/10	0	0	5/10
`Make`	8/15	1/8	7/15	0/15	0/15	0/15	0/15	0/15	0/15
Total	24/35	2/24	23/35	2/35	2/2	5/35	0/35	0/35	5/35

RQ1 How many regression errors can our approach repair compared to GenProg?

RQ2 Given only the test cases that evolves across the two versions, how likely is our approach to introduce new regressions, as compared to GenProg?

RQ3 Are our produced fixes suitable for patching latent regression errors or for patching errors due to code changes?

RQ4 Can we fix regression errors by making only small code changes without introducing new regressions?

The first question (RQ1) assesses the repairability of both approaches in the context of regression error, given the whole test suite. The second question (RQ2) evaluates the likelihood of both approaches in producing other test failures after repairing a regression error based on a reduced test suite (see Subsection 6.5.3 for definition of reduced test suite). The third question (RQ3) asks if our approach is more effective in fixing existing errors (i.e., latent errors) compared to new errors that are introduced due to the code modification. Lastly, the fourth question (RQ4) validates our hypothesis that mutant with small code changes (according to our below definition of small code changes) are less likely to introduce new regressions.

At the beginning of this chapter, we presented Criteria 1 which guides our regression repair. We now present Criteria 2 which checks whether our approach produces repairs with small code changes, and further clarifies the first property mentioned in Criteria 1.

Criteria 2 *Our repair should introduce small code changes, such that each repair should satisfy the following criteria:*

Least number of change hunks *Our repair should introduce the least number of change hunks. A change hunk is a single sequence of contiguous source codes which has been modified from one version to another [120, 116, 102].*

Least number of applied operators *Our repair should apply the least number of operators to the original program.*

6.6.1 Experimental Setup

We evaluate *relifix* on 35 real regression errors collected from seven open-source C projects. Table 6.2 lists information about these projects. The last column in Table 6.2 shows the number of progression tests(PT) and the total number of tests in the whole test suite. For each regression error, we run both *relifix* and GenProg [87] to generate repair. GenProg provides several options that control the fault localization scheme used (e.g., path-based and line-based). We use the line-based fault localization scheme and provide the changed lines for the faulty locations to simulate a specialized version of GenProg for fixing regression errors (we call this *rGenProg*). We then compare the repairability (RQ1) of all the three approaches: *relifix*, GenProg and *rGenProg*. We also compare how likely each approach introduces new regressions.

All subject programs in Table 6.2 are utilities or libraries that are commonly used. As we perform our evaluation only on real regression errors, we select two subjects (i.e., `Findutils` and `Make`) from the CoREBench benchmarks [48], (2) two subjects (i.e., `Indent` and `Tar`) from [147] and one subject used in GenProg experiments. We also add two additional subjects (i.e., `Vim` and `Perl`). We choose these regression errors because (S1) they contain detailed bug report that specifies the bug introducing commit, and (S2) all the regression errors are reproducible with at least one test that passes in previous version and fails in the faulty version. We exclude 8 bugs (i.e, 5 bugs from `Findutils` and 3 bugs from `Make`) from the CoREBench benchmarks as they violate (S2).

For running GenProg, we reuse the same parameters stated in [87]. One significant difference is that we switch to the deterministic adaptive search algorithm (AE) [141] to control potential randomness. Each run of relifix, GenProg and *rGenProg* is terminated after one hour or when a repair is found.

All experiments were performed on a machine with a dual-core Intel i5-2520M 2.50GHz processor and 4GB of memory.

6.6.2 Repairability (RQ1)

Table 6.4 presents the repairability and the regression rate for the 10 individual regression errors (subjects *outside* CoREBench). For all tables, we denote x bugs out of a total of y bugs with x/y. The last row of Table 6.4 shows the aggregated repairability (i.e., repairability for all bugs) and the aggregated regression rate for *relifix* and GenProg on the two CoREBench subjects. Given the entire test suite, *relifix* successfully repair 15 out of 25 regression errors (as stated in the "Whole test suite" column in Table 6.4) for the two CoREBench subjects, while GenProg only fixes 5 bugs. For the 10 regression errors in subjects outside CoREBench in Table 6.4, *relifix* fixes 8 out of 10 regression errors but GenProg fails to generate any repair for all the 10 bugs. Although GenProg is able to fix approximately half of the evaluated programs in their recent study in [87], GenProg can only fix 14.3% (i.e., 5 out of 35 bugs) of all the evaluated subjects. In comparison, *relifix* repairs 65.7% (i.e., 23 out of 35 bugs). We attribute the low repairability of GenProg to (1) the high complexity of the real regression errors as some regression errors in CoREBench has fairly high error complexity [48], (2) the lack of availability of fixes within the same program (i.e., the fixes may only exist in the previous version of the same program as argued in the recent paper [43]).

Next, we compare the repairability of both *relifix* and GenProg for the reduced test suites (read Subsection 6.5.3 for definition of reduced test suite). Given the reduced test suite, GenProg generates only 2 out of 35 repairs, whereas *relifix* produces 24 out of 35 repairs. In comparison, *rGenProg* that fixes only the changed lines fails to produce any repair with both set of test suites. The repairability of GenProg decreases (i.e., from generating 5 repairs to generating only 2 repairs) when provided with the reduced test suite compared to the whole test suite because the search space for the faulty locations increases significantly due the reduced test suite. *relifix* does not suffer from the same problem as (1) it reduces the search space for the faulty location by ignoring program location that are not within the set of code modifications, and (2) it further refines the fix location by applying fixes to each faulty location iteratively for a limited period of time.

On average, GenProg requires 44 patch evaluations (i.e., patch trials in [126]) before generating a repair while *relifix* takes 25 mutant evaluations for producing the final repair.

6.6.3 Regression Rate (RQ2)

The "RR" columns in Table 6.4 represent the regression rate of each approach using only the reduced test suite, while the "RP" columns denotes the measurement given the whole test suite. We define regression rate as the likelihood of introducing new regression errors after fixing all tests in the reduced test suite. We calculate the regression using the formula below:

$$RR = \frac{Number\ of\ Repairs\ that\ introduce\ new\ regression}{Number\ of\ All\ Generated\ Repairs} \tag{6.1}$$

In total, *relifix* introduces new regressions in 2 out of 24 repairs with the reduced test suite (see Subsection 6.5.3 for explanation of reduced test suite). In comparison, GenProg introduces regression in all repairs (i.e., 2 out of 2) generated with the reduced test suite, while *rGenProg* does not generate any repair with the reduced test suite.

We next discuss the regressions introduced by both approaches. *relifix* causes a regression in a test that check if parallel execution of `Make` works correctly when fixing the regression error for `Make-bug-#39203`[16]. This regression cannot be fixed when executing *relifix* on the entire test suite. *relifix* also introduces new regression when fixing the `Indent` program [5]. As discussed in section 6.4, this regression can be repaired given the entire test suite. GenProg causes 45 test failures out of 80 tests in the entire test suite when fixing the regression error for the `Findutils-bug-#18222`[17], while it makes one out of 81 tests fail in the whole test suite when repairing the `Findutils-bug-#19605`[18]. We classify the fixes for `Findutils-bug-#18222` as a bad fix because it causes more failures compared to the original buggy versions that has only one test failure. We think that the high regression rate of GenProg may be due to (1) the imprecise fault localization used, and (2) the massive number of modifications in the patches. The speculation regarding the problem with fault localization is supported by the fact that *rGenProg* , which fixes only the changed lines, does not share similar regression rate as the original GenProg.

6.6.4 Repairability of Latent Errors versus New Errors (RQ3) and the Simplicity of the Generated Repair (RQ4)

Table 6.3 lists the operators involved in the fixes generated by *relifix*. The table below explains the abbreviation used to denote the name of the operator in Table 6.3.

[16]`http://savannah.gnu.org/bugs/?39203`
[17]`http://savannah.gnu.org/bugs/?18222`
[18]`http://savannah.gnu.org/bugs/?19605`

Revert	**Revert to previous statement**
Swap	**Swap changed statement with neighboring statement**
AddIf	**Add condition**
AddOld	**Add condition to changed expression**
AddNegated	**Add negated condition**
Insert	**Add statement**

As the classification of a regression error as a latent error or a new error caused by code modifications requires deep understanding of the regression error, we cannot provide a precise answer for RQ3. However, since the *Revert to previous statement* operator are directly related to the **Local** regression error that are caused by a broken existing functionality, we can provide an rough estimate of latent errors repaired by *relifix* by calculating the number of fixes generated using the *Revert to previous statement* operators. Based on this estimation, 8 out of 24 generated fixes are latent errors. This suggest that *relifix* can fix both types of errors (i.e., latent errors and new errors due to code modification) equally well.

We hypothesize that repair that makes only small code changes are less likely to introduce new regressions. Hence, we check if our repair satisfies all criteria in Criteria 2. The last two columns in Table 6.3 shows the number of operators used and the number of change hunks involved for each generated repair. As shown in the last row of the table, the mean value for the number of operator used in the final repair is $24/23 \approx 1.04$, while the mean value for the number of change hunks involved in the generated fixes is $28/23 \approx 1.22$. The two low mean values suggest that most fixes generated by *relifix* involve making only small code changes to the original program. While we do not have enough data to support the claim that the small code changes are less likely to introduce new regressions, we observe that all the new regressions produced by *relifix* and GenProg with the reduced test suite involve introducing more than one change hunks and applying more than one mutation operators in the generated fixes.

6.7 Threats To Validity

We identify the following threats to the validity of our experiments:

Subjects While our evaluation uses subjects of various sizes and from various sources, we reuse two subjects, in which we obtained the set of contextual operators, for evaluation. This selection compensates for the lack of benchmarks with real regression errors but it may be biased towards *relifix* due the operators derived in our manual inspection. However, we note that the operators used in generating repair by *relifix* in those subjects are generally different from the original operators observed due to the gap between the error introducing commit and the bug-fixing commit.

