Advanced Techniques for Search-Based Program Repair

CHRISTOPHER STEVEN TIMPERLEY

Ph.D.

University of York,
Computer Science

June 2017

Abstract

Debugging and repairing software defects costs the global economy hundreds of billions of dollars annually, and accounts for as much as 50% of programmers' time. To tackle the burgeoning expense of repair, researchers have proposed the use of novel techniques to automatically localise and repair such defects. Collectively, these techniques are referred to as *automated program repair*.

Despite promising, early results, recent studies have demonstrated that existing automated program repair techniques are considerably less effective than previously believed. Current approaches are limited either in terms of the number and kinds of bugs they can fix, the size of patches they can produce, or the programs to which they can be applied. To become economically viable, automated program repair needs to overcome all of these limitations.

Search-based repair is the only approach to program repair which may be applied to any bug or program, without assuming the existence of formal specifications. Despite its generality, current search-based techniques are restricted; they are either efficient, or capable of fixing multiple-line bugs—no existing technique is both. Furthermore, most techniques rely on the assumption that the material necessary to craft a repair already exists within the faulty program. By using existing code to craft repairs, the size of the search space is vastly reduced, compared to generating code from scratch. However, recent results, which show that almost all repairs generated by a number of search-based techniques can be explained as deletion, lead us to question whether this assumption is valid.

In this thesis, we identify the challenges facing search-based program repair, and demonstrate ways of tackling them. We explore if and how the knowledge of candidate patch evaluations can be used to locate the source of bugs. We use software repository mining techniques to discover the form of a better repair model capable of addressing a greater number of bugs. We conduct a theoretical and empirical analysis of existing search algorithms for repair, before demonstrating a more effective alternative, inspired by greedy algorithms. To ensure reproducibility, we propose and use a methodology for conducting high-quality automated program research. Finally, we assess our progress towards solving the challenges of search-based program repair, and reflect on the future of the field.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Susan Stepney, for her continual support and guidance over the last five years, and for somehow always managing to give me feedback at a superhuman pace. I would also like to extend my thanks to the (former) members of EvoEvo research group for many stimulating discussions: Dr. Simon Hickinbotham, Dr. Tim Hoverd, Dr. Paul Andrews, and Dr. Tim Taylor. A special thanks goes to Dr. Dan Franks for his invaluable advice and support, and whose teaching inspired me to begin this pursuit.

I am indebted to Prof. Claire Le Goues for her mentorship and advice, and for bringing me into her research group. Our paths may never have crossed were it not for the generous support of the William Gibbs Foundation, which made my visit to Prof. Le Goues's lab possible.

Thank you to Mum, Dad, Joe, Becky and Rick for always believing in me and for never letting me forget where I'm from, and to Millie for her cuddly companionship.

And finally to Geraldine for living through every moment of this adventure with me, and giving me the courage to see its end.

Nothing I can write here can begin to express my gratitude to you all. Thank you.

Declaration

I declare that this thesis is a presentation of original work, and that I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references.

The study of mutation analysis for automated program repair, presented in Chapter 4, has been accepted for publication and is due to appear in the proceedings of SSBSE'17:

> Christopher Steven Timperley, Susan Stepney, Claire Le Goues. An Investigation into the Use of Mutation Analysis for Automated Program Repair. SSBSE'17.

CHAPTER 1

Introduction

According to a 2013 study [Judge Business School, Cambridge University, 2013], the worldwide cost of debugging and repairing software bugs is estimated to be $312 billion per year; on average, programmers spend roughly 50% of their time finding and fixing bugs. Automated Program Repair (APR) is a new and emerging technique that attempts to reduce this burden by automatically localising and fixing arbitrary bugs.

In 2009, the research area of APR was born when GenProg [Weimer et al., 2009] demonstrated the ability to fix arbitrary bugs in real-world C programs, using only their provided test suites as an oracle. Although previous work had examined the possibility of automatically detecting, and in some cases, fixing bugs, none of those works considered the general case—each addressed a particular sub-set of bugs, under restrictive assumptions [Demsky and Rinard, 2003; Perkins et al., 2009].

To automatically repair programs, GenProg uses a form of genetic programming (GP). Whereas genetic programming is typically used to evolve entire programs from scratch—albeit small ones—GenProg evolves patches, represented as a sequence of edit operations, instead. These operations are represented by one of three statement-level program transformations: deletion, insertion, and replacement.[1] To constrain the search space, and to transform program repair into a tractable problem, these operations are crafted using existing code, supplied by the program under repair, via a technique later referred to as *plastic surgery* [Barr et al., 2014]. Candidate patches are generated using this representation through a continual process of mutation, crossover, and selection. Patches that pass a greater number of tests, and especially those that pass previously failing tests, are identified as partial solutions by the search, and used as the basis of the next generation of patches. This process of *generation* and *validation* continues until a patch that passes the entire test suite is found.

At ICSE 2012 [Le Goues et al., 2012a], GenProg was shown to repair 55 out of 105 real-world bugs in large-scale C programs, including the PHP and Python interpreters, and libtiff. In the same year, at GECCO 2012, Le Goues et al. [2012c] demonstrated how modifications to GenProg's parameters could allow it to repair another five bugs within the same dataset, and to reduce the wall-clock time required to do so. Automated program repair—an idea dreamt about for more than half a century—appeared to have been conquered, overnight.

[1] Originally, GenProg implemented a swap operation, rather than replacement. This operation was dropped from future versions when its authors found replacement to be a more effective alternative [Le Goues et al., 2012c].

Two years later, at ICSE 2014, Qi et al. [2014] showed that a form of random search—restricted to a single-edit search space—outperformed the genetic algorithm used by GenProg, as measured by the average Number of Test Case Evaluations required to find a repair (NTCE). Whilst these results raised questions regarding the effectiveness of using genetic algorithms for program repair, they could also be explained by the aggressive optimisations made by the random search, and by its restriction to a single-edit search space.

Finally, at ISSTA 2015, Qi et al. [2015] examined the patches generated by GenProg for the ICSE 2012 dataset, and found that 104 of its 110 plausible patches could be explained by a single functionality-deleting modification. (e.g., deletion of an If statement.) For the majority of bug scenarios within the ICSE 2012 dataset,[2] the test harness only checked the exit status of the program was zero. As a result, many of the repairs reported by GenProg *overfit* to the test suite by simply ensuring the program produces the correct exit status, instead of fixing the underlying bug: Most patches either deleted parts of the program, or inserted premature `exit(0)` or **`return`** statements. After addressing issues in the test harnesses, and extending the original test suites, Qi et al. [2015] found that GenProg produced correct fixes for 2 out of 105 bugs; in both cases, these bugs could be patched through functionality deletion alone (i.e., removing a set of statements from the program).

Motivated by these problems, at ESEC-FSE 2015, Long and Rinard [2015] introduced an alternative search-based repair technique: Staged Program Repair (SPR). To avoid overfitting[3] due to functionality deletion, and to address a greater number of bugs, SPR uses a richer set of repair actions (i.e., types of transformation that can be applied to the program), without an explicit deletion action. Although SPR does not permit explicit functionality deletion, Mechtaev et al. [2016] demonstrated that SPR still produces such patches implicitly. In addition to introducing a new repair model, SPR uses a novel search technique to reduce the cost of finding repairs: *value search*. Value search allows SPR to determine *whether* a particular kind of transformation (e.g., modification of an **`if`** condition) can be used to fix the program; only once the feasibility of a repair has been determined, does SPR attempt to find the *values* required to concretely fix the bug (e.g., a fixed **`if`** condition).

Although SPR was shown to outperform GenProg in terms of the number of correct repairs found within a 12-hour window, it is unclear how much of this is due to SPR's larger repair model, and to what extent value search improves efficiency. Unfortunately, value search has not been compared to any alternative search techniques—using the same repair model—such as random search. Whilst SPR may offer a more efficient way of generating single-edit patches, it is difficult to see how its value search algorithm could be extended to cover multiple edits, as Long and Rinard [2015] note.

[2] The ICSE 2012 dataset would go on to form a sub-set of the bug scenarios within the ManyBugs [Le Goues et al., 2015] dataset.

[3] In the context of automated program repair, the term "overfitting" is used to refer to candidate repairs which pass the test suite but do not consitute correct repairs. (i.e., one could produce a test that rejects the candidate repair.)

Since these discoveries, automated program repair techniques have been categorised by three distinct approaches: *search-based* repair techniques, such as GenProg and SPR, which generate candidate repairs according to a search algorithm, and evaluate them until a patch is found which passes the entire test suite; *semantics-based* techniques, such as Angelix [Mechtaev et al., 2016] and SearchRepair [Ke et al., 2015], which use symbolic evaluation to guide patch synthesis; and *specification-based* approaches, such as AutoFix-E [Wei et al., 2010], which use formal specifications to synthesise fixes, rather than using test suites as their oracle.

Whilst semantics-based and specification-based approaches have demonstrated promising results, especially in terms of patch quality, their real-world use is highly restricted. Specification-based approaches rely on the existence of formal specifications, which are rarely found in practice [Kossak et al., 2014]. Although semantics-based approaches do not require the existence of formal specifications, instead relying on the existence of a test suite, the limits of symbolic execution prevent their application to real-world programs, in the general case. Existing semantics-based techniques are unable to deal with code that involves side-effects (e.g., reading from or writing to a file), non-primitive data types (e.g., structs), or looping and recursion.

Search-based program repair is the only approach capable of fixing arbitrary bugs, without restrictions on the program, or the need for formal specifications. To address a greater number of bugs, search-based techniques will need to expand their repair models (i.e., the operations used to construct patches), improve their ability to localise faults, and scale to bugs requiring multiple edits. In this thesis, we investigate the challenges facing search-based program repair, propose and demonstrate effective ways to tackle them, and increase our understanding of the problem along the way.

1.1. Motivation

In this section, we outline the motivation behind this thesis, and provide justification for our focus on search-based program repair of C programs.

Automated Repair of C Programs

Amongst other languages, automated program repair has been successfully applied to programs written in C [Le Goues et al., 2012a; Long and Rinard, 2015, 2016; Mechtaev et al., 2016; Tan and Roychoudhury, 2015], Java [Kim et al., 2013; Le et al., 2016; Xuan et al., 2017], and Python [Ackling et al., 2011]. Accompanying each of these languages is a unique set of opportunities and challenges:

- Being a dynamic language without the benefits of static type checking, bugs

in Python may involve the use of unreferenced variables, passing (incorrect) arguments of the wrong type, or unintended implicit conversions. At the same time, being an interpreted language, with the capacity for introspection and self-modification, Python offers new avenues for (automated) debugging and patch generation.

- Java, on the other hand, employs static type checking, allowing it to avoid this class of run-time bugs. Additionally, unlike C, built-in memory management (provided by the garbage collector) helps to prevent the majority of memory leaks and segmentation faults.[4]

- In contrast to Java and Python, C provides no (built-in) memory management and thus it suffers from its own class of memory leaks, buffer overflows, segmentation faults and pointer misuses. Furthermore, unlike Java and Python, which (by default) provide the programmer with detailed information about a program failure (in the form of a stack trace and exception messages), C programs tend to almost devoid of useful information in such cases. Making automated repair more difficult still, C also lacks a standard framework for testing. Projects tend to either provide an arbitrary set of scripts (usually written in bash or Perl), or in rarer cases, to use their own bespoke testing framework (PHP).

C, therefore, represents a challenging testing ground for automated program repair, where knowledge of the bug is limited to its bare minimum. By demonstrating the effectiveness of automated program techniques in such a barren environment, one gains an increased confidence in their applicability to a wider range of languages. Furthermore, C remains one of the most popular programming languages in the world (ranked at #2 in the January 2017 TIOBE popularity index[5], representing 9.35% of the market), and is widely used within production legacy code, where specifications and test suites tend to be woefully lacking, and the original developers are no longer around.

Most importantly, by using C as the focus of this thesis, we are able to build upon the large body of existing techniques that target this language.

Search-Based Program Repair

Having presented our case for using the C programming language as the focus of this thesis, below we put forward three reasons for pursuing search-based program repair, rather than semantics- or specification-based approaches.

1. **Generality of Fixes:** A number of techniques exist that are theoretically capable of identifying and correcting bugs belonging to a particular class, such

[4] Note, garbage collection does not necessarily ensure Java programs are invulnerable to memory leaks (or, in this case, heap overflows). A common way of introducing such bugs stems from careless storage of objects when using the singleton object pool pattern [Bloch, 2008].

[5] https://www.tiobe.com/tiobe-index/

as memory leaks and integer overflows [Coker and Hafiz, 2013; Gao et al., 2015]. In practice, however, these techniques are only effective under limited circumstances, and unlike search-based repair, they have no utility beyond the specific bug class for which they were designed.

Given its ability to perform arbitrary modifications to the source code, the search-based approach is (in theory) capable of solving any kind of bug, including those that cannot be cleanly delineated into specific bug classes. In contrast, many semantics-based techniques are limited to performing very specific code transformations, such as the replacement of an if-condition, or the RHS of an assignment expression.

Additionally, search-based approaches can be used to produce fixes spanning multiple lines and files, unlike the majority of alternatives, which tend to be limited to modification of a single statement (or expression), or the introduction of a contiguous block of statements (in the case of SearchRepair). Of the non-search-based approaches to automated repair, Angelix is the only technique capable of producing fixes spanning more than a single statement. Unlike search-based approaches, such as GenProg, Angelix suffers a number of limitations with respect to the types of programs to which it can be applied, discussed below.

2. **Generality of Subjects:** In addition to being free from limitations with respect to the types of fixes that search-based repair can address (since the technique involves modification of the source code), search-based repair is free from any limitations on the types of programs on which it may be used. In contrast, a number of semantics-based techniques, including NOPOL, Angelix, relifix and SearchRepair are unable to handle programs involving loops and recursion, or code involving I/O operations (e.g., database and file accesses). Furthermore, techniques based on symbolic evaluation are only able to handle code containing a limited set of primitive types (excluding floating-point numbers and structs).

3. **Minimal requirements:** Unlike specification-based approaches, search-based techniques do not assume the existence of any kind of contracts or specifications, which are seldom found in real-world code. Ideally, all software would be provided with at least a partial specification of its behaviour, but in reality, the uptake of formal methods within the software engineering has been highly limited. Assuming the existence of such artefacts highly restricts the domain of problems to which automated repair may be applied. By forgoing this need, the search-based approach becomes particularly well suited to legacy systems, where knowledge is particularly limited.

   Similarly, most search-based techniques do not require the existence of an oracle, unlike techniques such as relifix [Tan and Roychoudhury, 2015], which uses previous (functioning) versions of the program to fix regressions.

   The only requirements for repair are a test suite capable of exposing the bug

(i.e., one with at least a single failing test case, covering the faulty code) and the source code of the program.

In summary, search-based automated program repair poses the fewest limitations and makes the least assumptions of the bug, allowing it to be applied in the widest set of scenarios.

1.2. Challenges

Despite the potential of search-based repair, and promising early results produced by GenProg, recent studies have highlighted a number of major challenges facing the approach:

1. **Patch Quality:** Of the challenges facing search-based repair, and the wider field in general, the problem of plausible, but incorrect patches (i.e., those that pass the test suite as a result of overfitting) is perhaps the most controversial. Recent studies [Qi et al., 2015; Smith et al., 2015] have demonstrated that the majority of fixes reported for GenProg, the most popular search-based approach, were the result of overfitting to weak test suites (i.e., one which fails to provide sufficient coverage or to thoroughly check the output of the program). On further inspection, Qi et al. [2015] found that almost all patches produced by GenProg were destructive (i.e., deletion or replacement of code).

   Whilst most of the focus of the discussion with respect to patch quality has focused on the shortcomings within the GenProg-family of search-based approaches (i.e., GenProg, RSRepair, AE), this problem is by no means exclusive to them. The same phenomenon may also be observed in semantics-based approaches, such as SPR and Prophet.

   These results also cast uncertainty on the findings of previous studies within this sub-field of program repair. In particular, it is no longer clear how effective the repair model used by these approaches is, nor is it clear that improvements and parameter tweaks of the search algorithms used by them are effective in producing correct repairs. To move forward with search-based repair, we must first take a step back to better understand its current abilities and limitations.

2. **Effectiveness:** The findings of the aforementioned studies on patch quality also raise questions regarding the actual effectiveness of the search algorithm and repair model used by GenProg. Assuming the existence of a more robust test suite, less susceptible to overfitting, it is no longer clear to what extent these components help GenProg to find correct repairs.

   For automated program repair to see a wider acceptance within the software engineering community, techniques must demonstrate the ability to solve

a significant number of bugs across a diversity of programs. To be truly convincing, automated repair requires the ability to perform arbitrary code transformations—a feat that only search-based repair is capable of.

3. **Efficiency:** To become economically viable, repair techniques need to be able to discover repairs within a reasonable amount of time. Ideally, the time required to find a patch should be measured in hours, or even minutes, rather than days. This problem can be side-stepped through the use of cloud computing resources, which allow the repair process to be split across a large number of compute nodes for a short period of time. In this case, the repair process needs to be cost effective.

   By improving the efficiency of the search, the likelihood of encountering a patch within a fixed resource window is increased, leading to simultaneous gains in effectiveness. Moreover, increases in efficiency will allow larger, more complex search spaces to be explored, allowing the technique to solve a greater number of bugs.