Contextual Operators We derived the set of contextual operators from a benchmark that contains only C programs. The same set of operators may not be generalized to other languages. As we investigated only open-source projects, the operator may not be generalized to closed-source projects.

Readability of Patches We claim that the code generated by *relifix* are understandable with some examples in Section 6.4. This claim relies on the intuition that source code annotated with comments are generally more readable than CIL (i.e., Common Intermediate Language) file produced by other tools (e.g., GenProg). We leave detailed evaluation of the readability of automatically generated patches as future work.

Time We restrict the time limit for evaluating *relifix* and GenProg to one hour due to limited resources. The repairability for both tools may increase given a longer timeout.

6.8 Related Work

Fault Localization There are several fault localization techniques that utilizes multiple program versions [28, 42, 125, 129, 147, 148]. DARWIN uses the previous version to localize the regression bug using dynamic symbolic execution in both program versions [125]. Delta debugging isolates failure-inducing circumstances that are responsible to test failures using a divide-and-conqueer algorithm [148]. It can be used to repair regression errors by first isolating problematic changes and reverting these changes. This way of repairing regressions is part of our set of contextual operators.

Automatic Program Repair A few automated program repair techniques have been proposed to reduce the time and effort required to fix software bugs. Arcuri and Yao proposed adopting evolutionary algorithms for automatic program generation [34]. Weimer et al. proposed using genetic programming for automated program repair [67, 90]. GenProg generates fixes using statements that exist within the same program, while we utilize statements in the previous program version to repair regression errors. After generating a repair, GenProg requires a separate minization step to produce patches with simpler code changes. Our repair algorithm does not require this step as it generates repairs by applying only a small number of operators to some changed lines.

Kim et al. proposed an automated patch generation that utilizes common fix templates learned from manual inspection of human patches [82]. Their user study demonstrated that patches generated by PAR are more acceptable than patches produced by GenProg. While we also derive our contextual operators from human patches, our operators are more general as they are not designed to fix a particular defect class (i.e., null pointer exceptions and array out-of-bound errors) [112].

Wei et al. leverages software contract to automatically repair faulty Eiffel classes [140]. Our approach does not require manually written program contract as it utilizes syntactical information from the previous program version that may serve as an implicit specification.

Nguyen et al. employs program synthesis for discovering the intent pieces of code required for fixing the buggy program [117]. While we employed random search for the condition to hide the **Unmask** regression errors due to scalability issues, we believe that program synthesis may be used to generate the required condition.

Repair that uses domain specific knowledge There are several program repair techniques that utilizes domain specific information. In particular, PACHIKA relies on differences between passing and failing runs to automatically infer object behavioral model from Java program and produce fixes by either inserting or deleting method calls [61]. BugFix is a tool that incorporates information gathered from several debugging sessions in order to increase precision for producing bug-fix suggestion [74]. R2Fix closes the loop between bug report submission and patch generation by automatically classify the type of bug discussed in bug report and extracting pattern parameters to generate fixes based on a several predefined fix patterns [93]. PHPRepair fixes malformed HTML generation errors by encoding the string output for each test case execution as a constraint over variables corresponding to constant prints in the program and uses a constraint solver to generate string modification [132]. Martinez and Monperrus mine semantic code modifications(which they referred to as repair actions) from human patches and attach a probability distribution to the mined repair actions [104]. None of these techniques focus on repairing regression errors.

Utilizing Previous Version as a fix Various studies speculate on the possibility of locating fixes from various program versions [27, 43, 106]. Martinez et. al. demonstrates that statements or expressions that are required for fixing exist in previous commits of the programs [106]. Barr et al. analyzes commits from several open-source Java projects, and they found that commits can be reconstructed from codes from the preceeding versions [43]. Alkhalaf et al. uses semantic differences between a reference function and a target function to synthesize a validation, a length, and a sanitization patch for repairing web-application code [27]. We share similar observation that the previous program version may be used for automatic repair generation, specifically in fixing **Local** regression bugs. The key difference is that some of our contextual operators use program location information from the previous version, while other operators utilize program expressions from the current version. Furthermore, to the best of our knowledge, ours is the first work to *develop* a repair method and tool specifically for patching regression bugs.

6.9 Chapter Summary

In this chapter, we proposed *relifix*— an approach of automated repair of software regression. This was achieved by considering the regression repair problem as a problem of reconciling problematic changes. We justified our claim using a set of contextual operators derived from our manual inspection of 73 real software regressions. Our evaluation on 36 real regression bugs shows that *relifix* can repair 23 bugs, while GenProg only fixes five bugs. Our experimental results with the reduced test suite suggests that our approach is less likely to introduce new regression compared to GenProg.

Chapter 7

Emerging Applications: Repairing Crashes in Mobile Apps

As all repair operators studied in Chapter 3 are program transformations that mimic programming errors in C programs, a different set of repair operators may be required for fixing different types of applications.

This chapter presents a novel program repair approach that automatically generates fixes for Android applications. As event transitions in Android applications are governed by Android Activity/Fragment lifecycle management rules, violations of these rules may cause crashes in Android applications. To address this problem, we design a new set of repair operators that leverage Android Activity/Fragment lifecycle information for fixing crashes in Android applications.

7.1 Introduction

Smartphones have become pervasive, with 492 millions sold worldwide in the year of 2011 alone [70]. Users tend to rely more on their smartphones to conduct their daily computing tasks as smartphones are bundled with various mobile applications. Hence, it is important to ensure the reliability of mobile applications running in their smartphones.

Testing and analysis of mobile apps, with the goal of enhancing reliability, have been studied in prior work. Some of these works focus on static and dynamic analysis of mobile apps [25, 37, 59, 145], while other works focus on testing of mobile apps [29, 30, 98, 100, 128].

To further improve the reliability of mobile applications, several approaches go beyond automated testing of apps by issuing security-related patches [35, 114]. While fixing security-related vulnerabilities is important, a survey revealed that most of the respondents have experienced a problem when using a mobile application, with 62 percent of them reported a crash, freeze or error [21]. Indeed, frequent crashes of an app will lead to negative user experience and may eventually cause users to uninstall the app. In this chapter, we study automated approaches which alleviate the concern due to app crashes via the use of *automated repair*.

Recently, several automated program repair techniques have been introduced to reduce the time and effort in fixing software errors [82, 96, 108, 117, 126, 141]. These approaches take in a buggy program P and some correctness criterion in the form of a test-suite T, producing a modified program P' which passes all tests in T. Despite recent advances in automated program repair techniques, existing approaches cannot be directly applied for fixing crashes found in mobile applications due to various challenges.

The key challenge in adopting automated repair approaches to mobile applications is that the quality of the generated patches is heavily dependent on the quality of the given test suite T. Indeed, any repair technique tries to patch errors so as to achieve the intended behavior. Yet, in reality, the intended behavior is incompletely specified, often through a set of test cases. Thus, repair methods attempt to patch a given buggy program, so that the patched program passes all tests in a given test-suite T (We call repair techniques that use test cases to drive the patch generation process *test-driven repair*). Unsurprisingly, test-driven repair may not only produce incomplete fixes but the patched program may also end up introducing new errors, because the patched program may fail tests outside T, which were previously passing [134, 138]. Meanwhile, several unique properties of test cases for mobile applications pose unique challenges for test-driven repair. First, regression test cases may not be available for a given mobile app A. While prior researches on automated test generation for mobile apps could be used for generating crashing inputs, regression test inputs that ensure the correct behaviors of A are often absent. Secondly, instead of simple inputs, test inputs for mobile apps are often given as a sequence of UI commands (e.g., clicks and touches) leading to crashes in the app. Meanwhile, GUI tests are often flaky [97, 109]: their outcome is non-deterministic for the same program version. As current repair approaches rely solely on the test outcomes for their correctness criteria, they may not be able to correctly reproduce tests behavior and subsequently generate incorrect patches due to flaky tests.

Another key challenge in applying recent repair techniques to mobile applications lies on their reliance on the availability of source code. However, mobile applications are often distributed as standard Android .apk files since the source code for a given version of a mobile app may not be directly accessible nor actively maintained. Moreover, while previous automated repair techniques are applied for fixing programs used by developers and programmers, mobile applications may be utilized by general non-technical users who may not have any prior knowledge regarding source code and test compilations.

We present a novel framework, called *Droix* for automated repair of crashes in Android applications.

Our contributions can be summarized as follows:

Android repair: We propose a novel Android repair framework that automatically generates a fixed APK given a buggy APK and a UI test. Android applications were not studied in prior work in automated program repair, but various researches on analysis [25, 37, 59, 145] and automated testing [29, 30, 98, 100, 128] illustrate the importance of ensuring the reliability of Android apps.

Repairing UI-based test cases: Different from existing repair approaches based on a set of simple inputs, our approach fixes a crash with a single UI event sequence. Specifically, we employ techniques allowing end users to reproduce the crashing event sequences by recording user actions on Android devices instead of writing test codes. The crashing input could be either recorded manually by users or automatically generated by GUI testing approaches [98, 136].

Lifecycle-aware transformations Our approach is different from existing test-driven repair approaches since it does not seek to modify a program to pass a given test-suite. Instead, it seeks to repair the crashes witnessed by a single crashing input, by employing program transformations which are likely to repair the root-causes behind crashes. We introduce a novel set of lifecycle-aware crashing resolving strategies that could automatically patch crashing android apps by using management rules from the activity lifecycle and fragment lifecycle.

Evaluation: We propose DroixBench, a collection of 24 reproducible crashes in 15 open source Android apps. Our evaluation on 24 defects shows that Droix could repair 15 out of the 24 bugs, and seven of these repairs are syntactically equivalent to the human patches.