4. **Scalability:** Future program repair techniques will require the ability to produce multiple-edit patches for bugs in arbitrary programs. Due to the limits of symbolic execution associated with semantics-based repair, search-based techniques are the only foreseeable means of tackling this challenge. Despite this, the majority of recent work in search-based program repair has focused on increasing the ability to discover single-edit patches [Long and Rinard, 2015, 2016; Qi et al., 2014; Weimer et al., 2013].

   Genetic algorithms, such as those used by GenProg, PAR, and HDRepair, are the only proposed search technique capable of producing multi-edit patches, in theory. In practice, however, almost all patches generated by those tools can be reduced to a single edit; there is little evidence that this approach is effective at discovering multi-edit patches. With recent results showing that a form of random search outperformed the genetic algorithm used by GenProg, there is a clear need to establish whether GAs are a suitable algorithm for repair, and if not, what alternatives exist.

1.3. Research Questions

Motivated by the challenges facing search-based repair, this thesis seeks to increase our understanding of each of the components of the search, and to explore ways in which those components might be improved.

Motivated by recent results in the use of mutation analysis for fault localisation [Moon et al., 2014a; Papadakis and Le Traon, 2015], we ask the following question:

- **RQ1:** Can the results of candidate patch evaluations, gathered over the course of the search, be used to improve the accuracy of the fault localisation, online?

To understand how the repair model used by program repair techniques can be improved to allow a greater number of bugs to be repaired, we ask the following questions:

- **RQ2:** Is plastic surgery equally effective for all repair actions?
- **RQ3:** Can the effectiveness of plastic surgery be increased through the use of unlabelled code snippets?

Given that GenProg is the only search-based repair technique capable of producing multiple-edit patches, we ask the following questions:

- **RQ4:** Do biases within GenProg's operators degrade its performance?
- **RQ5:** Does fitness help to guide GenProg's search algorithm towards solutions?
- **RQ6:** Is there a more effective search algorithm for generating multiple-edit patches than the genetic algorithm used by GenProg?

To answer these questions, and to avoid the pitfalls of previous research, we propose and use a novel methodology. Our methodology allows researchers to quickly and easily reproduce the results of our experiments. It also provides a platform for others to conduct high-quality program repair research.

1.4. Contributions

Below, we present a summary of the main research contributions made by this thesis.

- We identify a number of weaknesses and compromises in the current approach to conducting APR experiments. Motivated by those problems, we present a better methodology for conducting APR experiments, and a suite of tools for implementing that methodology.
- As part of our theoretical and empirical analyses of GenProg's search algorithm, we identify a number of phenomena and unintended biases, which hamper the efficiency and effectiveness of the search.
- We investigate the role of fitness within the search for repairs, and find that the search is more effective when restricted to consideration of non-destructive patches (i.e., patches that do not fail any of the previously passing tests). Further gains in performance are attained when the number of previously failing tests that are passed by a candidate patch is used to perform selection between

non-destructive patches (i.e., those which do not fail any previously passing tests).

- We propose an alternative search algorithm for finding multiple-edit repairs, based on a greedy algorithm. We demonstrate that our algorithm outperforms GenProg's genetic algorithm in terms of effectiveness, efficiency, and reliability.
- We explore the feasibility of using test suite evaluations for candidate patches, gathered over the course of the search, to increase the accuracy of the fault localisation, online. In contrast to earlier studies of mutation-based fault localisation, conducted on artificial bugs in small programs, our results demonstrate that this information leads to little to no improvement for real-world bugs. We speculate on why these results might be the case, and outline directions for future research.
- Building on previous work concerning the effectiveness of plastic surgery, we investigate its effectiveness at the level of actual repair actions. We report the observed frequencies and graftabilities of a number of new and existing repair actions. As part of this study, we also provide a more formal definition of each of these repair actions.
- We find that repair actions that operate at finer levels of granularity (i.e., expressions), are more amenable to plastic surgery than repair actions with coarser levels of granularity (e.g., blocks, and to a lesser extent, statements). We also find that removing labels from donor code substantially increases the chances of successfully grafting a repair.
- We identify an unintended behaviour within GenProg's implementation, stemming from its use of CIL [Necula et al., 2002], which may severely impact GenProg's performance and prevent it from fixing certain otherwise fixable bugs.

In addition to these research contributions, we have developed a number of tools over the course of this thesis:

- **RepairBox:** a high-performance platform for conducting reproducible empirical studies of program repair, built on top of Docker. This tool was used to conduct all of the relevant experiments within this thesis.
- **Pythia:** a tool for sandboxing test executions and automatically increasing the quality of existing test suites. Pythia was used in combination with RepairBox to reduce the risk of overfitting during our experiments.
- **BugHunter:** a tool for automatically mining bug fixes from Git repositories. In addition to collecting (likely) bug fixing commits, BugHunter also extracts repair action instances from those commits (e.g., a statement was inserted by the fix), and allows a variety of analyses to be conducted on those instances.
- **GenProg:** we performed an extensive refactoring of the GenProg source

code. New features include asynchronous test suite evaluation, JSON configuration files, and a component-based architecture, which allows GenProg to be easily extended with new operators, algorithms and fault localisation approaches.

1.5. Document Structure

The rest of this thesis is structured as follows:

- **Chapter 2 – Background:** provides the reader with the necessary background in automated program repair to understand the rest of the thesis.
- **Chapter 3 – Tools and Techniques:** discusses a number of problems in the current approach to conducting APR experiments, before presenting a set of tools to address them. These tools are used throughout the thesis, to ensure reproducibility of results.
- **Chapter 4 – Fault Localisation:** investigates the feasibility of using the results of candidate patch evaluations, gathered over the course of the search, to increase the accuracy of the fault localisation.
- **Chapter 5 – Repair Model:** measures the frequency and graftability of a set of new and existing repair actions, to improve our understanding of effective repair models, by examining bugs mined from version control histories.
- **Chapter 6 – Search:** conducts a theoretical and empirical analysis of GenProg's search algorithm, before presenting a new search algorithm, based on greedy search.
- **Chapter 7 – Conclusion:** summarises the findings of this thesis, and outlines several directions for future work.

CHAPTER 2

Background

In this chapter, we provide the reader with the necessary background to understand the rest of this thesis. We first provide an overview of the general concepts of program repair, before reviewing a number of the most popular search-based, semantics-based and specification-based repair techniques. Finally, we discuss a number of loosely related techniques covering the space of code transformation and repair more broadly.

2.1. Automated Program Repair

Provided the source code for a program, and an oracle capable of exposing its bugs, APR attempts to find a variant of the program that satisfies the oracle; assuming a sufficiently strong oracle, a repair to those bugs is the output of the repair process. Although a few repair techniques use formal specifications, the majority make use of existing test suites as their oracles instead: failing tests within these suites are used to expose bugs, whilst passing tests are used to prevent the destruction of existing, correct behaviours.

Repair approaches can be split into three categories: *search-based*, *semantics-based*, and *specification-based*. For the most part, each of these approaches shares the same high-level approach to locating and repairing bugs, outlined in Figure 2.1. In the proceeding sections of this chapter, we discuss each of these approaches. For the rest of this section, we explore each of the high-level components in the repair process. To aid this discussion, we look at a fault in a small program, given in Figure 2.2, inspired by the real-world Zune leap year bug [Coldewey, 2008].

Input Program & Donor Pool

Before the repair process begins, each statement within the source code of the program (or the sub-set of files suspected to contain the bug) is identified and uniquely numbered with a statement identifier, or SID, as illustrated in Figure 2.3. Statements are assigned an integer, starting at one, based on the order in which they are encountered when walking through the collection of abstract syntax trees for the program (although the exact SID for a statement makes no difference, only that the SID is guaranteed to be unique). These SIDs are used to identify which areas

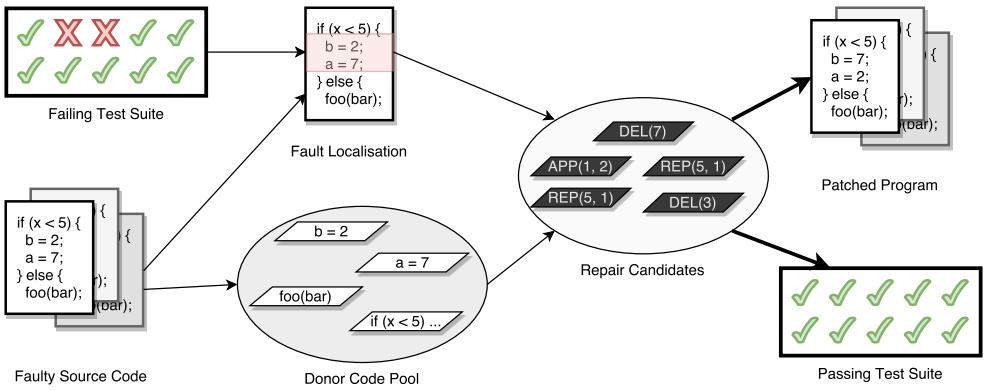

Figure 2.1: The general automated repair process accepts the source code for a faulty program, together with a test suite, containing failed test cases, exposing the faults within the program. From these, the possible locations of the faults are determined, and for some repair approaches, a pool of donor code is generated from the input program. Using the contents of the donor pool, together with a number of basic repair actions, the search generates and evaluates candidate patches, until one is found that passes all tests within the suite.

of the program were executed by the passing and failing test cases during the fault localisation stage of the search.

Whilst this process of annotation occurs, some repair techniques, such as GenProg [Le Goues et al., 2012a] and SPR [Long and Rinard, 2015], extract each of the identified statements and place them into a donor code pool, ready to provide the necessary code to produce repair candidates during the search. Other techniques, such as SearchRepair [Ke et al., 2015], use a pre-computed donor code pool, built from foreign source code, rather than the source code of the program under repair. Others avoid the need for a donor pool entirely, synthesising fragments of code from scratch [Long and Rinard, 2015; Mechtaev et al., 2016; Xuan et al., 2017].

Test Suite

With the exception of techniques where an oracle is provided, either in the form of a (partial) formal specification [Wei et al., 2010], or an alternative version of the program [Tan and Roychoudhury, 2015], a patch is assumed to be correct if it passes all tests within the test suite. For the purposes of conducting research, the test suite is divided into a set of *positive tests*, representing the passing tests (and the behaviours that should be preserved during repair), and a set of *negative tests*, representing the failing tests. The need for this distinction is to ensure the same tests pass or fail when an experiment is repeated.

```
year = 1980;
days = atoi(argv[1]);
while (days > 365) {
  if (isLeapYear(year)) {
    if (days > 366) {
      days -= 366;
      year += 1;
    }
  } else {
    days -= 365;
    year += 1;
  }
}
return year;
```

Figure 2.2: An example bug scenario, adapted from the real-world Zune leap year bug [Coldewey, 2008]. The program accepts a date, given as the number of days since January 1st 1980, and should determine the year to which that date belongs. In the event that `year` is a leap year and `days` ever becomes 366, the program will enter an infinite loop.

Fault Localisation

To reduce the scale of the search landscape and to bring automated repair into the realms of feasibility, the fault localisation stage of the repair process is used to select a (relatively) small number of suspicious locations within the program as candidates for modification. For most existing automated repair techniques, these locations correspond to statements within the program. (Rather than lines, expressions, or any other logical grouping of code.)

To determine the set of faulty statements, and thus the candidates for modification, the first step of the fault localisation process is to compute coverage information for the program. This coverage information describes the statements that are executed by the program for each test within the suite. To generate coverage information for the program, its files (or a sub-set of them, believed to contain the fault) are subject to a process of *source code instrumentation*. Each statement of the program is wrapped by a coverage statement, responsible for monitoring and logging its execution. An example of the output of the coverage generation process is given in Figure 2.5.

From this coverage information, the next step is to aggregate the data into a *fault spectrum*. The fault spectrum concisely describes which statements were executed (or *covered*) by each test, [1] and whether the result of the result of that test was failure

[1] Optionally, rather than recording *whether* a statement was executed, each cell in the spectrum may be used to specify *how many times* a given statement was executed. Whilst this information may be useful in dealing with infinite loops, the need to log individual executions increases the burden on

#	Statement
1	`year = 1980;`
2	`days = atoi(argv[1]);`
3	`while (days > 365) ...`
4	`  if (isLeapYear(year)) ...`
5	`    if (days > 366) ...`
6	`      days -= 366;`
7	`      year += 1;`
8	`    else {}`
9	`  else ...`
10	`   days -= 365;`
11	`   year += 1;`
12	`return year;`

Figure 2.3: Before the repair process can begin, each of the statements within the program is identified and annotated with a unique SID.

or success. An example of such a spectrum is given in Figure 2.6.

Finally, using a lightweight form of fault localisation known as *spectrum-based fault localisation*, the fault spectrum is transformed into a set of suspiciousness values, encoding a measure of the belief that a given statement is responsible for producing the bug. This process of transformation is performed using a *suspiciousness metric*, $\mu(e_p, e_f, n_p, n_f)$, which takes a count of the number of times a given statement was (and was not) executed for passing and failing bugs, respectively. For each statement, the suspiciousness metric transforms those counters into a real number, or *suspiciousness* value, where higher values indicate a greater degree of belief that the statement is (partly) responsible for producing the bug.

For the majority of automated repair techniques, a relatively simple suspiciousness metric is employed. In the case of GenProg, its metric assigns a suspiciousness of 1.0 to statements executed exclusively by failing tests, 0.1 to statements executed by both passing and failing tests, and 0.0 to statements that are not covered by a failing test (thus eliminating them from consideration by the repair). More details on the fault localisation process can be found in Chapter 4.

Repair Candidates

Once the fault localisation and donor code pool have been computed, the two are combined according to a *repair model* to generate the space of possible repairs. The

the instrumented program, causing it to run considerably slower. This slowdown is made worse when the user wishes to generate an *execution path* for each test, describing the *order* in which statements were executed. For most real-world programs, the disk requirements of this path file, potentially of the order of GBs per test, make such an approach infeasible.

Test	Input	Expected	Passed
P1	-366	1980	✓
P2	-100	1980	✓
P3	0	1980	✓
P4	365	1980	✓
P5	367	1981	✓
P6	1000	1982	✓
P7	1826	1984	✓
P8	2000	1985	✓
N1	10593	2008	✗
N2	12054	2012	✗
N3	1827	1984	✗
N4	366	1980	✗

Figure 2.4: A test suite for our running example, described by inputs and expected outputs for the program. Failing tests are assigned labels starting with the letter "N", indicating that they are negative test cases. Conversely, passing tests are assigned a label starting with the letter "P", indicating that they are positive test cases.

repair model of a technique is composed of a series of repair actions, each of which abstractly describe the kinds of changes (or mutations) to the program that can be made. For instance, the repair model of GenProg contains repair actions that allow the insertion, replacement, and deletion of statements. A more detailed definition of repair models is provided in Chapter 5.

An upper bound on the size of the search space S for a given problem and (search-based) repair technique is:

$$S \leq \sum_{i=1}^{l} [F \cdot (DR)]^i \tag{2.1}$$

where F is the number of suspicious statements, D is the number of donor code fragments, R is the number of repair actions, and l is the maximum number of edits that may be contained within the patch.

2.2. Search-Based Repair

Search-based repair techniques employ meta-heuristic search algorithms to explore the space of possible repairs. Some of these techniques (e.g., GenProg and PAR [Kim et al., 2013]) attempt to discover partial solutions during the search by treating repair as an optimisation problem. Others, such as RSRepair [Qi et al., 2014] and AE

#	Statement	P1	P6	N1	N4
1	`year = 1980;`	•	•	•	•
2	`days = atoi(argv[1]);`	•	•	•	•
3	`while (days > 365) ...`	•	•	•	•
4	`  if (isLeapYear(year)) ...`		•	•	•
5	`    if (days > 366) ...`		•	•	•
6	`      days -= 366;`		•	•	
7	`      year += 1;`		•	•	
8	`    else {}`			•	•
9	`  else ...`		•	•	
10	`   days -= 365;`		•	•	
11	`   year += 1;`		•	•	
12	`return year;`	•	•		

Figure 2.5: An example of the output produced by the coverage generation process. The columns on the right show which statements are covered by a particular test. For the sake of brevity, we omit coverage for the majority of the test suite.

[Weimer et al., 2013], treat repair as a decision problem, opting for higher efficiencies in the evaluation of candidate patches over the identification and exploitation of partial solution information. In all cases, candidate repairs are *generated* by the search, and subject to *evaluation*. This process of evaluation involves compiling the resulting program, and determining the outcomes of (a sub-set of) its test suite. The processes of generation and evaluation continue until either an acceptable repair is found (i.e., one which passes all of the tests) or a resource limit is reached.

For the rest of this section, we review the majority of search-based repair techniques.

Co-Evolutionary Program Repair

Arcuri and Yao [2008] introduce the first search-based program technique, a year prior to the publication of GenProg. Like GenProg, their technique uses evolutionary computation, but whereas GenProg assesses the fitness of an individual based on a fixed number of tests, their approach uses competitive co-evolution to evolve a more robust set of tests. To automatically generate test cases—and thereby facilitate co-evolution—users are required to provide a formal specification for the program.