7.2 Background: Lifecycle in Android

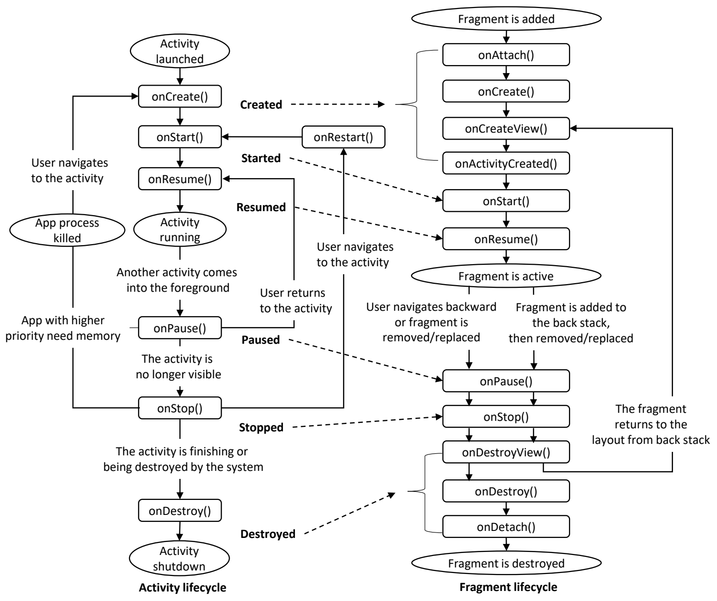


Figure 7-1: Activity Lifecycle, Fragment Lifecycle and the Activity-Fragment Coordination

Different from Java programs, Android applications do not have a single main method. Instead, Android apps provide multiple entry points such as *onCreate* and *onStart* methods. Via these methods, Android framework is able to control the execution of apps and maintain their lifecycle.

Figure 7-1 shows the lifecycle of activity and fragment in Android. Each method in Figure 7-1 represents a lifecycle *callback*, a method that gets called given a change of state. Lifecycle transition obeys certain principles. For instance, an activity with the paused state could move to the resumed state or the stopped state, or may be killed by the Android system to free up RAM.

A *fragment* is a portion of user interface or a behavior that can be put in an Activity. Each fragment can be modified independently of the *host activity* (activity containing the fragment) by performing a set of changes. For a fragment, it goes through more states than an Activity from being launched to the active

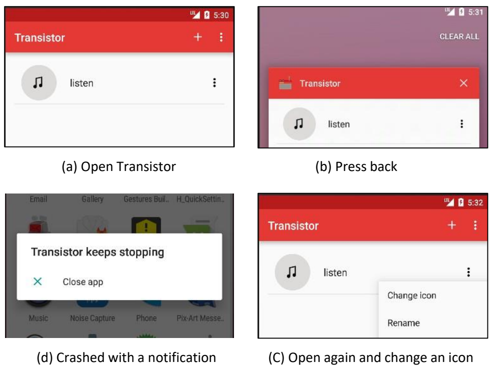


Figure 7-2: Continuous snapshots of a crash in Transistor.

state, e.g., *onAttach* and *onCreateView* states.

The communication between an activity and a fragment needs to obey certain principles as well. A fragment is embedded in an activity and is allowed to communicate with the host activity after it being attached. The available states of a fragment are determined by the state of its host activity. For instance, a fragment is not allowed to reach the *onStart* state before the host activity enters the *onStart* state. A violation of these principles may cause crashes in Android apps.

7.3 A Motivating Example

In this section, we show an example app, and its crash, to illustrate the workflow of our automated repair technique. The crash occurred in Transistor, a radio app for Android with 63 stars in GitHub. According to the bug report[1], Transistor crashes when performing the event sequence shown in Figure 7-2: (a) starting Transistor; (b) shutting it down by pressing the system back button; (c) starting Transistor again and changing the icon of any radio station. Then, it crashes with a notification "Transistor keeps stopping"(d). Listing 7.1 shows the log relevant to this crash. The stack trace information in Listing 7.1 suggests that the crash is caused by `IllegalStateException`.

Our automated repair framework, Droix performs analysis of the Activity-

[1]https://github.com/y20k/transistor/issues/21

Fragment coordination (dashed lines in Figure 7-1) and reports potential violations in the communication between a fragment and its host activity. Our manual analysis of the source code for this app further reveals that the crash occurs because the fragment attempts to call an inherited method *startActivityForResult* at line 482, which indirectly invokes a method of its host activity. However, the fragment is detached from the previous activity during the termination of the app and needs to be attached to a new activity in the restarting app. The method invocation occurs before the new activity has been completely created and leads to the crash.

```
FATAL EXCEPTION: main
Process: org.y20k.transistor, PID: 2416
java.lang.IllegalStateException:
   Fragment MainActivityFragment{82e1bec} not attached to Activity
at android...startActivityForResult(Fragment.java:925)
at y20k...selectFromImagePicker(MainActivityFragment.java:482)
```

Listing 7.1: Stack trace for the crash in Transistor

```
     +if(getActivity()!=null)
482: startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

Listing 7.2: Droix's patch for the crash in Transistor

```
     -startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
482: +mActivity.startActivityForResult(pickImageIntent, REQUEST_LOAD_IMAGE);
```

Listing 7.3: Developer's patch for the crash in Transistor

Droix defines specific repair operators based on our study of crashes in Android apps and the Android API documentation (see Section 7.4). One of the transformation operators identified through our study, `GetActivity-check`, is designed to check if the activity containing the fragment has been created. The condition `getActivity()!=null` prevents the scenario where a fragment communicates with its host activity before the activity is created.

Listing 7.2 shows the patch automatically generated by Droix. With the patch, method `startActivityForResult` will not be invoked if the host activity has not been created. The related function (i.e., changing station icon) works well after our repair. In contrast, although the developer's patch does not crash on the given input, it introduces regressions. Listing 7.3 shows the developer's patch where `mActivity` is a field of the fragment referencing its host activity. When restarting the app, this field still points to the previously attached activity. The developer's patch explicitly invokes `startActivityForResult` method of the previously attached activity instead of the newly created activity. After applying the Developer's patch, a user reports that the system back button no longer functions correctly when changing the station icon (i.e., pressing the back button does not close the app but mistakenly opens a window for selecting images). Specifically, the user reports the following event sequence when the app fails to function properly: `open Transistor` → `tap to change icon` → `press back twice` → `open Transistor` → `tap to change icon` → `press back twice`. We test the fixed APK generated by Droix with the user-provided event sequence and we observe that our fixed APK does not exhibit the faulty behavior reported by the user. In this case, we believe that the patch generated by Droix works better than the developer's patch.

7.4 Identifying Causes of Crashes in Android Applications

To investigate the possible causes of crashes in Android applications, we perform manual inspection of open-source Android apps on GitHub. Our goal is to identify a set of common causes for Android crashes. We first obtain a set of popular Android apps by crawling GitHub. Particularly, we search for the word "android app" written in Java using the GitHub API [2]. For each app repository, we search

[2]https://developer.github.com/v3/

Table 7.1: Root cause of crashes in Android apps

Category	Specific reason	Description	GitHub Issues (%)	Exception Type	Category Total (%)
Lifecycle	Configuration changes	activity recreation during configuration changes	5.61	NullPointer	14.02
	Stateloss	transaction loss during commit	2.80	IllegalState	
	GetActivity	activity-fragment coordination	2.80	IllegalState	
	Activity backstack	inappropriate handling of activity stack	1.87	IllegalArgument	
	Save instance	uninitialized object instances in onSaveInstance()	0.93	IllegalState	
Resource	Resource-related	resource type mismatches	10.28	NullPointer	16.82
	Resource limit	limited resources	4.67	OutOfMemory	
	Incorrect resource	retrieve a wrong resource id	1.87	SQLite	
Callback	Activity-related	missing activities	7.48	NullPointer	17.76
	View-related	missing views	6.54	NullPointer	
	Intent-related	missing intents	3.74	NullPointer	
	Unhandled callbacks	missing callbacks	2.80	NullPointer	
Others	Missing Null-check	missing check for null object reference	12.15	NullPointer	52.34
	External Service/Library	defects in external service/library	8.41	NullPointer	
	Workaround	temporary fixes for defect	4.67	IndexOutOfBound	
	API changes	API version changes	2.80	SQLite	
	Others	project-specific defects	24.30	-	

for closed issues (resolved bug report) with the word “crash”. We only focus on closed issues because those issues have been confirmed by the developers and are more likely to contain fixes for the crashes. From the list of closed issues concerning app crashes, we further extract issues that contain at least one corresponding commit associated with the crash. The final output of our crawler is a list of closed issues concerning crashes that have been fixed by Android developers. Overall, our crawler searches through 7691 GitHub closed issues where 1155 (15%) of these issues are related to crashes. The relatively high percentage of crash-related issues indicates the prevalence of crashes in Android apps. Among these 1155 issues, 107 of these issues are annotated with corresponding bug-fixing commits. We manually analyzed all 107 issues and attempted to answer two questions:

Q1: What are the possible root causes and exceptions that lead to crashes in Android apps?

Q2: How does the complexity of activity/fragment lifecycle affect crashes in Android apps?

We study $Q2$ because a survey of Android developers suggests that the topmost reasons (47%) for `NullPointerException` in Android apps occur due to the complexity of activity/fragment lifecycle [60]. Our goal is to identify a set of generic transformations that are often used by Android developers in fixing Android apps. To gain deeper understanding of the root causes of each crash (Q1) and to identify the affect of activity/fragment lifecycle on the likelihood of introducing crashes (Q2), we manually examine lifecycle management rules in the official Android API documentations [3].

Our results show that the most common exceptions among the studied apps are:

- NullPointerException (40.19%)
- IllegalStateException (7.48%)

The high percentage of `NullPointerException` confirms with the findings of prior study of Android apps [58].

Table 7.1 shows the common root causes of crashes in Android apps we investigated. Column "Category" in Table 7.1 describes the high-level causes of the crashes, while the "Specific reasons" column gives the specific causes that lead to the crash. The last column (Category Total (%)) presents the total percentage of issues that fits into a particular category. Overall, 14.02% of crashes in our study occur due to the violation of management rules for Android Activity/Fragment lifecycle. The reader can refer to Section 7.5 on the explanation of these lifecycle-related crashes. Meanwhile, 16.82% of the investigated crashes are due to improper handling of resources, including resources either not available (Resource-related) or limited resources like memory (Resource limit). Furthermore, improper handling of callbacks contributes to 17.76% of crashes. Note that this "Callback" category denotes implementation-specific problems of different components in Android library (e.g., Activity, View and Intent). Among 40.19% of NullPointerExceptions thrown in these crashing apps, only 12.15% is related to missing the check for `null` objects (Missing Null-check). Interestingly, 4.67% of the GitHub issues include comments by Android developers acknowledging the fact that the patch issued are merely temporary workaround (Workaround) for these crashes that

[3]https://developer.android.com/guide/components/activities/activity-lifecycle.html

Table 7.2: Supported Operators in Droix

Operator	Description
S1: GetActivity-check	Insert a condition to check whether the activity containing the fragment has been created.
S2: Retain object	Store objects and load them when configuration changes
S3: Replace resource id	Replace resource id with another resource id of same type.
S4: Replace method	Replace the current method call with another method call with similar name and compatible parameter types.
S5: Replace cast	Replace the current type cast with another compatible type.
S6: Move stmt	Removes a statement and add it to another location.
S7: Null-check	Insert condition to check if a given object is null.
S8: Try-catch	Insert try-catch blocks for the given exception.

may require future patches to completely resolve the crash.

Overall, Table 7.1 shows that the complexity of activity/fragment lifecycle and incorrect resource handling are two general causes of crashes in Android apps. Moreover, “Missing Null-check” in the “Other” category also often leads to crashes in Android apps.

7.5 Strategies to Resolve Crashes

Our manual analysis of crashes in Android apps identifies eight program transformation operators which are useful for repairing these crashes. Table 7.2 gives an overview of each operator derived through our analysis. As “Missing Null-check” is one of the common causes of crashes in Table 7.1, we include this operator (S7: Null-check) in our set of operators. Another frequently used operator (5%) that fixes crashes that occur across different categories in Table 7.1 is inserting exception handler (S8: Try-catch) which we also include into our set of operators. We now proceed to discuss other program transformation operators in Table 7.2 and the specific reasons of crashes associated with each operator in this section.

Retain stateful object Configuration changes (e.g., phone rotation and language) cause activity to be destroyed and recreated which allows apps to adapt to new configuration (transition from `onDestroy()`$\rightarrow$ `onCreate()`).

According to Android documentation [4], developer could resolve this kind of crashes by either (1) retaining a stateful object when the activity is recreated or (2) avoiding the activity recreation. We choose the first strategy because it is more flexible as it allows activity recreations instead of preventing the configuration changes altogether. Listing 7.4 presents an example that explains how we retain the `Option` object by using the saved instance after the configuration changes to prevent null reference of the object (S2: Retain object).