Unlike all other search-based techniques, which operate on the abstract syntax trees of a real-world program, whether directly, through representations such as the AST/WP, or indirectly, through variants of the Patch representation, Arcuri and Yao [2008]'s approach operates on a bespoke Strong-Typed Genetic Programming (STGP) representation [Montana, 1995]. As with traditional applications of GP, programs are

#	Statement	e_p	e_f	n_p	n_f	μ
1	`year = 1980;`	8	4	0	0	0.1
2	`days = atoi(argv[1]);`	8	4	0	0	0.1
3	`while (days > 365) ...`	8	4	0	0	0.1
4	`  if (isLeapYear(year)) ...`	4	4	4	0	0.1
5	`    if (days > 366) ...`	4	3	4	1	0.1
6	`      days -= 366;`	4	3	4	1	0.1
7	`      year += 1;`	4	3	4	1	0.1
8	`    else {}`	0	4	0	0	1.0
9	`  else ...`	3	3	5	1	0.1
10	`   days -= 365;`	3	3	5	1	0.1
11	`   year += 1;`	3	3	5	1	0.1
12	`return year;`	8	0	0	4	0.0

Figure 2.6: An example of a fault spectrum for the Zune bug, together with the suspiciousness values for each statement, as computed by GenProg's suspiciousness metric.

operated upon directly and mutations are synthesised from a set of primitives. Given the relative proximity of the buggy program to its (correctly) repaired form—compared to the large distances between the origin and a solution when attempting to synthesise a program *de novo*—Arcuri and Yao [2008] forgo the need for crossover.

Although this representation is Turing-complete and incorporates a number of high-level operations similar to those in C and Java, it significantly lacks the expressiveness of those languages. In the general case, programs written in this dialect of STGP may be easily transformed into C or Java. Transforming C or Java programs into this STGP dialect is less trivial, however. Even if C or Java programs could be transformed into this dialect of STGP, representing entire programs—or even single files—would require vast amounts of memory and incur a significant overhead to performance, as observed in earlier forms of GenProg using the AST/WP representation [Le Goues et al., 2012a]. Arcuri and Yao [2008]'s work is therefore a powerful proof of concept of search-based program repair, rather than its first practical application.

Rather than solely measuring fitness as a function of the number of passing and failing tests, Arcuri and Yao [2008] use a richer fitness function, defined in Equation 2.2:

$$f(g) = \frac{N(g)}{N(g)+1} + \frac{E(g)}{E(g)+1} + \sum_{t \in T_i} d_P(t, g(t)) \tag{2.2}$$

where g is a candidate solution, $N(g)$ is the number of nodes in its tree, $E(g)$ is the number of exceptions encountered during its execution, and $d_P(t, g(t))$ is a measure

of the distance between the expected and observed result for the execution of test t.

Unlike subsequent approaches, the number of nodes for a given individual $N(g)$ is directly incorporated into the fitness function, acting as a form of bloat control; a selective advantage is awarded to programs that are shorter than others. If the size of the program is less than or equal to $\delta N(\text{buggy})$, where $N(\text{buggy})$ is the number of nodes in the original program, then this bloat term is replaced by the constant 1.

The fitness function also incorporates a term based on the number of exceptions produced by a given individual, $E(g)$. Sharing a similar intuition to later work on the use of failure-obliviousness for evolving Ruby programs [Timperley, 2013; Timperley and Stepney, 2014], fallback semantics are provided for unsafe operations. Specifically, both division by zero, and out-of-bounds array accesses are set to return zero instead. In theory, and as demonstrated by [Timperley and Stepney, 2014], allowing execution to persist in the face of such exceptions creates a smoother fitness landscape, which may allow certain classes of multiple-line errors to be solved more easily.

Finally, Arcuri and Yao [2008] harness the results of the test suite by summing the distance between the intended and observed outcomes $d_P(t, g(t))$ for each test t. Whereas later work treats the outcome of a test case as a binary variable (i.e., pass/fail), their distance metric shares more in common with previous work in genetic programming, by measuring the distance between the observed and intended return value of the function under repair.

- For tests where the function returns the intended outcome, $d_P = 0$.
- In cases where the function returns an unexpected outcome, $d_P > 0$. The measurement of the magnitude of the difference between the two outcomes is performed by a necessarily arbitrary distance function, as is the case with traditional genetic programming. This distance function is automatically generated from a formal specification for the program.

To avoid solutions overfitting to a fixed test suite, the tests used for evaluation are competitively co-evolved in a separate population (of size 32). For the particular program studied in the paper—an implementation of bubble-sort—each test within the population is represented by a variable-length list of integers, representing an input list to the sorting algorithm. For every 5 generations that the solution population is evolved, the test population is subject to 1024 generations of evolution against the best program in the current population. The fitness of a given test t is measured using the fitness function described in Equation 2.3:

$$f(t) = \sum_{g \in G_i} d_P(t, g(t)) \tag{2.3}$$

where $g \in G_i$ is an individual from the i-th generation, and $d_P(t, g(t))$ is the dis-

tance between the expected and observed outcome for this individual's test execution. At the end of each of these 1024 generations, the best test case within the population is added to a *Hall of Fame* [Rosin and Belew, 1997], which then becomes the test suite for evaluating candidate patches.

In summary, Arcuri and Yao [2008]'s approach introduces the breakthrough idea of using genetic programming to evolve patches, rather than entire programs—an idea that underpins the seminal work in the field. Many of the ideas introduced in their work, such as bloat control and the use of a richer fitness function, have proved to be ahead of their time. Unlike the other approaches reviewed in this section, all of which are usable repair tools, Arcuri and Yao [2008]'s work is instead a proof of concept.

GenProg

GenProg can be viewed as both the seminal approach to the automated repair of real-world programs, and the first viable search-based repair technique. At its core, GenProg uses a form of genetic algorithm to search for its repairs. Like all other genetic algorithms, GenProg's search process can be seen as a continual cycle of *generation* and *evaluation*, also referred to as *validation* within the automated repair literature [Le Goues et al., 2012a; Long and Rinard, 2015].

The generation stage is responsible for generating a *population* of potential solutions, based on (implicit) knowledge gained about previously encountered solutions through evaluation of their fitness. Solutions deemed to be closer to a repair, where distance is measured by a weighted sum of the number of the positive and negative test cases passed by the candidate, are more likely to be used to inform the generation of future candidates. This assumes that fitness is correlated with edit distance, and that solutions which pass a higher proportion of the test suite are more likely to be close to a repair than solutions that pass fewer tests.

From a high-level perspective, GenProg, shares the same approach to generation as all other genetic algorithms. Following a stage of initialisation, wherein a population of single-edit patches is randomly sampled, the process of generation is sub-divided into sub-processes of selection, crossover, and mutation. The resulting search algorithm is given in Algorithm 1, together with a description of each of its stages.

Algorithm 1: GenProg's search algorithm

```
population ← initialise() ;
while solution not found and resources not exhausted do
    parents ← select(population) ;
    population ← crossover(parents) ;
    population ← mutation(population) ;
    evaluate(population) ;
end
```

Individual Representation

To represent a candidate repair, each individual within the population is encoded as a sequence of statement-level modifications to the abstract syntax tree of the faulty program. To generate the corresponding repair, each of these modifications, or *edits*, is applied in sequence. Each edit within this sequence, or *patch*, applies a particular transformation to the statement associated with a given SID. The particular *repair model* used by GenProg allows for three different kinds of statement transformation, described below:

- **Deletion:** The edit `DELETE x` permanently removes the statement with SID x from the program, together with any child statements that may be attached.
- **Append:** The edit `APPEND x y` copies the statement with SID y from the original program, and appends it immediately after the statement with SID x.
- **Replacement:** The edit `REPLACE x y` copies the statement with SID y from the original program, then permanently replaces the statement at SID x with the copied statement.

Selection

The selection phase within GenProg selects which candidate repairs from the population of the previous generation should be chosen as parents, whose edits will be used to construct the population for the next generation.

By default, this process of selection is performed using *tournament selection*. Tournament selection chooses n individuals as parents through a series of n *tournaments*. For each tournament, k individuals are chosen from the population to be participants, at random with replacement. The best individual within each tournament is granted entry into the next population.

The setting of k (a.k.a. *tournament size*) can be used to control *selection pressure*. Low values of k exhibit little selection pressure, allowing more subpar solutions to be accepted into the population. Conversely, high values of k increase the selection pressure, driving selection towards only the fittest solutions. By default, GenProg uses a minimal $k = 2$ ($k = 1$ would be equivalent to uniform random selection), preferring exploration of the search space over exploitation of its fittest solutions.

Crossover

Crossover within GenProg groups the selected parents into pairs and subjects each pair to a crossover operation with probability P_c, also known as the crossover rate. In the event of an application of the crossover operator, a pair of individuals are generated from the edits of the two selected parents and added to the offspring.

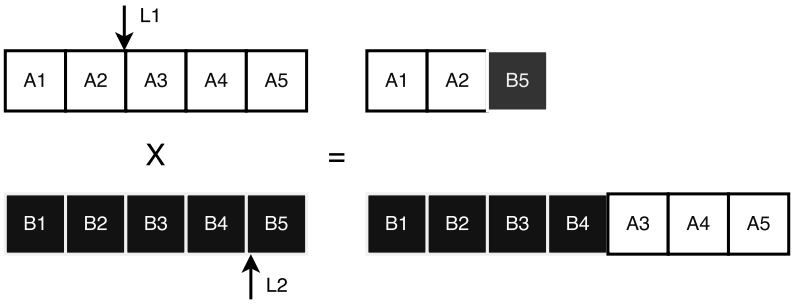

Figure 2.7: An illustration of GenProg's one-point crossover operator. Two parents A and B are accepted as input, and each is split into two parts at a random point. The first part of A is combined with the second part of B to form a child C. Similarly, the first part of B is combined with the second part of A to form a child D.

After iterating across all pairs of parents, a new proto-population is formed from the union of the multi-set of parents and the multi-set of offspring. Unlike most genetic algorithms [Eiben and Smith, 2015], where parents are replaced by their offspring in the event of a crossover, GenProg allows parents to be retained. This behaviour is similar to a $(\mu + \lambda)$ evolutionary strategy [Eiben and Smith, 2015], except that parents may also be subject to further mutation.

By default, GenProg uses a variant of variable-length one-point crossover, wherein the lists of edits for each parent are split into two at a random point and then joined together, as illustrated in Figure 2.7. Alternatively, a patch sub-set operator may be used, which has previously been shown to improve performance [Le Goues et al., 2012c].

Mutation

Following crossover, each individual within the proto-population is subject to mutation with probability P_m, also known as the mutation rate, to form the population. The mutation operator within GenProg accepts a single patch as its input, and returns a new patch, formed by appending a newly sampled edit to the input patch.

To generate the new edit, GenProg first selects a statement as the subject of the edit, using the distribution defined by the fault localisation information. After a statement has been selected as the site of the edit, GenProg decides whether the edit should be a replacement, insertion, or deletion. By default, each edit type has an equal probability of being selected, although their probabilities may be adjusted to improve performance [Le Goues et al., 2012c]. Once an edit location and type have been chosen, GenProg produces the set of all possible edits of that particular type at the given location, and selects one of them at random as the chosen edit.

Test Case Sampling

In an effort to reduce the number of test case evaluations required to find an acceptable patch, Fast et al. [2010] introduce the concept of test suite sampling to automated repair. Rather than computing the fitness of a candidate patch by subjecting it to the whole test suite, each candidate is instead subject to its own random sample of the test suite. Each sample is composed of all the negative tests, together with a random selection of $n\%$ of the positive test cases, where $n\%$ is usually set to 10% for larger problems, such as those in the ManyBugs benchmark set [Le Goues et al., 2015]. If the candidate passes all tests in the sample, then, and only then, is it subject to rest of the test suite, to determine whether it is an acceptable repair.

Fast et al. [2010] find that despite the noise introduced into the fitness function, the use of sampling increased the performance of GenProg by 81%, when measured by the number of test case evaluations required to find a repair.

Repair Minimisation

Upon finding an acceptable repair that passes all tests within the test suite, GenProg subjects the repair to a minimisation process, wherein redundant edits which have no effect upon the outcome of the patch are removed [Le Goues et al., 2012a]. To facilitate the minimisation process, the patch is first converted to an AST difference using a variant of the DiffX XML differencing algorithm [Al-Ekram et al., 2005], tailored to operate with CIL ASTs. This process transforms the patch into a series of structured tree operations, describing the steps required to transform the original program into its patched form, e.g., "Delete the node rooted at k". The minimised repair is then computed by applying the delta debugging algorithm [Zeller and Hildebrandt, 2002] to the sequence of edits, reducing them to their minimal form.

Revealingly, in almost all cases, patches yielded by the minimisation process consist of only a single edit, indicating that the algorithm is not finding and combining partial edits, as would be expected for a traditional genetic algorithm [Holland, 1992, 2000].

RSRepair

Qi et al. [2014] introduce RSRepair, an alternative search-based repair tool, to assess whether GenProg's search algorithm could be outperformed by random search. The RSRepair algorithm, given in Algorithm 2, uses the same fault localisation and search operators as used by GenProg, allowing a direct comparison between their

search algorithms.

Algorithm 2: RSRepair algorithm

```
while solution not found and resources not exhausted do
    location ← sample(faultLocalisation) ;
    edit ← sampleEdit(location) ;
    candidate ← apply(program, edit) ;
    outcomes ← evaluate(candidate) ;
    if candidate passes all tests then
        return candidate
end
```

As a result of abandoning the need for fitness, RSRepair is able to treat automated repair as a decision problem, rather than an optimisation problem. This allows the search to terminate the evaluation of a candidate patch on the first observation of failure, substantially reducing the number of test case evaluations required by each candidate. Additionally, this allows RSRepair to make use of test case prioritisation [Rothermel et al., 2001], whereby tests that have been observed to be more likely to fail over the course of the search are executed first, maximising the rate of incorrect patch detection (and by extension, improving efficiency, as measured by the number of test case evaluations).

Another important difference between RSRepair and GenProg is that RSRepair restricts itself to the much smaller space of single-edit patches; in contrast, GenProg patches may contain an arbitrary number of edits. Since most patches reduce to only a single edit, this allows RSRepair to consider substantially fewer candidates than GenProg.

Results show that on 100 runs of a sub-set of the ManyBugs dataset, RSRepair achieves a higher success rate (measured by the percentage of runs that found a repair) than GenProg on 24/25 problems, and a lower mean number of test case evaluations for 24/25 problems [Qi et al., 2014].

AE

Weimer et al. [2013] introduce the "Adaptive Search, Program Equivalence" (AE) repair technique. Similar to RSRepair, AE operates using the same fault localisation method as GenProg, and with a sub-set of its mutation operators. By default, AE restricts itself to consideration of single edits patches, transforming the problem into a decision problem. AE searches for repairs by iterating through each of the possible `Delete` and `Append` operations at each statement within the program, in order of suspiciousness.

By treating automated repair as a decision problem, AE is able to use test case prioritisation to increase the efficiency of the search. To achieve further efficiency gains, AE performs test suite reduction for each candidate patch, removing test cases

whose results are not affected by the patch, by checking statement coverage of each test.

In addition to its adaptive search process, AE introduces the notion of approximate program equivalency checking, which uses a series of syntactic and dataflow analyses to find and prune equivalent repairs from the search space. To determine approximate program equivalence, the following three techniques are used:

1. *Syntactic equality:* By exploiting the observation that programs that are syntactically identical are also semantically equivalent, AE removes duplicates of identical statements from the pool of donor statements.

2. *Dead code elimination:* Using CIL's [Necula et al., 2002] dataflow analysis framework, AE is able to detect whether a particular mutation would insert dead code into the program; this information is used to eliminate variants from the search space.

3. *Instruction scheduling:* In many cases, a statement may be inserted into the program at a number of different points, all of which will yield the same semantics. AE identifies instances where two (or more) insertions would produce equivalent results, and removes redundant insertions from the search space.

Results taken from a comparison between AE and GenProg [Weimer et al., 2013], using the 55 bugs from the ManyBugs benchmark suite that were previously solved by GenProg, showed that AE was able to fix 53 of the 55 bugs. In total, AE required 186 test suite evaluations across the search, compared to the 3252 required by GenProg, giving an order of magnitude improvement in test case efficiency.

PAR

Kim et al. [2013] propose a search-based program repair technique for repairing Java programs, PAR, based on applying hand-crafted repair operators to generate repairs. With the exception of its repair operators, PAR is identical to GenProg in all other respects. These repair operators, or fix templates, described below, were produced through manually inspection of more than 60,000 human-written patches in open-source projects for common fix patterns.

1. **Null checker:** inserts an if-statement checking that all objects within context at the target statement are not null.

2. **Parameter replacement:** replaces a randomly selected parameter from an arbitrary function call at the fault location with a compatible variable or expression.

3. **Method replacement:** replaces the target method of a randomly selected method call at the fault location with the name of a visible method with compatible parameters and return type.

4. **Parameter addition or removal:** adds or removes a single parameter from a method call at the fault location, provided the method has a compatible overloaded sibling. As with the parameter replacement transformation, suitable variables and expressions are extracted from within the scope of the fault statement to serve as additional parameters.

5. **Object initialisation:** introduces a new assignment statement, assigning a newly-created object to a new local variable, before using the object as a parameter of a method invocation.

6. **Sequence exchange:** swaps the order of a sequence of identified statements within the program, where each statement is exchanged with its most similar counterpart.

7. **Range checking:** inserts a series of if-statements, ensuring that all index variables of array accesses are within lower and upper bounds.

8. **Collection size checking:** inserts an if-statement checking that an index variable is smaller than the size of a collection object if faulty statements have collection object references.

9. **Lower bound setter:** assigns a lower bound value to an index variable if the faulty statements have array accesses.

10. **Upper bound setter:** assigns an upper bound value to an index variable if the faulty statements contain array accesses.

11. **Off-by-one mutator:** modifies an index variable by 1 if the faulty statements have array accesses.

12. **Class cast checker:** inserts an if-statement, ensuring that castees implement appropriate types when casting operations are performed.

13. **Caster mutator:** replaces the casting type of a casting operator within the faulty statement with another type.

14. **Castee mutator:** replaces the subject of a casting operation within the faulty statement with a different variable, sharing a similar name.

15. **Expression changer:** replaces a conditional expression in the faulty statement with a similar conditional expression, taken from the same source code.

16. **Expression adder:** inserts a similar expression into an existing expression in a faulty statement, provided it contains a conditional expression.

Across 119 real-world bugs, Kim et al. [2013] found that PAR was able to produce repairs for 27 of them, compared to 16 bug fixes achieved by GenProg. Furthermore, results of a human study involving 89 students and 164 developers found that patches generated by PAR were more likely to be accepted than those produced by GenProg.

Defect class according to...	Examples of defect class
The root cause	Incorrect variable initialisation, incorrect configuration, etc.
The symptom	Segmentation fault, null pointer exceptions, memory exhaustion, etc.
The fix	Adding an input check, changing a method call, restoring an invariant, etc.

Table 2.1: Examples of different types of defect classes and the shared properties by which they are defined. Adapted from [Monperrus, 2014].

A year after Kim et al. [2013] introduced PAR, Monperrus [2014] delivered a critical review of the repair, challenging both its methodology and results.

After introducing the concept of defect classes—a family of bugs sharing some shared property, outlined in Table 2.1—Monperrus [2014] argues that to fairly evaluate repair techniques, one must compose the dataset according to the defect classes one believes the tool addresses. Similarly, to achieve a fair comparison between techniques, one must take care to account for the different defect classes addressed by each. Failing to do so, one may compose an unbalanced dataset, overrepresenting the defect classes handled by one technique, whilst underrepresenting those of another.

Monperrus [2014] highlights that Kim et al. [2013] fail to identify *a priori*, or to discuss *a posteriori*, the defect classes tackled by PAR—a problem common to a number of repair techniques. Whereas GenProg aims to provide a highly generic approach to program repair, theoretically capable of addressing all defect classes, PAR focuses on addressing common programmer mistakes fitting its predefined repair templates. Monperrus [2014] argues that Kim et al. [2013] evaluate on a dataset that may unintentionally favour PAR, and that given the two techniques address different defect classes, that such a comparison is meaningless. Upon closer inspection, Monperrus [2014] found that the majority of PAR's repairs were produced using either the "Null Pointer Checker" or the "Expression Adder, Remover, Replacer"—the remaining operators fixed only a single bug each.

In addition to identifying problems in the (lack of) methodology used to construct the dataset, Monperrus [2014] recognises a number of flawed assumptions underlying the human study used to evaluate patch acceptability. Although the participants of the study included 67 developers, none of them were responsible for maintaining the programs used in the study. Therefore, the experiment assumes that the developer is able to judge the quality of a patch without knowledge of the codebase or the wider context of the bug. Monperrus [2014] argues that without prior experience with the codebase, participants are unlikely to accurately determine the quality and correctness of a patch within a reasonable amount of time (e.g., 30 minutes), armed only with the patch and a link to the bug report. Under these circumstances, then,

Monperrus [2014] believes that the participants conflate the notion of acceptability with understandability; that is, participants select patches on the basis of whether they or not they "look good".

To explain why this conflation is a problem, Monperrus [2014] discusses the different requirements of autonomous program repair techniques and patch recommendation systems:

- Automated repair systems are intended to operate mostly at run-time, without human assistance, and their patches are only required to serve as a temporary solution.
- Conversely, recommendation systems are required to generate a set of (partial) transformations, for a human developer during development or maintenance, who is given the responsibility of selecting an acceptable change from amongst them; the produced changes are expected to be long-lasting, rather than ephemeral.

As a consequence of the differing assumptions and requirements of these two approaches, each should be assessed with its own evaluation criteria, in terms of *understandability*, *correctness*, and *completeness*: [2]

- For fully automated repair techniques, such as GenProg and SPR, the only considerations should be that correct and complete solutions are generated. Given the temporary nature of its fixes, alien-looking solutions—lacking understandability—should be treated with equal consideration.
- In contrast, patch suggestion systems should concern themselves less with the completeness and correctness of a patch, and more with producing an understandable, possibly partial solution that is likely to persist.

Given that PAR is presented as an automated repair technique, and not as a fix suggestion tool, the methodology used to evaluate acceptability is compromised by the bias towards understandable repairs and against alien repairs.

Staged Program Repair

As a more efficient and higher quality means of generating repairs for C programs, Long and Rinard [2015] introduce Staged Program Repair (SPR): a search-based repair technique which combines parameterised transformation schemas and abstract conditions to reduce the size of the search space. In lieu of GenProg's generic statement-level repair operators, SPR uses a set of parameterised *transformation schemas*—each designed to tackle a particular class of defects—to generate its repairs. Given a particular transformation schema, SPR performs *target value search*

[2] In this context, "completeness" refers to the ability of the repair tool to generate an executable variant of the program. (i.e., all of the information required to repair the program should be encoded within the patch.)

to determine whether there exists a parameter which would cause the schema to yield an acceptable repair. This set of schemas is outlined below.

1. **Condition Refinement:** Transforms the condition of a given if-statement by conjoining or disjoining an abstract condition to its original condition.
2. **Condition Introduction:** Transforms a given statement such that the statement is only executed when an abstract condition is true.
3. **Conditional Control Flow Introduction:** Inserts a new control-flow statement (e.g., return, break, goto) that executes only when an abstract condition is true.
4. **Insert Initialisation:** Inserts a memory-initialisation statement before a selected statement.
5. **Value Replacement:** Given a target statement, this transformation performs one of the following replacements: one of its variables with another, an invoked function with another, or a constant with another constant.
6. **Copy and Replace:** Prepends an existing statement from elsewhere in the program to the program point immediately before a selected statement, before applying a *value replacement* transformation to the prepended statement.

After ranking the suspiciousness of statements according to spectrum-based fault localisation, SPR attempts each of these transformation schemas—in the order given above—at each statement, in order of their suspiciousness. For the first three of these transformation schemas, SPR uses *angelic debugging* [Chandra et al., 2011] to determine if there exists an abstract condition which would result in an acceptable repair. If so, SPR synthesises a suitable condition using the angelic values for that If condition with its *condition synthesis* algorithm. In cases where no abstract condition is found, SPR avoids the need to attempt concrete transformations, thus increasing the efficiency of the search. Similarly, SPR uses abstract expressions within print statements, allowing a suitable concrete expression to be synthesised from the observed, correct print behaviour.

Unlike GenProg, SPR is unable to compose multiple-edit fixes, opting to focus on single-edit patch generation instead. Long and Rinard [2015] state: "It is unclear how to combine multiple transformations and still efficiently explore the enlargened space."

Intentionally, SPR lacks a statement deletion operator, due to the previous findings by Qi et al. [2015], showing that the majority of plausible, but incorrect patches generated by GenProg were the result of deletion. Although this decision prevents explicit functionality deletion—and thus repairs which require such deletion—it does not prevent SPR from yielding implicit deletions, by replacing conditions with contradictions and tautologies [Mechtaev et al., 2016]. Mechtaev et al. [2016] found that 80% of the repairs generated by SPR for the libtiff bug scenarios involved implicit functionality deletion (whereas only 21% of repairs generated by Angelix were functionality-deleting).

To assess the efficacy of SPR, Long and Rinard [2015] evaluated it against 69 bugs from the ManyBugs suite.[3] Given the tendency for automatically generated repairs to overfit the test suite, the authors themselves assessed the correctness of the patches produced by SIR. Whilst any form of patch quality assessment is better than none at all, using humans to evaluate correctness is fraught with potential biases, and the possibility of false positives and negatives—all the more so when evaluating one's own patches against another system. A more robust solution—albeit far from perfect—is to use a held-out set of manually or automatically generated whitebox tests to independently verify the repairs.

Across the 69 bugs, SPR's search space contained repairs for 19 of them, and for 11 of the 19, the first plausible repair that was presented by SPR was correct. By comparison, GenProg generated correct fixes for 2 of the 69 bugs [Qi et al., 2015].

In later work, Long and Rinard [2016] proposed a variant of SPR, known as Prophet, wherein each candidate patch within the search space was ranked according to a learned model of likely bug fixes. This model, learnt by applying machine learning over a corpus of several hundred historical bug fixes, accepts as its input, a set of syntactic features for the original and modified versions of a line, each described as a binary variable, and produces a scalar value, describing the likelihood that the given fix is correct. To attain scores for each candidate fix, and thus rank them, the resulting score from the model for a given fix is combined with its fault localisation suspiciousness score, by computing their product. Long and Rinard [2016] found that the use of Prophet reduced the average time taken to find a repair, and allowed two more bugs to be fixed within the 12-hour repair window.

History-Driven Program Repair

To improve patch quality and search efficiency, Le et al. [2016] introduce a novel "history-driven" repair technique for for Java programs: HDRepair. Like, GenProg and PAR, this technique employs genetic algorithms to sample and combine edits into candidate patches. Whereas GenProg and PAR treat all edits equally, however, history-driven repair exploits knowledge mined from version control repositories to bias the search towards repairs that more closely resemble human-written repairs.

To achieve this, 3000 historical bug fixes are first mined from GitHub from across 700 Java projects, before being transformed into a graph-based representation. To produce this graph representation, an AST difference for each modified source file is first generated using GumTree [Falleri et al., 2014], a state-of-the-art, open-source source code differencing tool. Since the GumTree edit script still contains information specific to particular variable names, rather than capturing an abstract type of change (e.g., a change of method name), this script is transformed into a labelled, directed graph, capable of abstractly representing the change and capturing its surrounding context. Each edge within this graph $\{P, C\}$ represents a modification to

[3] "Bug scenarios" that were in fact feature additions were removed from consideration.

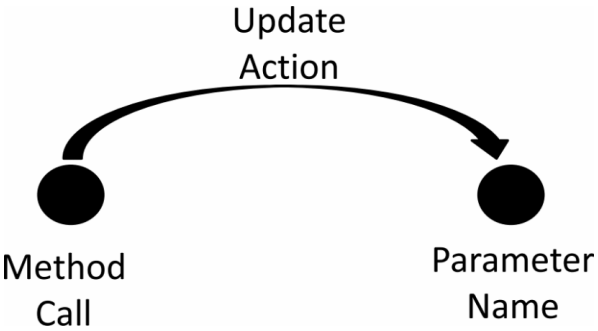

Figure 2.8: An example change graph produced during the offline, bug fix mining stage of history-driven program repair [Le et al., 2016]. Here, the change graph shows the name of a parameter being updated within the context of a method call.

a child node C within the context of its parent P. An example of such a change graph is given in Figure 2.8.

Once each bug fix has been transformed into its corresponding abstract change graph, the set of graphs is subject to a process of graph mining using gSpan [Yan and Han, 2002], wherein the largest, most frequent bug fix patterns are mined. For each mined pattern, all of its vertices, edges, and the supergraphs that contain that pattern are recorded to disk, forming a *historical bug fix database*. Using this information, the number of supergraphs of a given pattern can be used to find its frequency.

During the online phase of the search, candidate repairs are generated stochastically, using a similar process of generating repairs to GenProg and PAR, wherein a single edit is added onto an existing patch at each mutation step. An outline of the search algorithm employed by History-Driven Program Repair is given in Algorithm 4.

Algorithm 4: The SELECT function randomly selects a number of individuals from a population, either uniformly or weighted by a given function. The tunable parameters are: $PopSize$ (size of population), M (number of desired solutions), E (size of the initial population), L (number of locations considered during mutation). Sourced from [Le et al., 2016].

Input: $BugProg$: Buggy program
Input: $FaultLocs$: Fault locations
Input: $NegTests$: Initially failing test cases
Input: ops: Possible operators
Input: $params$: Tunable parameters $PopSize, M, E, L$
Output: A ranked list of possible solutions

`editFreq(`*cand*`)` **begin**
    $N \leftarrow |cand|$;
    return $\frac{\sum_{i=0}^{N-1} \text{FIXPAR}(cand_i)}{N}$
end
`mutate(`*cand*`)` **begin**
    $locs \leftarrow \text{SELECT}(FaultLocs, L)$;
    $pool \leftarrow \emptyset$;
    foreach $f \in locs$ **do**
        $op_f \leftarrow \bigcup_{op \in \text{APPLIES}(ops, loc)} \text{INSTANT}(\text{op,loc})$;
        $cand \leftarrow cand + \text{SELECT}(op_f, 1)$;
        $pool \leftarrow pool \cup \{cand'\}$;
    end
    return $\text{SELECT}(pool, 1, editFreq)$
end
`search()` **begin**
    $Solutions \leftarrow \emptyset$;
    $Pop \leftarrow \{E \text{ empty patches}\}$;
    while $|Pop| < PopSize$ **do**
        $Pop \leftarrow Pop \cup \text{MUTATE}([])$;
    end
    repeat
        foreach $c \in Pop$ **do**
            if $c \notin Solutions$ **then**
                if c *passes* $NegTests$ **then**
                    $Solutions \leftarrow Solutions \cup \{c\}$;
                else
                    $c \leftarrow \text{MUTATE}(c)$;
                end
            end
        end
    until $|Solutions| = m$;
    return *Solutions*
end

As with GenProg and PAR, a population composed of single-edit solutions is generated during the initialisation stage, before being supplemented with a given number of empty patches. Unlike GenProg and PAR, there is no selection between individuals; rather, each individual within the population is subject to mutation and carried as-is into the next generation, without being subject to crossover. As a result, the algorithm ignores the outcome of the test suite for a given solution—except in cases where a repair is found—allowing it to terminate upon the first instance of failure, as with RSRepair and AE. Instead, a form of selection is introduced into the process of mutation, wherein several edits are generated and only a single is selected and added to the patch, based upon its likeness to historical bug fixes within the mined database.

This selection between edits uses stochastic universal sampling to pseudo-randomly sample an edit based upon its corresponding frequency within the historical bug fix database. Specifically, each candidate edit is assigned a score, according to Equation 2.4, based upon the mean frequency (within the database) of each of the edits in the mutant patch that contains the edit under consideration.

$$score(patch) = \frac{\sum_{i=0}^{N-1} FixPar(patch_i)}{N} \tag{2.4}$$

The rationale given by Le et al. [2016] for using the mean frequency of the mutant patch—rather than the frequency of the singular edit—is that it allows the score for low frequency edits to be dampened. This increases the likelihood of their selection (although the distribution given by considering singular edit frequencies is isotone).

Rather than terminating upon the discovery of a repair, like PAR and GenProg, HDRepair continues to search until a specified number of solutions have been found, or a time limit has been reached. The user is then presented with each of the discovered fixes, ranked and ordered according to their mean frequencies. Unlike all other repair techniques, HDRepair does not validate its candidate repairs against the previously passing test cases. Consequently, repairs reported by HDRepair may introduce new bugs into the program. In addition to employing a novel search procedure, this repair technique also uses a combination of existing repair models and mutation testing operators to construct repairs, detailed in Table 2.2.

Results taken from an evaluation of the technique on 90 bugs taken from 5 programs within the Defects4J dataset [Just et al., 2014] demonstrate that HDRepair produces correct repairs for 23 bugs, compared to 4 and 1 fixed by PAR and GenProg respectively. It is unclear whether off-the-shelf versions of PAR and GenProg were used; if so, then it is important to note that the definition of a repair is different between these tools—PAR and GenProg require that a patch passes all of the test suite, whereas HDRepair requires that it only pass the previously failing tests. Whilst these results may suggest a higher effectiveness (i.e., HDRepair fixes more bugs), they also increase the wall-clock time taken to find a repair.

GenProg Mutation Operators	
Insert Statement	Insert a statement before or after a given statement
Replace Statement	Replaces a statement with another from the program
Delete Statement	Removes a statement from the program
Mutation Testing Operators	
Insert Type Cast	Cast an object to a compatible type
Delete Type Cast	Remove a type cast from an object
Change Type Cast	Replace a type cast with a compatible cast
Change Infix Expression	Changes primitive operator within an infix expression
Boolean Negation	Negates a boolean expression
PAR Mutation Operators	
Replace Call Parameter	Replaces a method call parameter with a compatible one
Replace Method Call Name/Invoker	Replaces the name of a method call, or a method-invoking expression with a compatible name or expression.
Remove Condition	Remove a boolean condition in an if-condition
Add Condition	Add a boolean condition to an existing if-condition

Table 2.2: A list of the repair actions within the repair model for History Driven Program Repair, separated by their sources [Le et al., 2016].

Although those results are encouraging, the methodology used to perform the experiments makes it difficult to gauge whether the effectiveness of the technique is bolstered by its use of historical bug fix information, or whether the results are largely due to a larger repair model. Ideally, one would like to see the technique compared to a near-identical variant, in which the selection procedure is replaced by a random selection. Furthermore, the results for each technique were gathered over a single run, failing to account for their stochastic nature; consequently, the comparison may be somewhat unrepresentative of the average performance of the techniques involved.