```
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
+  setRetainInstance(true);// retain this fragment
}

// new field for saving the object
+ private static Option saveOption;

public View onCreateView(LayoutInflater inflater,
            ViewGroup container,
            Bundle savedInstanceState) {
// saving and loading the object
+  if(option!=null){
+     saveOption = option;
+  }else{
+     option = saveOption;
+  }
  switch (option.getButtonStyle()) { //crashing point
```

Listing 7.4: Example of handling crashes during configuration changes

Commit transactions Each fragment can be modified independently of the host activity by performing a set of changes. Each set of changes that we *commit* (perform requested modifications atomically) to the activity is called a

[4]https://developer.android.com/guide/topics/resources/runtime-changes.html

transaction. Android documentation [5] specifies rules to prohibit committing transactions at certain stages of the lifecycle. Transactions that are committed in disallowed stages will cause the app to throw an exception. For example, invoking `commit()` after `onSaveInstanceState()` will lead to `IllegalStateException` since the transaction could not be recorded during this stage. We employ two strategies for resolving the incorrect commits: (S6: Move stmt) moving `commit()` to a legal callback (e.g., onPostResume()), (S4: Replace method) replacing `commit()` with `commitAllowingStateLoss()`.

Communication between activity and fragment The lifecycle of a fragment is affected by the lifecycle of its host activity [6]. For example, in Figure 7-1, when an activity is created (`onCreate()`), the fragment cannot proceed beyond the `onActivityCreated()` stage. Invoking `getActivity()` in the illegal stage of the lifecycle will return `null`, since the host activity has not been created or the fragment is detached from its host activity. A `NullPointerException` may be thrown in the following execution. We employ two strategies for resolving this problem: (S1: GetActivity-check) inserting condition `if(getActivity()==null)`, and (S6: Move stmt) moving `getActivity()` to another stage (when the host activity is created and the fragment is not detached from the host activity) of the fragment lifecycle.

Retrieve wrong resource id Android *resources* are the additional files and static content used in Android source code (e.g., bitmaps, and layout) [7]. A *resource id* is of the form *R.x.y* where x refers to the type of resource and y represents the name of the resource. The resource id is defined in XML files and it is the parameter of several Android API (e.g., `findViewbyId(id)` and `setText(id)`). Android developers may mistakenly use a non-existing resource id which leads to

[5]https://developer.android.com/reference/android/app/FragmentTransaction.html
[6]https://developer.android.com/guide/components/fragments.html
[7]https://developer.android.com/guide/topics/resources/accessing-resources.html

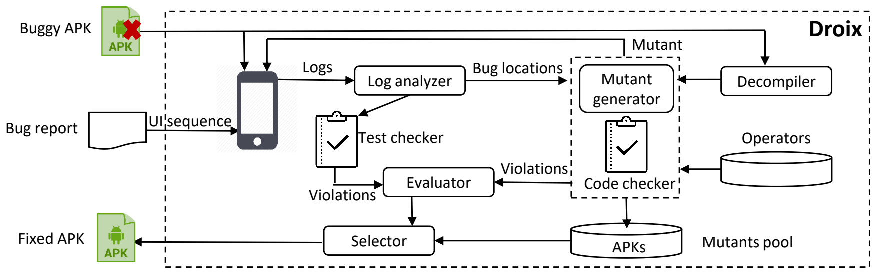


Figure 7-3: Droix's Android Repair Framework

`Resources$NotFound` exception. Listing 7.5 shows a scenario where the developers change the string resource id (S3: Replace resource id).

```
- int msgStrId = R.string.confirmation_remove_alert;
+ int msgStrId = R.string.confirmation_remove_file_alert;
```

Listing 7.5: Example of handling crashes due to wrong resource id

Incorrect type-cast of resource To implement UI interfaces, an Android API [8] (`findViewById(id)`) could be invoked to retrieve widgets (view) in the UI. As each widget is identified by attributes defined in the corresponding XML files, an Android developer may misinterpret the correct type of widget, resulting in crashes due to `ClassCastException`. We repair the crash by replacing the type cast expression with correct type (S5: Replace cast). Listing 7.6 presents an example where the `ImageButton` object is incorrectly type casted.