No comparison is given between the effectiveness of using the historical bug fix database against the graphical model employed by Prophet. Although each of these techniques operate on different programming languages, their fix weighting components are both usable within C. Ideally, then, one would like to see whether the historical bug fix database performs any better than Prophet's graphical model, given the significantly higher running costs associated with its memory and CPU consumption.

2.3. Semantics-Based Repair

Instead of generating and evaluating concrete patches, semantics-based approaches use symbolic execution to determine the intended semantics of the program, before using program synthesis to construct a patch based on that extracted semantic information. By avoiding the need to compile and execute the test suite for each candidate patch, semantics-based approaches are often significantly more efficient than their search-based counterparts. This speed typically comes at the cost of restricting the set of bugs and programs to which repair might be applied, however.

In this section, we provide the reader with a brief review of several of the most popular semantics-based approaches to program repair.

MintHint

As an alternative to fully automated program repair, Kaleeswaran et al. [2014] propose a semi-automated technique, MintHint, which provides the developer with patch suggestions. By delegating the responsibility of patch validation to the developer, MintHint is able to avoid concerns of patch quality and overfitting—a major challenge for test-based repair techniques (i.e., repair techniques that rely on test suites as their oracles).

To identify candidate fix locations, MintHint uses Zoltar [Janssen et al., 2009] to compute a ranked list of suspicious statements within the program, using the Ochiai spectrum-based fault localisation metric. Like the majority of existing repair approaches, MintHint assumes that only a single statement within the program contains a fault.

For each of the top k statements within this list, MintHint generates a *state transformer* f: a function that accepts each of the program states that reach the candidate statement F and computes the correct output for each of them. To ascertain the correct output—with respect to the provided test suite—dynamic symbolic execution is performed using KLEE [Cadar et al., 2008]. To simplify hint generation, all statements in the program are converted to the form `x := e` through semantics-preserving program transformations. After applying this transformation, MintHint computes a state transformer f for each suspicious statement F as follows:

- To capture the existing, *correct* semantics of F, MintHint records the observed input/output states for each of the positive tests.
- To determine the *intended* semantics of the program over the negative test cases, MintHint first transforms the LHS variable x to a symbolic one. Using symbolic execution and constraint solving, MintHint finds concrete values of x that cause the program to yield the correct output for each of the negative tests. Finally, the program is re-run using computed values of x, in place of

Nature of hint	Targeted fault
Insert	Missing expression
Replace	Incorrect operator, constant, variable, etc.
Remove	Spurious expressions
Retain	Filtering out non-faulty statements
Compound	One of more occurrences of the above

Table 2.3: MintHint's repair model, or hints, and the kinds of faults that each hint is designed to address. Taken from [Kaleeswaran et al., 2014].

its RHS e, allowing its input/output states to be extracted, thus giving the mapping f for the failing tests.

In some cases, the symbolic execution process may fail to determine values for x, due to either time-out or an unsatisfiable constraint. If, for all of the negative tests, symbolic execution fails to generate values of x for some F within a predefined time limit, F is removed from consideration for the remainder of the repair process. Alternatively, if x is deemed to be unsatisfiable for all of the failing tests, then MintHint generates a "retain statement" hint for that statement, indicating that the statement is irrelevant to the fault and should be left unaltered.

Using the computed state transformer for a statement as its *operational specification*, MintHint exhaustively searches its *repair space* for expressions to serve as the arguments for its hints at that statement. The possible forms of these hints are listed in Table 2.3. This repair space is generated by iteratively constructing all possible expressions up to a pre-defined maximum length, using the in-scope variables at that statement, according to a provided grammar, describing the syntax of the language. Additionally, the set of sub-expressions within the RHS e are incorporated into the search space, together with the RHS itself. By incorporating e and its constituent sub-expressions, MintHint is able to determine whether those elements are likely to be faulty, independent of the results of the fault localisation. Each expression within the repair space of F is ranked according to its likelihood of occurring in the RHS of the repaired form of F. To determine this likelihood, MintHint measures the correlation between the outputs of a candidate expression and the correct values of the LHS, provided by f. Expressions that appear to be highly correlated with the values of f are assumed to indicate possible repairs, whereas lowly correlated expressions are assumed to be indicative of faulty expressions.

By incorporating e and its consituent sub-expressions into the repair space, MintHint is able to operate effectively in the presence of poor fault localisation information. Statements that are correct should exhibit a high degree of correlation between e and the expected values of x. In such cases, MintHint will generate a "Retain" hint for that statement.

After ranking the expressions within the search space of F by their likelihood of ap-

pearing in the fixed RHS, MintHint generates repair hints using these expressions, as follows:

- Exhaustively searches the space of expressions at a given statement, and filters out expressions with likelihood values below a certain threshold.
- For each candidate expression, MintHint uses pattern matching to locate matching expressions within the suspected statement. Based on the edit distance d between the candidate and matched expressions, MintHint suggests a different hint:
  - If $d \leq k$, where k is a tunable threshold (set to 2, by default), a hint is generated wherein the matched expression is replaced by the candidate expression.
  - If $d > k$, an insertion hint is generated.
  - If $d = 0$, the expression already exists within the statement,

MintHint also allows compound hints to be synthesised. Compound hints are generated by combining non-overlapping atomic hints.

To evaluate MintHint, Kaleeswaran et al. [2014] conducted a user study, in which ten participants (all of whom were either professional developers, or graduate students with prior industrial experience) were asked to debug and fix one of ten hand-seeded bugs, containing a single fault, taken from small C programs in the Software Infrastructure Repository. The time taken by the users without the tool was compared to users equipped with the tool to assess its efficacy. In an effort to reduce the difficulty of the task, the authors used Zoltar to find the five most suspicious lines within the program. The user was informed that the bug was contained on one of these lines; the lines presented were reviewed to ensure that this was the case.

Kaleeswaran et al. [2014] found that the time taken by the user to find a repair was reduced from an average of 91 minutes (with 4 participants unable to complete the task) to 29 minutes, when the tool was used (with all participants completing the task).

Although these results look encouraging, we have identified a number of potential issues in the design and evaluation of the study. No raw timing data, specifying the time taken by each user to complete the given task, is provided. Instead, this data is aggregated by computing the mean. By computing the mean, rather than the more statistically-robust median, the evaluation is prone to influence by outliers. The paper also fails to conduct statistical hypothesis testing, to demonstrate a significant difference between the control and the treatment, and to measure the effect size. The lack of statistics makes it difficult to assess the true effectiveness of the tool.

A deeper concern lies in the design of the user study itself; the ten participants in the control group were the same as those given the treatment. The order in which the subject is given the control and the treatment may produce an unintended bias

in the results. For the user study, participants were asked to fix a bug without using the tool first, before then being asked to fix a bug using the tool. The paper makes no mention of the steps, if any, to avoid introducing bias.

From a technical perspective, questions remain over whether the techniques proposed by MintHint can be scaled to larger programs. For a number of bugs within small programs taken from the Software Infrastructure Repository, containing fewer than 20 KLOC, the symbolic execution phase timed out. The costs of applying the technique to programs containing hundreds of thousands of lines of code are likely to be several orders of magnitude higher.

In summary, MintHint transforms the exceptionally difficult problem of generating high-quality repairs into the significantly easier problem of suggesting bug fixes. In practice, there is no difference between the approach of hint generation tools and repair tools—the only difference lies in the way such tools are treated. There is nothing to prevent us from using GenProg, SPR, or any other repair technique as a hint generation tool.

SemFix, DirectFix and Angelix

SemFix [Nguyen et al., 2013], DirectFix [Mechtaev et al., 2015] and Angelix [Mechtaev et al., 2016] belong to the same family of semantics-based repair techniques. All of these techniques leverage symbolic execution and component-based program synthesis to craft provably minimal repairs to faulty program expressions. Below, we summarise the abilities and limitations of these techniques:

- SemFix, the first of these techniques to be proposed, allows the repair process to scale with the size of the program, through the use of controlled symbolic execution, but is restricted to generating single-edit patches.
- DirectFix, the successor to SemFix, allows multiple-edit patches to be generated, but does so by restricting the technique to small programs. DirectFix derives its ability to craft multiple-edit patches—and its associated limitations—from describing the semantics of the *entire* program in a single logical formula.
- Angelix, the newest of the three techniques, combines the strengths of SemFix and DirectFix by using lightweight *angelic forests* to describe the semantics of the areas of the program under repair. By using angelic forests to encode the necessary semantic information, Angelix is able to produce multiple-edit patches without restriction on the size of the program—the cost of the technique scales with the size the patch, and not the program.

For the most part, the high-level approach employed by each of these techniques is split into the same four stages:

1. *Preprocessing:* A series of semantics-preserving code transformations are applied to the program under repair to allow it to address a greater number of bugs. These transformations wrap each unguarded statement in its own

`if` statement, allowing their execution to be controlled (e.g., `x = y + 1;` becomes `if (1) x = y + 1;`).

2. *Fault Localisation:* Like most search-based approaches, spectrum-based fault localisation is used to narrow down the possible locations of the faults, and to prioritise the efforts of the repair process. Fault localisation is performed at the level of expressions, rather than statements, since expressions are the target of the repair model shared by these techniques.

3. *Symbolic Execution:* Once a set of suspicious expressions has been determined, one (in the case of SemFix) or more (in case of Angelix) expressions are selected as candidates for repair. Each of the selected expressions is then replaced with symbolic variables. Symbolic execution is then used to determine whether there exists a program path through which a given test is passed; SemFix and Angelix constrain this search to consider different outcomes associated with the symbolic variables, rather than the inputs to the program. This process is carried out for each test within the suite, until either a test is failed, or all tests are passed. In the event that a set of paths are discovered that result in success for each of the tests, a constraint solver is used to infer concrete values (referred to as *angelic values*) for each of the symbolic variables. Additionally, the state of the program at each visit to these locations is recorded as the *angelic state*. For Angelix, the sets of angelic values and angelic states are combined together to form the angelic forest.

4. *Repair Synthesis:* Using the semantic information extracted during symbolic execution, a provably minimal repair to the implicated expressions is constructed using a variation of component-based program synthesis [Jha et al., 2010]. Each component is represented by a variable, constant, or a term over components and supported operations (e.g., $+$, $-$, $*$), defined in the given background theory. Through the use of soft and hard Partial MaxSMT clauses, the solver finds a patch that satisfies the synthesis specification, provided by the semantic information, and that is syntactically closest to the original program. Underlying this decision is an assumption that smaller patches, closer to the original, buggy program, are more likely to be correct, and easier to maintain, than larger patches.

SemFix and DirectFix were evaluated on artificial bugs in small programs taken from the Siemens and SIR datasets, and a small number of real-world bugs in small programs from the Coreutils [Böhme and Roychoudhury, 2014] dataset [Mechtaev et al., 2015; Nguyen et al., 2013]. SemFix was found to repair more bugs than GenProg (48 vs. 16), and to do so in an average of 3.8 minutes, compared to the average of 6 minutes required by GenProg [Nguyen et al., 2013]. DirectFix was found to repair a greater number of bugs than SemFix; 53% of those repairs were found to to be equivalent to those by the programmer, compared to 17% for SemFix. In the evaluation of DirectFix, for all scenarios longer than 135 lines of code, the fault localisation information was manually—and substantially—improved by restricting the consideration of the tool to the single function known to contain the faults.

Angelix was evaluated against GenProg, AE, and SPR on a modified sub-set of the ManyBugs dataset, containing manual improvements to some of its weak test harnesses (i.e., those that only checked the exit status of the program). Due to limits of the symbolic execution engine used by Angelix, the fbc, Python and lighttpd subjects were omitted from the evaluation. Angelix was shown to fix 28 out of 82 bugs, compared to 31, 11 and 19 achieved by SPR, GenProg and AE, respectively. Angelix was able to fix six bugs that were unsolved by other techniques, whereas SPR exclusively fixed four bugs; none of the bugs fixed by AE or GenProg were exclusive. On average, Angelix took 32 minutes to find a patch.

Although Angelix shows promise as a viable approach for constructing multiple-edit patches, it also suffers from a number of limitations, also shared by SemFix and DirectFix. Part of these limitations are inherited from the limits of current symbolic evaluation engines. Angelix is unable to generate patches involving floating point numbers, non-primitive types, or side-effects, nor is it able to introduce new statements or variables into the program.

NOPOL

Xuan et al. [2017] propose NOPOL, a semantics-based repair technique, designed to address a narrow class of defects in Java programs. This class of defects covers faulty if-conditions, and missing if-conditions around individual statements.

To fix those bugs, NOPOL uses a form of angelic debugging, similar to that performed by SemFix and SPR, to determine if there exists a set of branch outcomes for a particular if-statement that would allow the program to pass its test suite. To allow missing if-conditions to be repaired, NOPOL performs the same semantics-preserving program transformations as Angelix, wherein (unguarded) statements are wrapped in a trivial if-condition (e.g., `x = 0;` becomes `if (true) x = 0;`). Once a set of branch outcomes has been determined, NOPOL synthesises a branch condition from the in-scope variables, using its knowledge of the program state at each branch evaluation, that yields those outcomes.

Due to the nature of its implementation, the category of defects that NOPOL can actually fix is smaller than intended, however. NOPOL is unable to record information for branches that are visited more than once by the program. In contrast, SPR and SemFix make no such assumption. Moreover, NOPOL assumes that the program contains only a single faulty condition, preventing it from scaling to multiple-line faults.

SearchRepair

[Ke et al., 2015] propose an automated repair technique driven by *semantic code search* [Stolee et al., 2014]—a method for identifying code by its behaviour, rather than its syntactic features. To compose repairs, SearchRepair searches through a

large, pre-computed database of code snippets—taken from existing programs—for a single snippet which satisfies a partial specification for the program. From a high-level perspective, the approach taken by SearchRepair to construct repairs is as follows:

1. A database of code snippets is constructed from a set of donor programs. To capture its semantics, each snippet is encoded as a set of satisifiability modulo theory (SMT) constraints over its input-output behaviour.
2. Tarantula, a technique for spectrum-based fault localisation, used by a majority of repair approaches, is used to rank the suspiciousness of program *fragments*, rather than lines or statements.
3. For each suspicious fragment, a lightweight profile of its desired input-output behaviour—determined using symbolic execution [Cadar et al., 2008] over its test suite—is generated, in the form of a set of SMT constraints.
4. Using the Z3 SMT constraint solver [de Moura and Bjørner, 2008], the database of donor code is searched for snippets which satisfy the desired behaviour of the faulty program fragment. Matching snippets are thereafter contextualised, through a process of variable substitution, and transformed into a patch, wherein the original fragment is replaced by the discovered snippet.
5. Finally, as with search-based methods, candidate patches are evaluated against the test suite to determine whether they represent a repair.

SearchRepair sits between search-based techniques, which randomly generate mutants and assess their correctness by evaluation against a test suite, and specification-based techniques, which use program synthesis to generate a repair that satisfies a partial specification.[4]

Like GenProg and its variants, SearchRepair relies on the plastic-surgery hypothesis to find redundant code within programs from which to craft its repairs. Whereas GenProg operates on the level of statements, SearchRepair operates on more-coarsely-grained entities, known as fragments. Fragments include the basic blocks within if- and while-statements, the if- and while-statements themselves, and sequences of one to five statements. By operating at a higher level of granularity, Ke et al. [2015] believe that SearchRepair is less susceptible to overfitting whilst remaining sufficiently expressive. Ke et al. [2015] state:

> Our core assumption is that a larger block of human-written code, such as a method body, that fits a given partial specification is more likely to satisfy the unwritten specification than a randomly chosen set of edits generated with respect to the same partial specification.

Although results from the paper show that 97.3% of patches generated by SearchRepair over the IntroClass benchmarks are correct—with respect to an independent test

[4]Note that patches generated by specification-based techniques are only guaranteed to be correct with respect to a partial specification, and not necessarily to the complete, intended specification of the program. Therefore, such methods are equally as susceptible to overfitting as search-based techniques.

suite—compared to 68.7% for GenProg, Ke et al. [2015] do not explicitly validate this hypothesis, and these results alone do not confirm it. Whilst almost all patches generated by SearchRepair were deemed correct, SearchRepair also fixed the fewest bugs, with respect to the original test suite.

Additionally, the benchmarks used consist of bugs in small programs, each containing fewer than 100 lines of code, and so questions remain over the generality of the technique, and its effectiveness when applied to bugs in large-scale programs, such as those in the ManyBugs dataset. Moreover, due to the limitations of KLEE—the symbolic execution engine used to compute profiles—SearchRepair is unable to insert code containing loops, recursion, non-primitive types, side-effects, or I/O operations (e.g., file manipulation, database queries).

Despite suffering from a number of limitations, SearchRepair presents a promising approach to automated program repair by combining the strengths of search-based methods with semantics-based techniques.

2.4. Specification-Based Repair

Unlike search-based and semantics-based program repair, both of which rely on user-provided test suites to establish correctness, *specification-based* repair techniques assume the existence of formal specifications. Due to this requirement, the applicability of these approaches is far more limited; none of the datasets previously used to evaluated search-based or semantics-based repair approaches (e.g., SIR, ManyBugs, Coreutils) possess any such specifications. On the other hand, *specification-based* approaches are able to leverage these specifications to generate higher quality patches (that are likely to satisfy them). Concerns are shifted from the adequacy of the test suite, to the soundness and completeness of the specifications that are used. Like semantics-based repair, specification-based approaches use program synthesis techniques to generate patches.

Below, we briefly discuss AutoFix-E, one of the few and foremost approaches to *specification-based* program repair.

AutoFix-E

AutoFix-E [Wei et al., 2010] leverages partial specifications, provided in the form of contracts, to automatically repair faults in Eiffel classes. These contracts are used to specify preconditions, postconditions and intermediary assertions over Eiffel classes.

Provided a defective Eiffel class, AutoFix-E attempts to expose and repair the underlying bug, following the steps below:

1. *Test Suite Generation:* In contrast to almost all search-based and semantics-based repair approaches, AutoFix-E generates a test suite automatically, rather than relying on one provided by the programmer. AutoTest, an automated test generation tool for Eiffel, is used to generate as large a test suite as possible within a fixed window of time, covering the defective class.

   By evaluating the program under repair against this test suite, the tests are partitioned into *passing runs* and *failing runs* (equivalent to *positive* and *negative* tests). A run is determined to be a failure if it results in a contract violation.

2. *Object State:* To aid in determining the source of the fault, AutoFix-E examines object state by observing the output of argument-less, boolean-valued functions (referred to as boolean queries). Boolean queries are widely used in Eiffel contracts as a means of concisely capturing the key properties of object state. Together, the n boolean queries for a given class describe its 2^n abstract states. Although the resulting state space may seem intractable, a study of a large Eiffel library, conducted by Wei et al. [2010], suggests that the majority of Eiffel classes have 15 or fewer such queries.

   Using the set of boolean queries for the class under repair, AutoFix-E uses contract mining to discover implications between queries. For each mined implication, AutoFix-E also generates three mutants of that implication: the negation of its antecendent, consequent, and both. Together, the sets of boolean queries and implications form the *predicate set* P; this set describes a set of simple facts about the class under repair, encoded as logical statements. To improve the efficiency of later stages in the repair process, the predicate set is pruned, with the help of an automated theorem prover.

3. *Fault Profiling:* Using Daikon, a popular invariant mining tool, and a superset Π of the predicate set P, also containing the negation of each predicate, AutoFix-E generates a pair of invariants for each executed program location ℓ. This pair, (I_ℓ^+, I_ℓ^-), is comprised of the predicates $I_\ell^+ \subseteq \Pi$ that hold for all passing runs, and the predicates $I_\ell^- \subseteq \Pi$ that hold for all failing runs. Together, this sequence of pairs forms a *fault profile*, describing the possible causes for the failed runs, in terms of abstract state.

4. *Behaviour Model Generation:* To serve as a source of ingredients for its repair process, AutoFix-E mines a simple *finite-state behavioural model* over all of its passing runs, encoded as a finite-state automaton. The states of this automation are labelled by the predicates that hold over that state. Transitions are labelled with the name of a routine, and are used to describe a particular input-output behaviour of a routine, in terms of abstract state; implicitly, each transition describes a Hoare triple.

   The resulting model is used by AutoFix-E to determine how to reach a desired state (i.e., the intended state) from some current state (i.e., a faulty state). In general, the model is neither sound nor complete, since it is inferred over a

(a)

```
snippet
old_stmt
```

(b)

```
if fail then
  snippet
end
old_stmt
```

(c)

```
if not fail then
  old_stmt
end
```

(d)

```
if fail then
  snippet
else
  old_stmt
end
```

Figure 2.9: The fix schemas implemented in AutoFix-E [Wei et al., 2010]. snippet is replaced with a sequence of routine calls that move the program from a faulty state into a desired state. old_stmt may either be a single statement, or the block to which a statement belongs. fail is used to monitor the conditions under which the fault manifests, and not to affect the appropriate action.

finite set of executions. In practice, the model appears to be sufficiently precise for the purposes of finding a repair.

5. *Fix Generation:* Beginning at the location where the fault occurred and iterating backwards, AutoFix-E uses its fault profile and behaviour model to generate a set of candidate repairs. At each location, AutoFix-E finds all possible instantiations of the four repair schemas within its model, outlined in Figure 2.9. Each of these candidate repairs uses the fault profile to move the program from a possibly faulty state to a possibly correct state. To achieve this change in state, AutoFix-E uses its behavioural model to finds the sequence of routine calls that results in the desired post-condition.

   A special fix schema is also introduced for repairing linear assertion violations (e.g., $count \geq 0$).

6. *Fix Validation:* Once a set of candidate repairs has been generated, AutoFix-E evaluates them, to determine the set of valid repairs (i.e., those which pass all of the tests). Finally, AutoFix-E uses a series of metrics to rank the valid repairs according to some proxy to quality. These metrics measure the size of the snippet, the distance in state, the number of old statements captured by the fix schema, and the number of branches required to reach old_stmt from the point of injection of the instantiated fix schema.

To evaluate AutoFix-E, Wei et al. [2010] ran it on a dataset of 42 faults detected by AutoTest in 10 data structure classes taken from popular, open-source Eiffel libraries. AutoFix-E was able to repair 16 of these faults. To assess the quality of AutoFix-E's patches, they manually inspected the top five patches reported for each bug to determine if the patch fixed the underlying bug without introducing a new defect. 13 of the 16 bugs fixed by AutoFix-E were found to satisfy this criterion.

2.5. Related Techniques

To conclude our review of the relevant literature, in this section we briefly discuss a number of techniques loosely related to general-purpose program repair:

- **Reflective Grammatical Evolution:** [Timperley, 2013; Timperley and Stepney, 2014] incorporate concepts from failure-obliviousness into genetic programming. After using computational reflection to create a failure-oblivious dialect of Ruby, the authors demonstrate a higher success rate and efficiency—measured by candidate evaluations—when programs are evolved with these measures enabled. These results suggest that allowing the program to persist in the presence of errors, through the use of such measures, may smoothen the fitness landscape and reduce the difficulty of the search.

- **Data Structure Repair:** [Demsky and Rinard, 2003, 2005] present techniques for automatically recovering from data structure corruption errors, at runtime. Repair is achieved by enforcing a data structure consistency specification, which may be manually provided by the developer, or automatically inferred from correct program executions, using tools such as Daikon.

- **Integer Bug Fixing:** [Coker and Hafiz, 2013] propose a set of three program transformations for repairing all instances of integer bug within C programs:

  1. *Add Integer Cast:* adds explicit casts to disambiguate integer usage, and to address signedness and widthness problems.

  2. *Replace Arithmetic Operator:* replaces arithmetic operations with safe equivalents, which detect overflows and underflows at runtime.

  3. *Change Integer Type:* modifies types of integers to avoid signedness and widthness problems.

  Together, these three transformations fixed all 7,147 programs within NIST's SAMATE dataset [Black, 2007], covering over 15 million lines of code. Unlike general-purpose repair techniques, Coker and Hafiz's program transformations produce sound and complete repairs for a restricted class of defects, forgoing the need for test suite evalution or symbolic execution. Although these transformations prevent integer bugs from manifesting, they do so by ensuring safe behaviour, rather than finding particular unsafe instances of integer usage and patching the source code directly, thus avoiding the need for potentially expensive instrumentation.

- **Bolt and Jolt:** Bolt [Kling et al., 2012] and Jolt [Carbin et al., 2011] are techniques for monitoring the execution of a program, detecting whether an infinite loop occurs, and if so, allowing the loop to be executed or the program to be terminated, at the request of the user. Whereas Jolt requires source code instrumentation to inject monitoring code, Bolt obviates this need through on-demand dynamic binary instrumentation; the (unstripped) binaries remain

unmodified until the user attaches Bolt to the application.

To detect the occurrence of (some) infinite loops, both techniques monitor the state of the program upon entry to the loop. If upon the next iteration the current state is the same as the previous state, an infinite loop is detected and the user is given the option to escape the loop or to terminate the program.

Across eight infinite loops in five small but real programs (grep, ctags, indent, ping, look), Jolt was able to detect an infinite loop in seven cases [Carbin et al., 2011]. Importantly, in each case, the program was allowed to continue executing, producing a more useful output than simply terminating the program [Carbin et al., 2011; Kling et al., 2012].

- **CodePhage:** Instead of addressing bugs via source code modification, CodePhage [Sidiroglou-Douskos et al., 2015] attempts to fix bugs by automatically identifying correct code in foreign, donor applications, and transferring that code into the faulty program. CodePhage allows programs to be repaired without the source code of either the program under repair or the source code of the donor applications, by operating at the binary level. In its evaluation, CodePhage was able to successfully repair five out of ten security bugs across seven different programs.

- **ClearView:** ClearView [Perkins et al., 2009] uses Daikon to infer likely invariants for a given system, based on data collected from several training executions. Prior to deployment, ClearView uses binary instrumentation to inject monitoring code into the program. Two of these monitors, HeapGuard and Determina Memory Firewall, check for out-of-bounds memory accesses, and illegal control flow transfer errors, respectively. A third monitor, Shadow Stack, allows invariant violations along the call stack to be recorded. In the event that a monitor reports erroneous behaviour at run-time, ClearView attempts to generate a patch that restores violated invariants and satisfies the monitor. The generated patches re-establish the inferred invariants by altering control flow, register values, and/or values of memory locations.

  To evaluate ClearView, Perkins et al. [2009] conducted a Red Team exercise, wherein members of the Red Team were asked to perform attacks on a program protected by ClearView, using exploits discovered on the unprotected version of the program. Firefox, a popular open-source web browser, was used as the target application for the exercise. The external Red Team was able to generate ten code-injection exploits in the target application. When ClearView was used, all of these attacks were detected by its monitors, and thus prevented. In seven out of ten cases, ClearView generated a patch that allowed the program to survive the attack and to safely resume its execution.

- **LeakFix:** LeakFix [Gao et al., 2015] uses a series of program analyses to locate and repair (a sub-set of) memory leaks in C programs. Identified leaks are patched by inserting deallocation statements at appropriate points in the program. The resulting patches are guaranteed to not interrupt normal pro-

gram execution. On an evaluation of 15 programs, comprising 522 KLOC, each containing multiple leaks, LeakFix generated patches for 28% of the leaks. LeakFix exemplifies an alternative approach to automated program repair: tackling specific defect classes with high quality and accuracy. This approach is an appealing one, but in practice, it is difficult to cleanly assign bugs to any one particular defect class.

- **Genetic Improvement:** Search-based program repair can be viewed as part of the wider field of *Genetic Improvement* (GI) [Langdon, 2015], which seeks to apply machine learning and search techniques to improving existing programs, more generally. In contrast to Genetic Programming, which typically attempts to evolve a program from scratch, GI uses existing code as its seed. In addition to program repair, GI has been used to automatically improve runtime performance, reduce power consumption, port functionality, and more.

2.6. Concluding Remarks

In this chapter, we conducted a review of the majority of existing program repair techniques. Below, we summarise each of the three main approaches to repair:

- Semantics-based approaches, such as Angelix [Mechtaev et al., 2016], have been shown to be more efficient than search-based approaches and to successfully produce multi-line patches. The bugs and programs to which semantics-based repair can be applied is limited, however. Due to limitations in its underlying techniques, namely those inherited from symbolic execution, semantics-based approaches are unable to repair bugs involving side-effects, iteration, recursion, non-primitive data types, and more.

- Specification-based repair has demonstrated promising results, specifically AutoFix-E [Wei et al., 2010], but its reliance on formal specifications prevents it from being applied to the majority of bugs. Nonetheless, it may be possible to use contracts to *guide* search-based repair towards a repair, rather than to require them. The information provided by contracts could be integrated into a richer fitness function, similar to the one introduced by Arcuri and Yao [2008].

  Alternatively, specification mining [Ernst et al., 2007] may be used to automatically infer partial specifications. Fast et al. [2010] attempt to realise this with their predicate-based fitness function (described in Section 6.1. However, their approach requires knowledge of at least one patch *a priori*, preventing it from being used in practice. B. Le et al. [2016] use mined specifications to improve the accuracy of function-level fault localisation (see Section 4.1.4 for more details). Additionally, it may be possible to use the degree to which

a potential solution conforms to an inferred specification as a proxy to the quality of a solution (i.e., whether the solution destroys existing functionality or overfits to the test suite). To our knowledge, no studies have explored a possible link between specification conformance and patch quality.

- Search-based repair is the only approach that neither requires formal specifications nor has any limitations on the bugs it can fix. GenProg, the seminal approach to search-based repair, is one of few search-based techniques capable of generating multiple-edit patches—PAR and HDRepair, the other approaches capable of multiple-edit repair, share the same underlying genetic algorithm. Despite this capability, almost all patches that were generated by GenProg in previous studies were reduced to a single edit [Le Goues et al., 2012a; Qi et al., 2015]. Techniques such as PAR, AE, and SPR sacrifice the ability to generate multiple-edit patches in exchange for significant efficiency gains. It is unlikely that the algorithms used by these techniques (random search, exhaustive search and value search) can be scaled to generating multiple-edit patches, owing to the combinatorial explosion of the search space. To tackle the problem of multiple-edit repair, new *active* search algorithms are needed, capable of identifying and exploiting partial fixes, and localising bugs over the course of the search.

CHAPTER 3

Tools and Techniques

Research in automatic program repair is typically evaluated by applying a new technique to a set of real-world bugs, and comparing the results to those produced by prior techniques. Such an evaluation ideally involves a set of reproducible real-world defects. Open-source software repositories provide a plentiful source of such bugs, but accurately reproducing them reliably can prove challenging. Program behaviour is often dependent upon the particular configuration of its host's environment, such as the versions of particular libraries; many bugs only manifest under certain configurations. Additionally, because most program repair techniques involve evaluating candidate patches that may apply arbitrary changes to the program under repair, experiments must be appropriately sandboxed to ensure system safety and test idempotency.

What is needed, therefore, is both a dataset of bugs, and a platform for interacting with them that ensures reproducibility and safety. The former is provided by benchmarks such as ManyBugs [Le Goues et al., 2015], Defects4J [Just et al., 2014] and the Software Infrastructure Repository (SIR) [Do et al., 2005]. For Java bugs, Defects4J provides both a dataset of bugs, and a platform for executing them, implicitly supplied by the Java Virtual Machine. For C bugs, this platform is either provided in the form of a monolithic virtual machine (VM) [Le Goues et al., 2015; Long and Rinard, 2015; Qi et al., 2015], shared by multiple bugs within the same dataset, or, in some cases, not provided at all [Kim et al., 2013].

Although the use of a VM as an execution platform provides reproducibility and safety, it does so at the cost of performance and extensibility. Running the experiment within a VM incurs the significant performance penalties associated with the overheads of virtualisation. Using a single VM to host both the bug scenario and the repair tool limits its future usage. Newer techniques may be unusable within the VM due to library incompatibilities, leading to the need to perform unsound modifications to VM, which compromises its reproducibility.

Motivated by these problems and the issues of repair quality, we present RepairBox, a platform for safely conducting controlled, reproducible program repair experiments, with minimal compromise to performance. RepairBox operates by isolating individual bug scenarios and repair tools into separate, minimal Docker[1] containers that can be easily inspected and extended. By packaging bug scenarios as containers, RepairBox attains close to bare metal performance, an important trait when performing a large number of repeats of computationally-intensive program repair experiments.

[1] https://www.docker.com. [Accessed April, 2017].

A summary of the contributions of this chapter is given below:

- Identified undiscovered weaknesses in a number of publicly available APR benchmarks previously used to evaluate the performance of existing techniques.
- Developed a tool, Pythia, to automatically improve the reliability of existing test suites, within the context of APR; ensures all passing tests meet a minimal set of quality requirements.
- Proposed a novel, efficient technique, RepairBox, for conducting controlled experiments into APR, together with a set of accompanying open-source tools.
- Collated and edited over 400 bug scenarios for C programs from a number of existing sources, including ManyBugs, GenProg's TSE benchmarks, and the Software Infrastructure Repository.

The rest of the chapter is structured as follows: In Section 3.1, we review a collection of publicly available datasets of bugs, used to conduct APR experiments. In Section 3.2, we discuss the problem of repair quality and its importance to empirical studies of APR, before presenting Pythia, a tool to avoid this problem. In Section 3.3, we review existing approaches to reproducibility within APR before proposing RepairBox, a high-performance, controlled platform for conducting APR experiments. Finally, in Section 3.4, we put forward an efficient and robust methodology for conducting APR experiments, based on these tools, which is used for the remainder of this thesis.

3.1. Bug Scenarios

In this section, we first identify and review a number of publicly-available datasets of bug scenarios, used to conduct empirical studies of program repair. We then discuss some of the current challenges involved in performing such studies.

- **GenProg ICSE 2009, GECCO 2010, TSE 2012:** each of these sets of bug scenarios was used to conduct early experiments on GenProg [Fast et al., 2010; Le Goues et al., 2012b; Weimer et al., 2009]. Across these three sets, 20 bugs, each from different programs, are covered. Of these 20 bugs, two are found in artificial programs, consisting of fewer than 100 lines of code (`gcd`, `zune`). 17 of the remaining 18 bugs are sourced from genuine bugs in large, open-source projects (many taken from the results of Miller's work on fuzz testing [Miller et al., 1990], with the other bug (`atris`) being sourced from a small game created by one of the authors of the paper.