```
- mDefinition=(TextView)findViewById(R.id.definition);
+ mDefinition=(ImageButton)findViewById(R.id.definition);
```

Listing 7.6: Example of handling crashes due to incorrect type-cast of resource

7.6 Methodology

Figure 7-3 presents the overall workflow of Droix's repair framework. Droix consists of several components: a test re player, a log analyzer, a mutant

[8]https://developer.android.com/reference/android/app/Activity.html

Table 7.3: Code-level and Test-level Properties Enforced in Droix

Level	Type	Description
Code-level	Well-formedness	Verify that a mutated APK is compilable and the structural type of the program matches with the program context of the selected operator.
	Bug hazard	Checks whether a transformation violates Java exception-handling best practices.
	Exception Type	Checks whether a transformation matches a given exception type. (e.g., Insert Null-check should be used for fixing `NullPointerException` exclusively)
Test-level	Lifecycle	Checks that the event transition matches with the activity and fragment lifecycle model (edges in Figure 7-1).
	Activity-Fragment	Checks that the interaction between a fragment and its parent activity matches the activity-fragment coordination model (dashed lines in Figure 7-1)
	Commit	Checks that a commit of a fragment's transactions is performed in the allowed states (i.e., after an activity's state is saved).

generator, a test checker, a code checker, and a selector. Given a buggy APK_P and UI event sequences U extracted from its bug report, Droix produces a patched $APK_{P'}$ that passes U and has the minimum number of properties violations.

Droix fixes a crash using a two-phase approach. In the first phase, Droix generates an instrumented APK_I to log all executed callbacks. With the instrumented APK, Droix replays the UI event sequences U on a device. The log analyzer parses the logs dumped from the execution, extracts program locations $Locs$ from the stack trace, and identifies test-level property Rt_{orig} using the recorded callbacks.

In the second phase, Droix compiles APK_P to the intermediate representation. Based on the intermediate representation, our mutant generator produces a set of candidate apps (stored in the mutant pool) by applying a set of operators at each location l in $Locs$. For each operator op, our code checker records code-level property Rc_{cand} based on the program structure of l and the information in thrown exception. For each candidate APK_C, Droix reinstalls APK_C onto the device and replays U on APK_C. Then, our log analyzer parses the dumped logs that include the execution information of callback methods to extract new buggy locations and information of test-level property Rt_{cand}. Given as input Rt_{cand} for APK_C, the test checker compares Rt_{orig} with Rt_{cand} to check if APK_C introduces any new property violations. Finally, our evaluator analyzes Rt_{cand} and Rc_{cand} to compute the number of property violations and passes the results to the selector, which chooses the best app as the final fixed APK.

We next give detailed descriptions of various components in our repair framework. As the ability to repair using UI event sequences is one of the main features of Droix, we describe how we handle UI event sequences in Subsection 7.6.1. We explain the fault localization step (part of the log analyzer) in Subsection 7.6.2, followed by the code checker and the test checker in Subsection 7.6.3. Subsection 7.6.4 introduces our mutant generation and evaluation algorithm.

7.6.1 Test with UI Sequences

Existing techniques in automated program repair typically rely on unit tests [103] or test scripts [142, 96, 108] to guide repair process. Droix uses user event sequences (e.g., clicks and touches) as input to repair buggy apps, which introduces new challenges: (1) these event sequences are often not included as part of the source code repository and reproducing these event sequences is often time-consuming; (2) ensuring that a recorded sequence has been reliably replayed multiple times is difficult as UI tests tend to be *flaky* (the test execution results may vary for the same configuration).

To reduce manual effort in obtaining UI sequences U, Droix allows users to specify different kinds of event sequences, including: (1) a set of actions (e.g., clicks, touches, swipes) leading to the crash which can be recorded using `monkeyrunner` [9] GUI interfaces, (2) a set of Android Debug Bridge (adb) commands [10], and (3) scripts containing a mixture of recorded actions and adb commands. Non-technical users could record their actions with `monkeyrunner` while Android developers could make use of adb commands to have better

[9]Monkeyrunner contains API that allows controlling Android devices: https://developer.android.com/studio/test/monkeyrunner/index.html

[10]ADB is a command-line tool that are used to control Android devices: https://developer.android.com/studio/command-line/adb.html

control of the devices (e.g., rotate screen).

Droix employs several strategies to ensure that each UI test outcome is consistent across different executions [109]. Specifically, for each UI test of a buggy app, Droix automatically launches the app from the home screen, inserts pauses in between each event sequence, terminates the apps after test execution, and brings the android device back to home screen (ensure that the last state of the device is the same as the initial state of the device). Furthermore, Droix executes each UI test for at least three times where each test execution has pauses of different duration (pauses for 5, 10, 15 seconds) inserted in between events.

7.6.2 Fault Localization

Our fault localization step aims to pinpoint faulty program locations that lead to the crash. Since our approach does not require source code nor heavy test suite, we leverage stack trace information for fault localization. The stack trace contains (1) the type of exceptions being thrown, (2) the specific lines of code where the exception is thrown, and (3) the list of classes and method calls in the runtime stack when the exception occurs. We choose to perform fault localization using stack trace information because (1) this information is often included in the bug report of crashes (which allows us to compare the actual exception thrown with the excepted exception) and (2) previous study has demonstrated the effectiveness of using stack trace to locate Java runtime exceptions [133].

7.6.3 Code Checker and Test Checker

Instead of relying solely on the UI test outcome, Droix enforces two kinds of properties: *code-level properties* (properties that are checked prior to test execution) and *test-level properties* (properties that are verified during/after test execution). These properties are important because (1) they serve as additional test oracles for checking the validity of a candidate app; (2) they could compensate for the lack of passing UI test sequences.

Table 7.3 shows different properties enforced in Droix. *Bug hazard* is a circumstance that increases likelihood of a bug being present in a program [47]. A recent study of Android apps reveals several exception handling bug hazards and Java exception handling best practices [58]. Given an exception E that leads to a crash, our code checker categorizes an exception as either checked exception, or unchecked exception, or error to determine if we could insert a handler for the given exception E. According the Java exception handling best practice "Error represents an unrecoverable condition which should not be handled", hence, our code checker considers inserting handler (try-catch block) for runtime errors a hard constraint and eliminates such patches from our search space. In contrast, inserting handlers for unchecked and checked exceptions are encoded as soft constraints that could affect the score of a mutant. In contrast, our code checker encodes the well-formedness property and the exception type property as hard constraints that are required to be satisfied.

Given a previous lifecycle callback *prev* and a current lifecycle callback *curr*, our test checker verifies if $prev \rightarrow curr$ obeys the activity/fragment lifecycle management rules (Figure 7-1). Droix considers all test-level properties as soft constraints because these properties may not be directly related to the crash (e.g., resource-related crashes).

7.6.4 Mutant Generation and Evaluation

Droix supports eight operators derived from our study of crashes in Android apps (Section 7.4). Readers could refer to Table 7.2 for the details of each operator.

Algorithm 5 presents our patch generation algorithm. Droix leverages evolutionary algorithm where each population is reproduced, evaluated and selected (we adopt similar patch algorithm in PAR [82]). Given as input

```
Input: Buggy $APK_P$, Operators $Op$, Population size $PopSize$, UI test $U$, Program
       Locations $Locs$
Input: Fitness $Fit$: $< Patch, Rc, Rt > \rightarrow \mathbb{Z}$
Result: APK that passes $U$ and contains least property violations
1 $Pop \leftarrow initialPopulation(APK_P, PopSize)$;
2 while $\nexists C \in Pop.C$ passes $U$ do
3     $Mutants \leftarrow Mutate(Pop, Op, Locs)$ ;            // apply $Op$ at $l \in Locs$
      /* select mutant with least $Rc$ and $Rt$ violations              */
4     $Pop \leftarrow Select(Mutants, PopSize, Fit)$;
5 end
```

Algorithm 5: Patch generation algorithm

population size *PopSize*, fitness function *Fit*, and a list of faulty locations *Locs*, our approach iteratively generates new mutants by applying one of the operators listed in Table 7.2 at each buggy location in *Locs*, evaluates each mutant by executing the input UI event sequences U, and computes the number of code-level property *Rc* and test-level property *Rt* violations. The generate-and-validate process terminates when either there exists at least one mutant in the population that passes U or the time limit is exceeded. Our patch generation algorithm differs from existing patch generation systems that use evolutionary algorithm [82, 142] in which we use a different patch representation and fitness function. Specifically, each mutant is an APK in our representation. Instead of using the number of passing tests as the fitness function, our fitness function *Fit* computes the number of code-level and test-level property violations.

7.7 Implementation

Our Android repair framework leverages various open source tools to support different components. Specifically, our log analyzer uses Logcat [11], a command-line tool that generates logs when events occur on an Android device. We implement the eight operators in Table 7.2 on top of the Soot framework (v2.5.0) [86]. Soot is a Java optimization framework that supports analysis and transformation of Java bytecode. Dexpler, a module included in Soot leverages a Dalvik bytecode

[11]https://developer.android.com/studio/command-line/logcat.html

disassembler to produce Jimple (a Soot representation) which enables reading and writing Dalvik bytecode directly [44]. We use the Dexpler module in Soot for our decompiler component in Figure 7-3. To support the “S4: Replace method" operator, we use the Levenshtein distance to select a method with similar method name and compatible parameter types. Our implementation for the “S3: Replace resource id" operator uses Android resource parser in FlowDroid [37] to obtain a resource id of the same type. As each compiled APK needs to be signed before installation, we use jarsigner [12] for signing the compiled APK. We re-install the signed APK onto the device using adb commands [13]. Instead of uninstalling and re-installing each signed app, app re-installation allows us to keep the app data (e.g. account information and settings) to save time in re-entering the required information during subsequent execution of U.

7.8 Subjects

While there are various benchmarks used in evaluating the effectiveness of automated testing of Android applications [53, 98, 30, 31], a recent study [54] showed that the crashes in these benchmarks cannot be adequately reproduced by existing Android testing tools. Meanwhile, Android-specific benchmark like DROIDBENCH [37] does not contain real Android apps and it is designed for evaluating taint-analysis tools. Although empirical studies on Android apps [46, 59] investigated the bug reports of real Android apps, none of these studies try to replicate the reported crashes. Therefore, all existing benchmarks cannot be used for evaluating the effectiveness of analyzing crashes in Android apps.

We introduce a new benchmark, called DroixBench that contains 24 reproducible crashes in 15 real-world Android apps. While we developed

[12]http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html
[13]https://developer.android.com/studio/command-line/adb.html

DroixBench specifically for evaluating Droix, this benchmark could be used to assess the effectiveness of detecting and analyzing crashes in Android apps. To facilitate future research on analysis of crashes, we made DroixBench publicly available at the following anonymous link: https://droix2017.github.io/.

DroixBench is a new set of Android apps for evaluating Droix. Apps used for deriving transformation operators in Section 7.4 are not included in DroixBench so as to avoid the overfitting problem in the evaluation. Specifically, we modified our crawler to find the most recent issues (bug reports) on Android apps crashes on GitHub. Our goal is to identify a set of reproducible crashes in Android apps. To reduce the time in manual inspection of these bug reports, our crawler excludes (1) issues that do not contain bug-fixing commits (which is essential for our study of patch quality); (2) unresolved issues (to avoid invalid failures); and (3) non-Android related issues (e.g., iOS crashes). This step yields more than 300 GitHub issues in which we manually inspect to check for the validity of these bug reports. For each of these bug reports, we further exclude defects that do not fulfill the following criteria:

Device-specific defects. We eliminate defects that require specific versions/brands of Android devices.

Resource-dependent defects. We eliminate defects that require specific resources (e.g., making phone calls) as we may not be able to replicate these issues easily on an Android emulator.

Irreproducible crashes. We eliminate crashes that could not be reproduced by the developers by checking whether the developers mention any difficulty in reproducing the reported crashes.

7.9 Evaluation

We perform evaluation on the effectiveness of Droix in repairing crashes on real Android apps and we compare the quality of Droix's patch with the quality of the human patch. Our evaluation aims to address the following research questions:

RQ1 How many crashes in Android apps can Droix fix?

RQ2 How is the quality of the patches generated by Droix compared with the patches generated by developers?

7.9.1 Experimental Setup

We evaluate Droix on 24 defects from 15 real Android apps in DroixBench. Table 7.4 lists information about the evaluated apps. The "Type" column contains information about the specific type of exception that causes the crash, whereas the "TestEx" column represents the time taken in seconds to execute the UI test. Overall, DroixBench contains a wide variety of apps of various sizes (4-115K lines of code) and different types of exceptions that lead to crashes.

As Droix relies on randomized evolutionary algorithm, we use the same parameters (10 trials for each defect with *PopSize*=40 and a maximum of 10 generations) as used in GenProg [141] for our experiments. Each run of Droix is terminated after one hour or when a patch with minimal violations is generated. All experiments were performed on a machine with a quad-core Intel Core i7-5600U 2.60GHz processor and 12GB of memory. All apps were executed on Google-Nexus-5x-API25 emulator.

For each defect, we manually inspect the source code of human patched program and the source code decompiled from Droix's patched program. If the source code of automatically patched program differs from the human patched program, we further investigate the UI behavior of patched programs by installing both the human generated APK and the automatically generated APK

Table 7.4: Subject Apps and Their Basic Statistics

App Name	Description	Version	LOC	Type	TestEx(s)
Transistor	radio players	1.2.3	4K	NullPointer	42.1
		1.1.5	4K	IllegalState	40.1
Pix-art	photo editor	1.17.1	54K	NullPointer	37.2
		1.17.0	60K	NullPointer	42.0
PoetAssistant	poet writing helper	1.18.2	12K	NullPointer	42.3
		1.10.4	6K	SQLite	60.9
Anymemo	flashcard learning	10.10.1	29K	NullPointer	50.5
		10.9.922	33K	NullPointer	83.9
AnkiDroid	flashcard learning	2.8.1	73K	IllegalState	50.6
		2.7b1	73K	ClassCast	37.2
Fdroid	opensoure app repository	0.103.2	50K	IllegalState	38.7
		0.98	38K	SQLite	37.3
Yalp	app repository	0.17	11K	NullPointer	57.4
LabCoat	GitLab client	2.2.4	45K	NullPointer	49.2
GnuCash	finance expense tracker	2.1.4	42K	IllegalArgument	32.0
		2.1.3	40K	NullPointer	37.2
		2.0.5	37K	IllegalArgument	42.2
NoiseCapture	noise evaluator	0.4.2b	10K	NullPointer	42.5
		0.4.2b	10K	ClassCast	41.2
ConnectBot	secure shell client	1.9.2	26K	OutOfBounds	57.4
K9	email client	5.111	115K	NullPointer	42.2
OpenMF	Mifosx client	1.0.1	75K	IllegalState	134.0
Transdroid	torrents client	2.5.0b1	37K	NullPointer	45.9
Beem	communication tool	0.1.7rc1	21K	NullPointer	61.3

onto the Android device. For each APK, we manually perform visual comparison of the screens triggered by a set of available UI actions (clicks, swipes) after the crashing point.

Definition 10 *Given the source code of human patched program Src_{human}, the source of an automatically generated patch $Src_{machine}$, the compiled APK of human patched program APK_{human}, the compiled APK of automatically generated patch $APK_{machine}$, we measure patch quality using the criteria defined below:*

(C1) Syntactically Equivalent. *$Src_{machine}$ is "Syntactically Equivalent" if both $Src_{machine}$ and Src_{human} are syntactically the same.*

(C2) Semantically Equivalent. *$Src_{machine}$ is "Semantically Equivalent" if both $Src_{machine}$ and Src_{human} are not syntactically the same but produce the same semantic behavior.*

(C3) UI-behavior Equivalent. *$APK_{machine}$ is "UI-behavior Equivalent" to APK_{human}, if the UI-state at the crashing point after applying the automatic fix is same to the UI-state at the crashing point after applying the human patch. Two UI-state are considered to be same if their UI layouts are same, the set of events enabled are same, and these events again (recursively) lead to UI-equivalent states. UI-behavior equivalence of APK_{human} against $APK_{machine}$ is checked manually in our experiments.*

(C4) Incorrect. *We label a $APK_{machine}$ as "Incorrect" when $APK_{machine}$ leads to undesirable behavior (e.g., causes another crash) but this behavior is not observed in APK_{human}.*

(C5) Better. *We label a $APK_{machine}$ as "Better" when APK_{human} leads to regression witnessed by another UI test U_R whereas $APK_{machine}$ passes U_R .*

Formally, $C1 \implies C2 \wedge C2 \implies C3$, hence, a generated patch that is syntactically equivalent to the human patch is superior than both semantically equivalent patch and UI-behavior equivalent patch. We acknowledge that, in general, checking whether a patch is semantically equivalent to the human patch (`C2`) is an undecidable problem. However, in our manual analysis, the correct behavior for all the evaluated patches are well-defined. While `C1` and `C2` investigate the behavior of patches at the source-code level, we introduce `C3: UI-behavior Equivalent.` to compare the behavior of patches at the GUI-level. We consider `C3` because our approach uses GUI tests for guiding the repair process. Furthermore, since our approach does not require source code, direct manual checking of criterion `C2` may be sometimes tedious.

Table 7.5: Patch Quality Results

App	Version	Time (s)	Fix type	Repair	Syntactic Equiv.	Semantic Equiv.	UI-behavior Equiv.	Others
Transistor	1.2.3	616	-					
	1.1.5	987	GetActivity-check	√				better(⊕)
PixArt	1.17.1	1164	-					
	1.17.0	1525	Null-check	√			△	
PoetAssistant	1.18.2	955	Null-check	√			△	
	1.10.4	3600	-					
Anymemo	10.10.1	2104	-					
	10.9.922	1336	Retain Object	√		⊙		
AnkiDroid	2.8.1	3600	-					
	2.7b1	3600	Try-catch	√				text missing(×)
Fdroid	0.103.2	2293	Replace method	√	⋆			
	0.98	518	-					
Yalp	0.17	2970	-					
LabCoat	2.2.4	2074	Null-check	√	⋆			
GnuCash	2.1.3	360	-					
	2.0.5	1492	Try-catch	√			△	
	2.1.4	3600	-					
ConnectBot	1.9.2	572	Try-catch	√				text missing(×)
NoiseCapture	0.4.2b	340	Null-check	√	⋆			
	0.4.2b	520	Replace cast	√	⋆			
K9	5.111	1718	Try-catch	√				crash(×)
OpenMF	1.0.1	3600	GetActivity-check	√	⋆			
Beem	0.1.7rc1	2378	Null-check	√	⋆			
Transdroid	2.5.0b1	1315	Null-check	√	⋆			
	24			15	7	1	3	4

7.9.2 Evaluation Results

Table 7.5 shows the patch quality results for Droix. The “Time" column in Table 7.5 indicates the time taken in seconds for generating the patch before the one hour time limit is reached. On average, Droix takes 30 minutes to generate a patch. Meanwhile, the “Repair" column denotes the number of *plausible patches* (APKs that pass the UI test) generated by Droix. Overall, Droix generates 15 plausible patches (rows marked with √) out of 24 evaluated defects. Our analysis of the 9 defects that are not repaired by Droix reveals that all of these defects are difficult to fix because all the corresponding human patches require at least 10 lines of edits.

The “Fix type” column in Table 7.5 shows the operator used in each patch (Refer to Table 7.2 for the description of each operator). The “Null-check” operator is the most frequently used operators (used in six patches and 4/6=67%

of these patches are syntactically equivalent to the human patches). These results match with the high frequency of "Null-check" operator in our empirical study (Table 7.1). Interestingly, we also observe that the "GetActivity-check" operator tends to produce high quality patches because this operator aims to enforce the "Activity-Fragment" property that checks for the coordination between the host activity and its embedded fragment.

The "Syntactic Equiv." column in Table 7.5 shows the number of patches that fulfill the $C1$ criteria, while the "Semantic Equiv." column denotes the number of patches that fulfill the $C2$ criteria. Similarly, the "UI-behavior Equiv" column demonstrates the number of fixed APKs that fulfill the $C3$ criteria. Particularly, we consider the patch generated by Droix for Anymemo v10.9.922 as semantically equivalent to the human patch because both patches use an object of the same type retained before configuration changes to fix the `NullPointerException` but the used object is retained in different program locations (i.e., not syntactically equivalent). As shown in Table 7.5, there are three patched APKs that are UI-behavior equivalent to the human generated APKs. Interestingly, we observed that although the corresponding human patches for these defects require multi-lines fixes, the bug reports for these UI-behavior equivalent patches indicate that specific conditions (e.g., `mSpinner.getSelectedItemId()!=INVALID_ROW_ID` for the GnuCash v2.0.5 defect) are required to trigger the crashes. As these conditions are complex and may be impractical to trigger from the UI level, synthesizing precise conditions is not required for repairing these crashes to ensure UI-behavior equivalent.