  Although the majority of these scenarios represent genuine bugs in large projects, each uses a very small, artificial test suite, hand-written by the authors. Each of these test suites contains fewer than 21 tests, designed to solely

expose the bug, rather than thoroughly test the functionality of the program. This behaviour allows destructive changes to be silently introduced into the program. Whilst the test suites for these bugs are unrepresentative of standard test suites, their small sizes, together with the relatively short compilation times for most of their programs,[2] makes them ideal for high-cost, non-deterministic experiments, where large numbers of repeats are required to perform statistical hypothesis testing (e.g., to demonstrate a difference in performance between two search algorithms).

- **ManyBugs:** to validate the effectiveness of GenProg on a set of bug scenarios more representative of a real-world environment, the ManyBugs [Le Goues et al., 2015] benchmarks contain 185 bug scenarios across 9 large, open-source programs, where each employs a genuine test suite. Since real test suites are provided with each bug, these benchmarks are appropriate for gauging and comparing the overall effectiveness of APR approaches in the wild.

  Since real test suites are used in each of these bug scenarios, they tend to be several orders of magnitude larger than the artificial test suites found in the earlier GenProg datasets. For instance, each of the PHP bug scenarios contains around 8000 tests. As such, performing multiple runs of a given algorithm on one of these scenarios to achieve statistical significance can become particularly expensive, requiring (tens of) thousands of CPU hours.

- **IntroClass:** Whilst ManyBugs and the earlier GenProg benchmarks are almost exclusively made up of large, open-source projects, consisting of millions of lines of code, the IntroClass [Le Goues et al., 2015] bug scenarios provide an invaluable set of real-world bugs from small programs, consisting of fewer than 100 lines of code, taken from students' laboratory assignments. In total, this dataset contains 998 bugs. Each bug is accompanied by a pair of independent test suites containing 95 test cases in total, one of which the user could use during the development of their solutions; the other may be used to validate solutions, and to reduce the likelihood of overfitting. Additionally, an oracle program is provided for each of the six assignments.

  The small-size and low complexity of this set of benchmarks make it particularly well suited to studies involving a large number of bug scenarios, or those in which certain levels of statistical significance must be achieved. Additionally, these bugs provide insight into the ability of APR techniques to generalise to smaller programs, written by novice developers. This particular trait may also be useful in assessing the effectiveness of *plastic surgery*-driven approaches for small code bases.

- **Defects4J:** Defects4J [Just et al., 2014] provides a database of high quality Java bug scenarios, designed for studies on software quality and testing. This database covers 395 real-world bugs across 6 open-source projects.

[2] For `php` and `imagemagick`, using the provided compilation scripts could result in compilation times taking as long as 8 minutes.

Another source of bugs within C programs, albeit one that has yet to be used by the APR community, is provided by Software Infrastructure Repository [Do et al., 2005]. SIR contains a collection of thousands of manually seeded bugs across 85 small to medium-sized programs (1 to 100 KLOC), written in C, C++, C#, and Java. Each program comes with a series of manually engineered test suites, designed to test the program at various levels of coverage.

Although artificial in the nature of its tests and bugs, the SIR offers a rich and plentiful source of bugs, suitable for conducting studies on large numbers of medium-sized programs.

The Siemens benchmarks [Hutchins et al., 1994] also provide a source of synthetic bugs within C programs. In comparison to the bugs within the SIR, the Siemens bugs span between 100 and 800 lines of code. Each bug scenario is accompanied by a test suite containing thousands of tests, providing a level of coverage rarely seen in real-world programs.

Importantly, while these datasets provide a plentiful source of bug scenarios, none provides an execution platform in which to reproduce these bugs, with the exception of Defects4J; Defects4J implicitly provides such a platform in the form of the Java Virtual Machine. For programs written in compiled languages such as C, this lack remains an important and unsolved problem.

3.2. Pythia

In this section, we look at the problems that inadequate test suites cause when performing APR experiments, before introducing a lightweight, open-source tool to address the problem, and to improve our confidence in the results of APR experiments.

Motivation

Recent findings [Qi et al., 2015; Smith et al., 2015] have demonstrated how inadequate test suites can lead to (unintentionally) misleading results and claims regarding the effectiveness of repair techniques. The majority of test suites provided within publicly-available automated repair benchmarks (for C) have been shown to suffer from serious problems of inadequacy.

Qi et al. [2015] found that previously published results for GenProg and AE on the ManyBugs suite, wherein 55 of 105 bugs are solved, are mostly the result of overfitting. Following a manual inspection of the patches produced by GenProg and AE, they observe that only 2 of the 55 solved bugs yielded correct patches. Upon closer examination of the ManyBugs test suite, they discovered that many of the tests were failing to compare the output of the program against its expected

output. In some cases, the only requirement of the test was that the program should exit with a status of zero, allowing trivial patches such as `exit(0);` to be quickly discovered and accepted.

Motivated by those findings, we subjected the ManyBugs scenarios to further scrutiny, discovering a different class of weakness within certain test suites in the process. Specifically, for programs such as `libtiff` and `gzip`, the accompanying test harnesses neither check for expected side effects on the filesystem, such as the presence (or absence) of a particular file (e.g., `file` should be replaced by `file.z`), nor do they check the state of particular files. We also examined the GenProg TSE 2012 bug scenarios, and found that the majority of tests also failed to validate on any other criteria than the exit status of the program. After manually modifying each of the test suites to compare their outputs against an expected output, GenProg (and AE) failed to yield a solution, where previously they did. [3]

Such weaknesses compromise the validity of the conclusions that are drawn during APR experiments, and thus make performing experiments exceptionally challenging with such test suites. To reason about the effectiveness of APR approaches—and avoid potentially misleading results—one must introduce further rigour to the test suites that are used. No existing (publicly available) benchmarks for C programs exist that are free from such weaknesses.

Existing Solutions

To tackle this problem, there are a number of existing techniques which one may use. We critique each of these techniques below.

Manual Extension and Modification

The simplest solution to the problem is to manually add more test cases to each of the test suites that are used within the experiment. However, doing so risks losing the real-world relevance of the results as the test suites become more artificial and less realistic.

One may argue that the user of an automated repair technique (in practice) could be expected to supply these additional test cases. In cases where bugs are less trivial, and the user is unable to fully comprehend them, such an assumption may prove impractical. Furthermore, such an approach would require a significant investment of time to carefully craft a higher-quality test suite for each problem.

Similarly, one may attempt to address the problem at its root by manually addressing each of the weaknesses in the test suites, thereby reducing the number of false positive results. By taking such measures, however, one's results become predicated on

[3] For some bug scenarios, this was not possible since the program failed to produce any output when run on a 64-bit variant of Fedora 24. In these cases, the program appeared to have no side-effects on either the file system, the standard output, or the standard error.

the assumption that the test suite is of a high quality—an assumption that often fails to hold in the real-world, as exemplified by the weaknesses within the ManyBugs benchmarks (e.g., PHP, Libtiff, gzip). As with manually extending the test suite, this approach also requires a significant investment of time as (thousands of) existing test suites need to be analysed and adjusted.

Automatic Extension

Alternatively, one may use the developer-provided patch for the bug as an oracle, allowing automated test generation [Cadar et al., 2008] to be used to supplement the test suite with more positive and negative tests. As a simple, albeit limited criterion for the adequacy of a repair, one may use test case generation to increase statement coverage.

In practice, aiming for complete branch, statement, or MC/DC coverage across the test suite is likely to prove intractable for all but the smallest and most artificial of problems. Even if one were to relax the coverage criterion to full coverage at the statement level (within a particular set of files), the resulting test suite may still prove too large as the size of the program grows. However, one may use test suite prioritisation and sampling to reduce the cost of this extended test suite.

For the purposes of assessing the potential effectiveness of automated repair techniques within the real-world, where complete coverage (at any level) is scarce, augmenting the test suite in any way may produce overly optimistic results. In the general case, we believe that ideal, representative bug scenarios used within APR experiments should place no assumptions on the user extending the original test suite of the program, whether manually or automatically.

Pythia

To address the problem of inadequate test suites, without the need for manual intervention, we introduce an open-source, automated tool for improving the quality of experiments, Pythia.[4] Pythia takes a test suite manifest file (provided in a human-readable JSON format), together with an oracle program (i.e., the human-repaired form of the program), and generates an oracle file, describing the expected behaviour of the program for each test within the manifest. This process is illustrated in Figure 3.1. Pythia may be used to improve the detection and automated repair of regression faults in real-world programs by using an existing version of the program.

Once the oracle file has been generated, all tests (when performed using Pythia) are guaranteed to meet three quality requirements:

[4]The name "Pythia" comes from the name of the mythical Oracle at Delphi (https://en.wikipedia.org/wiki/Pythia). Pythia is available to download at: https://github.com/ChrisTimperley/Pythia.

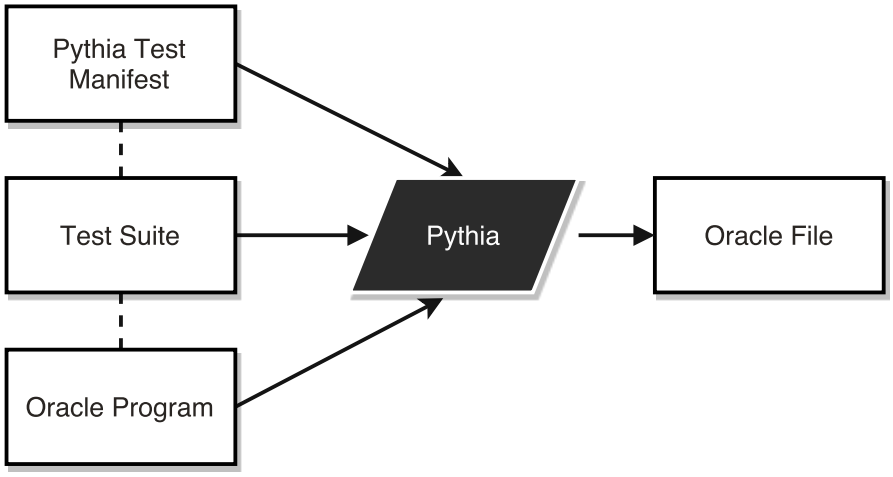

Figure 3.1: An overview of the inputs and outputs of PYTHIA's oracle generation process.

1. *Checking of standard output:* the standard output of the program should match that of the oracle.
2. *Checking of standard error:* the standard error of the program should match that of the oracle.
3. *Checking of exit code:* the program should exit with the same code as the oracle.
4. *Checking of file system:* the program should leave the relevant areas of the file system in the same state as the oracle (i.e., the content, presence, and absence of files are checked).

Rather than describing tests and their expected outputs within the same file, as is the typical approach within the ManyBugs and early GENPROG benchmark sets, PYTHIA uses the description of a test to create a (faux) isolated environment for its execution (known as a sandbox), before executing its associated shell command within that sandbox and recording its various behaviours. After capturing the behaviour of a test execution, those behaviours are compared against those given in the oracle file—if the program exhibits all of the same behaviours, it passes the test; otherwise, it fails.

Sandbox Environment

Prior to the execution of each test, an isolated environment, or *sandbox*, is created for the execution. By reading the description of a test from its corresponding manifest file, PYTHIA copies the files relevant to this test into the sandbox, ensuring a minimal, working environment for the execution. In practice, sandboxes are implemented as temporary directories that are disposed of after the execution.

```
$ ./gzip ../inputs/testdir/file26 -c
```

Figure 3.2: In this example, taken from the `gzip` object within the SIR, the test suite attempts to destructively compress a given file, deleting the original version of the file and replacing it with its zipped form.

```
[
  ...,
  {
    "description": "Compresses a file at a given location",
    "command": "<<PROGRAM>> -c '<<SANDBOX>>/foo'",
    "sandbox": {
      "foo": "inputs/example.txt",
      "bar": "inputs/example2.txt"
    }
  },
  ...
]
```

Figure 3.3: An example test case description within a Pythia test manifest file. Each test case is described by its corresponding shell command, the contents of its sandbox directory, and an optional human-readable description.

Pythia includes this feature to avoid the side-effects of certain test executions on the file system. One example of where this behaviour is essential, amongst several, can be found in the `gzip` object from the Software Infrastructure Repository. An exemplar test within the original test suite is given in Figure 3.2. When one attempts to run the test a second time, the oracle program will produce a different result, as the input file to the test was destroyed during the initial invocation. By copying the necessary files to the sandbox, rather than using their original versions on the file system, such side-effects can be avoided.

Test Manifest File Format

Test manifest files within Pythia are described using a JSON array of objects, where each object describes a single test case by a shell command, the contents of its sandbox, and an optional human-readable description of the test case (useful for debugging). An example test case description within a manifest file is given in Figure 3.3.

To ensure the program behaves appropriately in all environments, two special variables may be used in the shell command for the test case: `<<PROGRAM>>`, which specifies the absolute path to the program executable, and `<<SANDBOX>>`, which gives the absolute path of the sandbox for the particular execution. Prior to execution, Pythia substitutes these special variables, replacing them with their appropriate

```
[
  ...,
  {
    "retval": 0,
    "time": 1.432,
    "out": "zipped f to f.z",
    "err": "",
    "sandbox": {
      "f.z": "cf23df2207d99a74fbe169e3eba035e633b65d94"
    }
  },
  ...
]
```

Figure 3.4: An entry in an example Pythia oracle file, describing the expected behaviour of a test case from the corresponding manifest file.

values.

The contents of the sandbox for each test case are described using a dictionary, where each entry describes a file that should be copied into the sandbox. The value (i.e., right-hand side) of each entry specifies the location of the file that should be copied to the sandbox (the path must be either absolute, or relative to the location of the manifest file). The key for each entry (i.e., left-hand side), specifies the path that the file should be copied to, relative to the root of the sandbox directory. Note that no distinction is made between files and directories when copying; when a directory is specified by the value of a sandbox entry, that directory will be copied recursively to a directory with the name given by the key of that entry.

Oracle File Format

Pythia uses oracle files to describe the expected behaviour of each test within a corresponding test suite. As with the test manifest, oracle files are encoded as a JSON array of objects, where the *nth* object describes the intended behaviour for the *nth* entry within its associated manifest. Each object specifies the expected exit code, standard output, standard error, the state of the sandbox after execution, and the time taken to reach completion.

An example oracle file entry is given in Figure 3.4. The `retval`, `out` and `err` properties record the exit code and the state of the standard output and standard error respectively. The resulting state of the sandbox is stored by computing a SHA1 hash of each file therein, recursively. The contents of each file may be just as effectively described using a hash, rather than its raw contents, since the state is only used for comparison. By recording the state in such a manner, Pythia is able to check for the presence and absence of files within the sandbox, and to ensure the correctness

of their contents.

The `time` property specifies the number of seconds taken for the test to finish its execution. PYTHIA uses this information to automatically set appropriate time limits for each test. Specifically, PYTHIA scales this value using the formula given in Equation 3.1:

$$t' = P \cdot G \cdot t + \Delta \tag{3.1}$$

where t' is the calculated time limit for a given test, t is the time taken by the oracle to finish that test, Δ is an offset, and P and G are platform and generic scaling factors. Δ and G are used to control for variance in execution time. P is used to take the expected execution time for one machine, and to apply it to another; this term is used when the oracle is built on a faster machine than the one used to conduct repair.

By assigning individual time limits to each test, rather than a single time limit for all tests—as is the current standard—PYTHIA significantly reduces the time taken to evaluate the entire test suite for worst-case repairs.

Limitations

Although PYTHIA addresses certain kinds of test suite weaknesses, allowing practioners to gain a higher degree of confidence in the results of their experiments, it is not without its limitations and trade-offs:

- **Non-determinism:** in cases where some aspect of the behaviour of a particular test case is non-deterministic, correct results may fail to agree with the oracle, resulting in false negatives. To our knowledge, this problem does not affect any of the benchmarks within the benchmark sets discussed earlier; the SIR benchmarks, in particular, specifically go about removing sources of non-determinism from the program.
- **Inadequate test coverage:** although PYTHIA avoids certain, common degenerate solutions from being accepted (e.g., early program exits) it is unable to prevent false positives stemming from a lack of coverage (e.g., a faulty `If` condition that is only covered once may be replaced by a tautology or a fallacy).
- **Inefficient copying of files:** in cases where the program does not write to its input files, copying them to the sandbox is a waste of resources. However, in the general case, variants of the program may have the ability to write to that file, and so to avoid side-effects, a copy of that file must be made. Alternatively, one could solve the problem by ensuring the file is read-only and creating a symbolic link within the sandbox.
- **Ignorance of file attributes:** properties of files, such as their permissions, ownership, and associated dates are not recorded by PYTHIA. In general, we

observe few instances where such attributes are relevant to testing. Moreover, adding certain of these properties may introduce non-determinism.

- **Modification of files outside of sandbox:** certain tests may involve files outside of the sandbox (e.g., `htpasswd`). To protect these files from being infected and left in an altered state following a test execution, one must take further steps to isolate the test execution. A particularly expensive option would be to launch a separate container (discussed further in Section 3.3) for each execution. An alternative solution—albeit complex—would be to (optionally) use the sandbox as a `chroot` jail for each test execution, guaranteeing a side-effect free execution and a complete description of the state of the file system.

Despite these drawbacks, PYTHIA provides a lightweight way of improving the quality of existing test suites and makes performing experiments easier and their results more robust. Future work may address some of these issues to further enhance the utility of PYTHIA.