The "Others" column in Table 7.5 includes one patch that is better than the human patch (marked as $\oplus$) and three patches that are incorrect (marked as $\times$). We consider the patch for Transistor v1.1.5 to be better than human patch as it passes regression test included in the bug report whereas the human patch introduces a new regression (See Section 7.3 for detailed explanations). For two of the incorrect patches, we notice that some texts that appear on the screen of human APKs are missing in the screen of fixed APKs (text missing). Meanwhile, the crash in k9 v5.111 occurs due to an invalid email address for a particular contact. In this case, the human APK treats the contact as a non-existing contact while the patched APK displays the contact as unknown recipient and crashes when the unknown recipient is selected. We think that both the human APK and the patched APK could be improved (e.g., prompt the user to enter a valid email address instead of ignoring the existence of the contact). Although the patch generated by Droix for k9 violates the bug hazard property (catching a runtime exception), we select this patch as no other patches exist in the search space.

7.10 Threats to Validity

We identify the following threats to the validity of our experiments: **Operators used.** While we derive our operators from frequently used operators in fixing open source apps and from Android API documentation, we note that our set of operators is not exhaustive.

Reproducing crashes. We manually reproduce each crash in our proposed benchmark. As we rely on Android emulator for reproducing these crash, the crashes in our benchmark are limited to crashes that could be reliably reproduced on Android emulators. Crashes that require specific setup (e.g., making phone calls) may be more challenging or impractical to replay.

Crashes investigated. As we only investigate open source Android apps in our empirical study and in our proposed benchmark, our results may not generalize to closed-source apps. We focus on open source apps because our patch analysis requires the availability of source codes. Nevertheless, as Droix takes as input Android APK, it could be used for fixing closed source apps. We leave the empirical evaluation of closed source apps as our future work.

Patch Quality. During our manual patch analysis, at least two of the authors analyze the quality of human patches versus the quality of automatically generated patches separately and meet to resolve any disagreement. As most bug reports include detailed explanations of human patches and the expected behavior of the crashing UI test, the patch analysis is relatively straightforward.

7.11 Related work

Testing and Analysis of Android Apps. Many automated techniques (AndroidRipper [30], ACTEVE [31], A^3E [40], Collider [75], Dynodroid [98], FSMdroid [136], Fuzzdroid [128], Orbit [145], Sapienz [100], Swifthand [53], and work by Mirzaei et al. [110]) are proposed to generate test inputs for Android apps. Our approach is orthogonal to these approaches and the tests generated by these approaches could serve as inputs to our Android repair system. Several approaches focus on reproducing crashes in Java projects [52, 144, 135]. Meanwhile, CRASHSCOPE [113] automatically detects and reproduces crashes in Android apps. Our benchmark with 24 reproducible crashes could be used for evaluating the effectiveness of these approaches. Similar to Flowdroid [37], we implement our fix operators on top of the Soot framework, and we use activity lifecycle information for our analysis of Android apps. Instead of considering only the activity lifecycle as in Flowdroid, we also encode fragment lifecycle and activity-fragment coordination as test-level properties. RERAN [71] could precisely record and replay UI events on Android devices, including gestures (e.g., multitouch). While our approach allows UI sequences in forms of scripts recorded in the user interface, the record-and-replay mechanism in RERAN could allow Droix to handle more complex UI events. Although our code checker incorporates some Java exception handling best practices listed in recent study of Android apps [60], our empirical study of crashes that occur in Android apps goes beyond prior study by performing a thorough investigation of the common root causes of Android crashes.

Automated Program Repair. Several techniques (Angelix [108], ASTOR [105], ClearView [124], Directfix [107], GenProg [142], PAR [82], Prophet [96], NOPOL [143], *relifix* [138]) have been introduced to automatically generate patches. There are several key differences of our Android repair framework compared to other existing repair approaches. Firstly, instead of relying on the quality of the test suite for guiding the repair process, our approach augments a given UI test with code-level and test-level properties for ranking generated patches. Secondly, existing approaches could not handle flaky UI tests as they may misinterpret the test outcome of UI tests and may mistakenly produce invalid patches. Finally, our repair framework modifies compiled APK and each test execution is performed remotely on Android emulators, whereas other approaches modify source code directly where each test is being executed on the same platform as other components of the repair system. Other studies for automated repair use benchmark for C programs [89], whereas Droixbench contains a set of reproducible crashes for Android apps. QACrashFix [68] and work by Azim et al. [39] use Android apps as dataset for experiments, without any Android-specific study of cause for crashes. Their repair operators are Android-agnostic. Specifically, QACrashFix merely add/delete/replace single node in the Abstract Syntax Tree, wheareas work by Azim et al only inserts fault-avoiding code that is similar to workaround identified in our study in Section 7.4. To eliminate invalid patches, anti-patterns are proposed as a set of forbidden rules that can be enforced on top of search-based repair approaches [139]. Although our code-level and test-level properties could be considered as different forms of anti-patterns that are examined prior to and after test executions, we use these properties for selecting mutants that violate fewer properties instead of eliminating these mutants. Similar to Droix that uses stack trace information for fault localization, the work of Sinha et al. uses stack trace information for locating Java exceptions [133]. However, their approach only supports analysis of `NullPointerException`, whereas our approach could automatically repair different types of exceptions.

Other Repairs of Android Apps EnergyPatch fixes energy bugs in Android apps using a repair expression that captures the resource expression and releases system calls [41]. The battery-aware transformations proposed in [55] aims to reduce power consumption of mobile devices. Several approaches generate security patches for Android apps [150, 114]. While energy bugs and security-related vulnerabilities may cause crashes in Android apps, we present a generic framework for automated repair of Android crashes, focusing on crashes that occur due to the misunderstanding of Android activity and fragment lifecycles.

UI Repair. FlowFixer is an approach that repairs broken workflow in GUI applications that evolve due to GUI refactoring. SITAR uses annotated event-flow graph for fixing unusable GUI test scripts [69]. Although Droix takes as input UI test, it automatically fixes buggy Android apps rather than the inputs that crash the GUI applications.

7.12 Chapter Summary

In this chapter, we study the common causes of 107 crashes in Android apps. Our investigation reveals that app crashes occur due to missing callback handler (17.76%), improper handling of resources (16%), and violations of management rules for the Android activity and fragment lifecycle (14%). Based on our analysis of patches issued by Android developers to fix these crashes and the Android API documentations that specify the correct usage of Android API, we derive a set of lifecycle-aware transformations. To reduce time and effort in fixing crashes in Android apps, we also introduce Droix, a novel Android repair framework that automatically generates a fixed APK when given as input a buggy APK and UI event sequences. To encourage future research of Android crashes, we propose DroixBench, a benchmark that contains 24 reproducible crashes occurring in 15 open source Android apps. Our evaluation on DroixBench demonstrates that Droix could generate repair for 63% of the evaluated crashes and seven of the automatically generated patches are syntactically equivalent to the human patches.

Although our repair framework currently performs analysis and mutation of Android apps on desktop machine while executing UI tests on an Android emulator, in future, it is feasible to have a standalone repair system that could be installed as an app that automatically fixes crashes occurring in other apps on Android devices. Since our GUI interface based on monkeyrunner does not assume any programming knowledge, our repair framework could potentially benefit general non-technical users who would like to have their own versions of fixed apps instead of waiting for the official releases.

As we observe that many crashes occur due to the misunderstanding of activity/fragment lifecycle that are specified in the Android API documentations. we think that Droix could be used as a plugin that automatically provides management rule violations together with patch suggestions to assist developers in understanding the Android API specifications.

Chapter 8

Conclusion

In this thesis, we first studied the effectiveness of current repair operators in existing program repair tools and suggested several selection strategies for choosing a reduced set of repair operators among all operators used by repair tools. Moreover, we investigate the relationship between the quality of automatically generated patches and the quality of test suite to improve the effectiveness of program repair tools using test cases. To further reduce the number of invalid patches generated by each repair operator, we proposed a set of *anti-patterns* that eliminate patches fulfilling certain transformation rules. Furthermore, we also introduced a program repair tools with repair operators that are designed to solve the problems of software regressions. Lastly, we designed specialized repair operators and proposed a novel repair framework to fix crashes in Android apps.

Overall, this thesis attempts to solve several important challenges in automated program repair. Our study of the effectiveness of repair operators in existing repair tools and our proposed benchmark in Chapter 3 provides direct insights to the current design of repair operators and aims to drive the future design of repair operators. Our correlation study on automated program repair and test suite metrics suggests the possibility of enhancing quality of automated generated patches by improving the quality of test suite. On the other hand, our proposed anti-patterns in Chapter 5 not only provide deeper understanding of existing problems in automatically generated patches, but also allow more flexible rules to be enforced for each program transformation of the corresponding repair operator. Furthermore, in Chapter 6, we present a concrete example of a repair technique that is designed for fixing software regressions. In our proposed repair tool, *relifix*, we also demonstrate how to utilize syntactic changes between different program versions in the design of repair operators.

8.0.1 Research Outputs

To encourage the usage of the Codeflaws benchmark for evaluating program repair approaches, the benchmark is publicly available at https://codeflaws.github.io/.

As the work on anti-patterns are part of a software patent[1] with our collaborators from Fujitsu Laboratories of America, the source code of our modified versions of SPR and GenProg could not be made publicly available but other information (including the study in which we derive our anti-patterns and the patches generated by all evaluated approaches) is publicly available at https://anti-patterns.github.io/search-based-repair/.

We think that DroixBench that contains a set of reproducible crashes could be useful for evaluating the effectiveness of Android testing approaches. Hence, we have made DroixBench publicly available at https://droix2017.github.io/. The link contains the bug report for each defect in DroixBench together with the corresponding bug-fixing commit. However, the source code for Droix will not be made publicly available due to the copyright restrictions with our collaboration with Singapore Telecommunications Ltd.

The source code for *relifix* is publicly available at: https://github.com/stan6/relifix

[1]Software program repair: https://www.google.com/patents/US20170060735

8.0.2 Future Work

Apart from automated program repair, we believe that the ideas proposed in this thesis could be applied to other domains. In particular, the benchmark extracted from submissions of participants in online competitive programming platform is a first step towards the applications of program repair techniques in automated grading and intelligent tutoring. Moreover, as the usage of anti-patterns could improve on fix localization, this suggests the possibility of using automated program repair techniques for improvement of fault localization techniques. In a broader view, the perspective of reconciling problematic changes introduced in *relifix* is inspired by the idea of merging several conflicting program changes. With the rapid development of internet of things, the adaptation of automated program techniques to other platforms (e.g., mobile devices as discussed in Chapter 7) encourages practical usage of bug-fixing techniques.

Most evaluations proposed in this thesis are performed on mature software subjects (e.g., Vim, PHP, and the F-droid app), except for the Codeflaws benchmark with small programs from programming competition platform. Although our selection of software subjects may appear to be biased towards mature software, we select these subjects because they are well-known real-world software subjects, and they are part from existing benchmarks (e.g., ManyBugs [89] and CoREBench [48]). Meanwhile, in our proposed benchmark, DroixBench, we eliminate this selection bias by including newer Android apps with frequent changes (e.g., NoiseCapture and OpenMF). Instead of selecting the most popular Android apps, we only require that all crashes in DroixBench should be reproducible. In general, the question "Is automated repair best suited to mature software?" has to be investigated through detailed experiments. However, the fact that Droix could repair crashes in newer Android apps as well as more mature Android apps (e.g., F-droid and K9) show that the maturity of the software subjects may not be directly related to the suitability of these subjects for automated program repair.

Although we propose an anti-pattern that prohibits the deletion of all statements that serve as test proxies (e.g., assertions), all the approaches proposed in this thesis treat the test oracles embedded within a test case as black-box. To guide the repair system in generating higher quality patches, it is worthwhile to investigate the possibility of leveraging assertion. An assertion is a boolean expression that is expected to be always true at the indicated point during execution. The predicate captured by assertions could be used as repair constraints in a semantic-based repair approach or augmenting formal specification in a contract-based repair approach.

One of the major problems in automated program repair discussed in this thesis is the lack of specification. Although most program repair approaches only modify source code, the concept of program repair may be useful in the refinement of specification. For example, when a program repair technique generates an incomplete fix due to the input-output constraint represented by a single test, we could automatically generate a list of possible refinement to the constraint and give the generated list to the developer to choose a more general constraint that capture all other test inputs outside the given test suite.

In conclusion, this thesis not only attempts to solve several important challenges in automated program repair (i.e., investigating the effectiveness of repair operators, enhancing quality of automatically generated patches by improving test suite, and supplementing repair operators with anti-patterns), but also opens many opportunities for future research works beyond program repair, including potential applications in intelligent tutoring and ensuring the reliability of mobile applications.

Bibliography

[1] Cpython 5b0fda regression bug report. http://bugs.python.org/issue21233.

[2] Cpython 5b0fda source code. http://hg.python.org/cpython/rev/5b0f-da8f5718.

[3] Cpython b878df regression bug report. http://bugs.python.org/issue16012.

[4] Cpython b878df source code. http://hg.python.org/cpython/rev/b878d f1d23b1/.

[5] Indent version 2.2.10 regression bug report. http://savannah.gnu.org/bugs/?27036.

[6] Indent version 2.2.10 source code. http://ftp.gnu.org/gnu/indent/indent-2.2.10.tar.gz.

[7] Perl bb9ee9 regression bug report. http://perl5.git.perl.org/perl.git/commit/ bb9ee97444732c84b33c2f2432-aa28e52e4651dc.

[8] Perl bb9ee9 source code. http://perl5.git.perl.org/perl.git/commit/ bb9ee97444732c84b33c2f2432aa28e52e4651dc.

[9] Perl dca606 regression bug report. https://rt.perl.org/Public/Bug/ Display.html?id=74290.

[10] Perl dca606 source code. http://perl5.git.perl.org/perl.git/commitdiff/ dca6062a863d0.

[11] Tar version 1.14 regression bug report. http://lists.gnu.org/archive/html/bug-tar/2004-10/msg00034.html.

[12] Tar version 1.14 source code. http://ftp.gnu.org/gnu/tar/tar-1.14.tar.gz.

[13] Vim commit 220906 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_dev/ DbW2gnNqj04/6KaQn2jsDvAJ.

[14] Vim commit 220906 source code. http://code.google.com/p/vim/source/ detail?r=2209060c340d.

[15] Vim version 7.2.50 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_dev/9hG4HrhQuhk/ogbOOYwfPPkJ.

[16] Vim version 7.2.50 source code. http://code.google.com/p/vim/source/-detail?name=v7-3-202.

[17] Vim version 7.3.202 regression bug report. http://code.google.com/p/vim/issues/detail?id=9.

[18] Vim version 7.3.202 source code. http://code.google.com/p/vim/source/detail?name=v7-3-202.

[19] Vim version 7.3.251 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_dev/TUbaixgUilQ/YV38sAkof10J.

[20] Vim version 7.3.251 source code. http://code.google.com/p/vim/source/detail?name=v7-3-251.

[21] What consumers really need and want. `https://goo.gl/puYdkG`, 2017. Accessed 2017-03-27.

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

[23] 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.

[24] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 39–46. IEEE, 2006.

[25] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 22. ACM, 2010.

[26] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J Martin, Aditya Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[27] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 225–236, New York, NY, USA, 2014. ACM.

[28] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 520–523, Nov 2011.

[29] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261. IEEE, 2011.

[30] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[31] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[32] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[33] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.

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

[35] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. Breaking and fixing the android launching flow. *Computers & Security*, 39:104–115, 2013.

[36] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, 2010.

[37] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[38] F. Y. Assiri and J. M. Bieman. An assessment of the quality of automated program operator repair. In *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ICSE '14, pages 273–282, 2014.

[39] Md Tanzirul Azim, Iulian Neamtiu, and Lisa M Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

[40] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[41] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[42] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

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

[44] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[45] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 82–91, 2006.

[46] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. An empirical analysis of bug reports and bug fixing in open source

android apps. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 133–143. IEEE, 2013.

[47] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[48] Marcel Böhme and Abhik Roychoudhury. CoREBench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 105–115, New York, NY, USA, 2014. ACM.

[49] Cari Borrás. Overexposure of radiation therapy patients in panama: problem recognition and follow-up measures. *Revista Panamericana de Salud Pública*, 20(2-3):173–187, 2006.

[50] William H Brown, Raphael C Malveau, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[51] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI' 08, pages 209–224, 2008.

[52] N. Chen and S. Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, Feb 2015.

[53] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

[54] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[55] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. Battery-aware transformations in mobile applications. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 702–707. IEEE, 2016.

[56] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[57] William S Cleveland and Clive Loader. Smoothing by local regression: Principles and methods. In *Statistical theory and computational aspects of smoothing*, pages 10–49. Springer, 1996.

[58] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. Exception handling bug hazards in android. *Empirical Software Engineering*, 22(3):1264–1304, Jun 2017.

[59] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in android based on github and google code issues. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 134–145. IEEE, 2015.

[60] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. Exception handling bug hazards in android. *Empirical Software Engineering*, pages 1–41, 2016.

[61] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 550–554, Washington, DC, USA, 2009. IEEE Computer Society.

[62] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:65–74, 2010.

[63] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[64] RA DeMillo, Richard J Lipton, and FG Sayward. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing*, 2:107–126, 1979.

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

[66] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[67] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

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

[69] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. Sitar: Gui test script repair. *Ieee transactions on software engineering*, 42(2):170–186, 2016.

[70] Laurence Goasduff and Christy Pettey. Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. *Visited April*, 2012.

[71] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA, 2013. IEEE Press.

[72] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *ACM SIGPLAN Notices*, volume 36, pages 312–326. ACM, 2001.

[73] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, 10(3):171–194, 2000.

[74] D. Jeffrey, Min Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 70–79, May 2009.

[75] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[76] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software*, 2016.

[77] 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.

[78] 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.

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

[80] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 295–306, 2015.

[81] M. G. Kendall. The treatment of ties in ranking problems. *Biometrika*, 33(3):239–251, 1945.

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

[83] X. Kong, L. Zhang, W. E. Wong, and B. Li. Experience report: How do techniques, programs, and tests impact automated program repair? In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering*, ISSRE '15, pages 194–204, 2015.

[84] Philip Koopman. Elements of the self-healing system problem space. *Proceedings of the ICSE WAD03*, 2003.

[85] J. R. Koza. *Genetic Programming: On the Programming of computers by Means of Natural Selection.* MIT Press, 1992.

[86] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

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

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

[89] 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.

[90] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[91] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 959–966. ACM, 2012.

[92] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4), 2007.

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

[94] 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. ACM.

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

[96] 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*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[97] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.

[98] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[99] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 113–116. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[100] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[101] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[102] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 388–391. IEEE, 2013.

[103] 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*, pages 1–29, 2016.

[104] Matias Martinez and Martin Monperrus. Mining repair actions for guiding automated program fixing. Technical report, INRIA, Tech. Rep, 2012.

[105] Matias Martinez and Martin Monperrus. Astor: A program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 441–444, New York, NY, USA, 2016. ACM.

[106] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. *arXiv preprint arXiv:1403.6322*, 2014.

[107] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458, May 2015.

[108] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[109] Atif M. Memon and Myra B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.

[110] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.

[111] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[112] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[113] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.

[114] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[115] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the 8th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 57–68, 2009.

[116] Shimul Kumar Nath, Robert Merkel, Man Fai Lau, and Tanay Kanti Paul. Towards a better understanding of testing if conditionals. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 772–777. IEEE, 2012.

[117] 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*, pages 772–781. IEEE Press, 2013.

[118] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[119] Jie Pan and Loudon Tech Center. Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software*, 1995.

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

[121] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

[122] Karl Pearson. *On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to have Arisen from Random Sampling*, pages 11–28. Springer New York, New York, NY, 1992.

[123] Yu Pei, Yi Wei, Carlo A Furia, and Martin Nordio Bertr. Evidence-based automated program fixing. *CoRR*, 2011.

[124] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, 2009.

[125] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):19, 2012.

[126] 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*, ICSE 2014, New York, NY, USA, 2014.

[127] 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.

[128] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 300–311, Piscataway, NJ, USA, 2017. IEEE Press.

[129] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[130] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[131] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[132] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[133] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164. ACM, 2009.

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

[135] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

[136] Ting Su. Fsmdroid: Guided gui testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 689–691, New York, NY, USA, 2016. ACM.

[137] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2006.

[138] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, pages 471–482. ACM, 2015.

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

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

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

[142] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[143] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[144] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 910–913. ACM, 2015.

[145] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[146] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 919–930, 2014.

[147] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *J. Syst. Softw.*, 85(10):2305–2317, October 2012.

[148] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, October 1999.

[149] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[150] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[151] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2), 2009.