3.3. RepairBox

Having studied and addressed the problem of inadequate test suites within APR experimentation, in this section we look at the related problem of reproducibility of results, before introducing a framework to eradicate the problem with minimal loss of efficiency.

Motivation

In addition to exposing weaknesses within publicly available bug scenarios, recent studies have also highlighted the difficulties in accurately reproducing the results of previous experiments. Whilst investigating the false positive repairs generated by AE and GENPROG on the ManyBugs bug scenarios, [Qi et al., 2015] were unable to reproduce the original (plausible, but incorrect) repairs generated by GENPROG and AE in some instances. With no information about the environment of the experiment beyond a (partially configured) virtual machine, the configuration required to produce these original results is difficult to replicate.

Through personal experience with the ManyBugs bug scenarios, we have encountered difficulties in installing newer software on the virtual machine images provided by previous experiments. Given the lack of any additional documentation with these VM images, creating a new, minimal environment that replicates the bug can prove challenging.

In other work, where template-driven automated program repair was applied to Java programs [Kim et al., 2013], none of the source code used within the experiment was

made available; only informal, ambiguous descriptions of the system were given, making it impossible to reproduce the results of the original system.

Existing Solutions

In this section, existing approaches to allowing the results of APR experiments to be reproduced are critiqued.

Source Files

The simplest technique for achieving reproducibility is to specify the name of a bug scenario within a publicly available set of benchmarks and the release version of the operating system used by the experiment, and to provide a set of source code files for the faulty and fixed versions of the program, omitting all other details. Crucially, this approach misses details, including the state of relevant environmental variables, installed packages on the system (and their associated versions), and the state of various configuration files (many of which may be unknown to the experimenter).

Running the experiments within the specified operating system is far from guaranteed to produce the same results, unless all other relevant setup stages are made clear and followed correctly. Additionally, in some cases, the correct functioning of the program also relies on the absence of certain files and packages from the system. In practice, this leads to needless hours spent trying to discover why a particular experiment won't reproduce the original results and taking measures to try to align the results with those that are expected. In the worst case, the original experiment may have suffered from a mistake somewhere in its configuration, but without clear reference to the complete setup this is unknowable.

Virtual Machine Image

The most common approach to the problem of reproducing results is to provide an accompanying virtual machine image, used to conduct the experiments. Conducting experiments within a VM introduces considerable performance overheads, significantly increasing the already long run-time of APR experiments. Additionally, such an approach increases disk space and bandwidth requirements substantially as the user needs to download a complete image of the VM to disk for each bug scenario should they wish to replicate its results.

Furthermore, such an approach may prove excessively cumbersome when one wishes to modify or extend the experiment. In particular, years later, aspects of the virtual machine may cease to work correctly (e.g., package managers), or additional, conflicting software may need to be installed to perform a modified version of the

experiment. Since the steps taken to create the VM are unclear, modification can prove difficult and impractical. [5]

Amazon EC2 AMI

Rather than supplying a virtual machine image in a common, open-source format, one may provide a machine image as a proprietary AMI (Amazon Machine Image),[6] usable on Amazon's EC2 cloud computing service. This approach, used in later experiments performed with GenProg, avoids some of the overheads involved in running a standard VM on the user's hardware, allowing it to be run on a faster, purpose-built cloud computing platform. One may then provision (multiple) machines on the EC2 platform using the provided AMI to reproduce the results of an experiment.

Since this approach only provides an image of the system used to perform the experiment, without details of its creation, it also inherits the problems of extendibility and bandwidth from the VM approach. As the AMI file format is proprietary, the problem of extendibility is further exacerbated. This restriction also prevents the machine from being run on locally available hardware (e.g., university compute clusters), and forces the user to use—and pay for—Amazon's compute services to reproduce the result.

Vagrant and Puppet

In earlier attempts to produce a suitable platform for program repair research, we explored the combined use of Vagrant[7] and Puppet[8]. We used these technologies to generate and provision a VM from a `Vagrantfile`, describing a base VM in a human-readable format, and a set of `Puppetfiles`, providing instructions on how to configure the VM and install the necessary packages. Whilst this approach partially solved the problem of transparency encountered when using VM images, extending the VM is still overly tedious and the performance remains inferior compared to a container-based approach.

RepairBox

To avoid these problems, we present RepairBox, a platform for safely conducting controlled, reproducible program repair experiments, with minimal compromise to

[5] We encountered this problem of extendibility when attempting to install a newer version of OCaml, required by the software we were using, onto the VM of GenProg's 2010 results. The package manager used by Fedora had transitioned from `yum` to `dnf`, and all of its sources were outdated. Further, packages installed on the system prevented this version of OCaml from being installed.

[6] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html`. [Accessed: May, 2017].

[7] `https://www.vagrantup.com/`. [Accessed: May, 2017].

[8] `https://puppet.com/`. [Accessed: May, 2017].

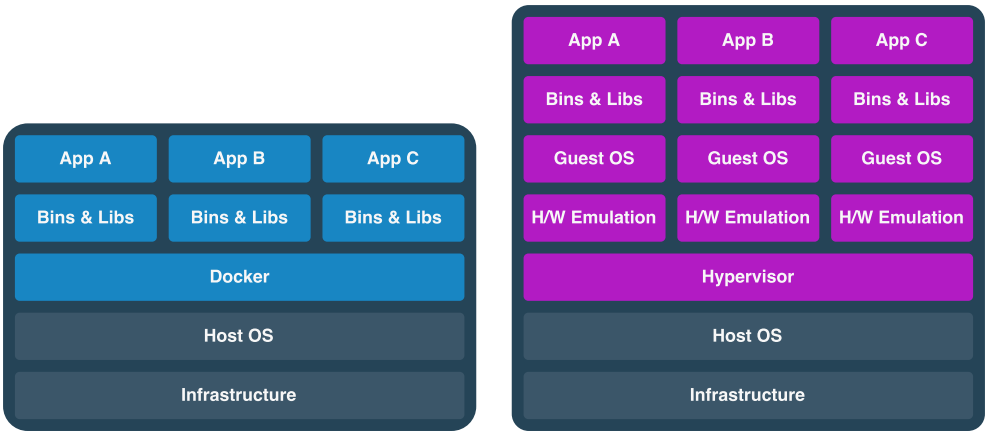

Figure 3.5: Docker Containers (left) vs. Virtual Machines (right). Each container sits on top of the Docker runtime, which provides access to the kernel of the host machine. Each virtual machine virtualises its own stack, down to the level of the hardware, and sits on top of a hypervisor, which allows multiple VMs to run on the same machine.

performance. RepairBox operates by isolating individual bug scenarios and repair tools into separate, minimal Docker containers that can be easily inspected and extended. By packaging bug scenarios as containers, RepairBox attains close to bare metal performance, an important trait when performing a large number of repeats of computationally-intensive program repair experiments.

As its means of packaging each bug scenario into a controlled, self-contained, executable environment, RepairBox uses Docker containers. Container images provide a lightweight means of encapsulating a piece of software and all of its dependencies into a single, portable executable package, referred to as a container.[9] Unlike virtual machines, which virtualise at the hardware level, containers virtualise at the operating system level (illustrated in Figure 3.5). To do so, containers share the operating system kernel of their host machine. As a result, containers use less compute and memory resources than their VM counterparts, giving them a significantly higher level of performance [Felter et al., 2015]. Additionally, since containers are built as a series of file system *layers*, they typically size in the tens of MBs, rather than tens of GBs.

By packaging bug scenarios as Docker images, users can quickly download pre-built images from DockerHub[10], using the `repairbox download` command.

To conduct experiments using these *repair boxes*, users can combine them with containers holding the repair tools required for the experiment, as illustrated in Figure 3.6. By adopting this approach, users can provide transparent, high performance *executable experiment designs* as the replication package for their research.

[9] `https://www.docker.com/what-container`. [Accessed April, 2017].
[10] `https://hub.docker.com`. [Accessed: April, 2017].

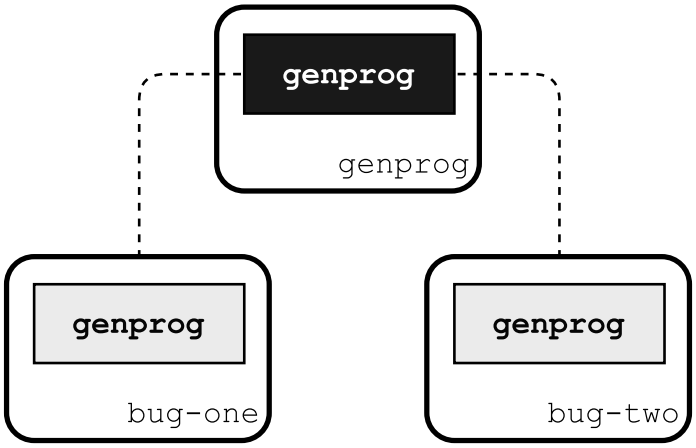

Figure 3.6: Repair boxes can be used with repair tools by mounting the binaries, provided by a repair tool container, into the repair box, via volume mounting.

Each bug scenario in the repository is accompanied by a Dockerfile. This file provides an executable set of instructions for constructing the corresponding image. An except of the Dockerfile for one of the LIBTIFF scenarios is given in Figure 3.7.

To compose the image, Docker uses each instruction in the Dockerfile (e.g., **RUN**, **FROM**, **ADD**) to construct a series of read-only layers. Each of these layers represents a filesystem difference.[11] This approach allows Docker to cache layers, thus reducing build time. Importantly, it also allows images to be used as the base layer of another image, using the **FROM** instruction.

RepairBox uses layering to efficiently provide minimal images for its bug scenarios. Each bug scenario is comprised of a stack of images, illustrated in Figure 3.8. At the bottom of this stack sits the universal base image used by all bug scenarios, which provides common functionality to all repair boxes. Above this image there rests an image for the particular dataset to which the scenario belongs. On top of the dataset image sits an image for the particular program. Finally, at the top of the stack rests an image for the bug scenario.

Each repair box is structured identically, with all of its relevant files (e.g., source code, test suite, meta-data) within the `/experiment` directory. To allow test prioritisation, reduction, and sampling to be used, each scenario implements a simple bash-based test harness, which allows a single, specified test to be executed, rather than the entire suite.

For some bugs within REPAIRBOX, the test suite is provided by the original dataset. For the rest, where we observe issues with test case interference and inadequate checking (i.e., only the exit status of the program), we use Pythia, described in Section 3.2, to generate a stronger oracle from the original test suite. For each test execution, PYTHIA creates a sandboxed environment (realised as a temporary direc-

[11] https://www.docker.com/what-container. [Accessed: April, 2017].

```
FROM squareslab/repairbox:manybugs64
MAINTAINER Chris Timperley "christimperley@gmail.com"

# install dependencies
RUN sudo apt-get update && \
    sudo apt-get install -y --force-yes \
      gcc-multilib psmisc zlib1g-dev && \
    sudo apt-get clean && \
    sudo apt-get autoremove && \
    sudo rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# download Libtiff source code
RUN git clone https://github.com/vadz/libtiff src
...
```

Figure 3.7: An example Dockerfile for one of the LIBTIFF scenarios within REPAIRBOX.

tory) in which to perform the test. This behaviour allows us to avoid the problem of test interference.

Containers

By using a Docker container as a means of accurately replicating a particular bug scenario, rather than employing a VM-based approach, we can create reproduction packages for APR experiments with the following advantages:

- **Transparency:** since the `Dockerfile` specifies the exact steps required to build the container, all of its details available to the user, making future modification and bug understanding simpler.
- **Minimality:** each `Dockerfile` is based on a minimal base image (for either Ubuntu 14.04 or Fedora 23), between 200 to 400 MB, which is extended with the minimal set of packages and configuration required to replicate a particular bug and nothing more.
- **Extendibility:** Docker container images are built around the concept of layering, where the image from a container is built in a series of layer images, each of which are cached, starting from a base image. Each layer represents the resulting system image when a given instruction (e.g., `apt-get install ...`) is executed on the resulting container for the previous layer.

  Instead of building each of the REPAIRBOX bug scenarios from scratch (using the provided `Makefile`), the user may download a pre-built image from DockerHub using REPAIRBOX's command-line interface.

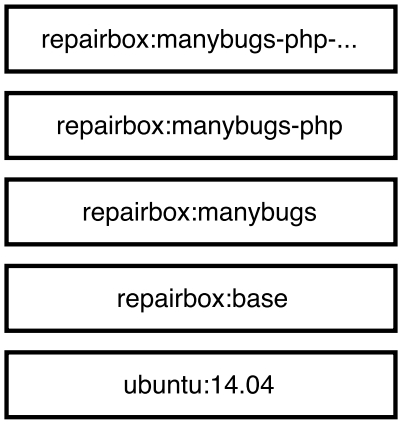

Figure 3.8: The Docker image for a given bug scenario is built as a series of layers. Each layer is shared by a number of images on the layer above, reducing disk usage and build time.

- **Performance:** unlike traditional, hypervisor-based virtual machines, containers share the kernel of their host operating system, removing the need to virtualise the kernel and hardware stack. As a result, the performance of containers tend to be significantly better than VMs, using fewer clock cycles, and less memory and disk space.

  Due to the fact that containers share the kernel of the host machine, bugs that rely on a particular kernel are difficult, and in some cases, impossible, to reproduce within a Docker container. This limitation prevents the inclusion of the VALGRIND bug scenarios from ManyBugs. A similar limitation applies to bug scenarios that only occur on 32-bit architectures. In this case, however, we can usually reproduce the bug by passing the appropriate flags to MAKE, or by using a Docker image with 32-bit libraries as the base image.

  At the cost of performance, those bug scenarios could be included into REPAIRBOX by hosting them within a suitable virtual machine (possibly provided by BOOT2DOCKER[12]). We leave this to future work. In most cases, however, we have found that a specific kernel is not necessary.

Composition

To simplify the management of repair boxes through the REPAIRBOX command-line interface, discussed in Section 3.3.5, we require that repair boxes provide manifests. These manifests, encoded as YAML files, specify: the name of the bug scenario, the dataset (and optionally, the program within a dataset) to which they belong, the name of the Dockerfile used for their construction, as well as any build time

[12] http://boot2docker.io. [Accessed: April, 2017].

```
bug: 69223-69224
build-arguments:
  scenario: python-bug-69223-69224
dataset: manybugs
dockerfile: Dockerfile.bug
program: python
version: 0
```

Figure 3.9: An example bug scenario manifest.

```
version: 0
tool: genprog
image: christimperley/genprog:latest
environment:
  PATH: "/opt/genprog3:${PATH}"
```

Figure 3.10: An example tool manifest describing GenProg.

arguments that should be passed to Docker. An example manifest is given in Figure 3.9. Similar manifests must also be provided for datasets and programs.

All of these entities (bug scenarios, programs, datasets) must provide a version suffix parameter. Complete version numbers for entities are generated by concatenating these numbers. For instance, for the bug scenario described in Figure 3.9, the complete version number will be given by "D.P.B", where D, P, and B are the version suffixes for the dataset, program, and bug, respectively. This information is used by RepairBox to determine whether a particular component is out-of-date and should be updated.

Manifests may also be provided for containerised tools (e.g., GenProg, or Daikon). An example tool manifest is given in Figure 3.10. This manifest specifies a name and version for the tool, the name of the Docker image for its container, and a dictionary describing which environmental variables should be modified within the repair box upon launch.

Command-Line Interface

RepairBox hides the details of its implementation from its end-users through a simple, self-documenting command-line interface (CLI). Below, we briefly discuss the features of this CLI and provide examples of its use:

- `repairbox -h`: describes each of the commands provided by the RepairBox CLI.
- `repairbox list`: produces a list of all artefacts of a given kind (e.g., bug scenarios, datasets, tools). Example uses of this command are given in Figure

```
$ repairbox list bugs

Bug                         Latest    Installed    Remote
-------------------------   ------- - ---------    -------
manybugs:libtiff:2005...    0.0.0.0   0.0.0.0      0.0.0.0
manybugs:libtiff:2005...    0.0.0.0   0.0.0.0      0.0.0.0
manybugs:libtiff:2005...    0.0.0.0   0.0.0.0      0.0.0.0
manybugs:libtiff:2006...    0.0.0.0   0.0.0.0      0.0.0.0
...

$ repairbox list tools

Tool          Image                            Installed
------------  -------------------------------  -----------
genprog       christimperley/genprog:latest  True
searchrepair  searchrepair:latest            True
shuriken      shuriken:latest                True
...
```

Figure 3.11: Example uses of the `repairbox list` command.

3.11.

- `repairbox install`: installs a specified repair box (or collection of boxes). If the latest version of the repair box is hosted on DockerHub, its image will be downloaded. If not, the repair box and its dependencies will be built from scratch locally.

  This command may also be used to install a registered tool onto the local machine by downloading its image from DockerHub.

- `repairbox uninstall:` removes a specified (set of) repair boxes from the local machine.

- `repairbox build`: constructs a named repair box (or collection of boxes) and their dependencies locally.

- `repairbox download`: downloads a pre-built image for a (set of) repair box(es) or a repair tool from DockerHub.

- `repairbox launch`: spawns an interactive container for a given repair box. The names of supported repair tools may also be passed via the `--tools` argument, causing the binaries for those tools to be imported into the container. Internally, this command oversees the generation, linking and clean-up of the required containers. An example use of this command is given in Figure 3.12.

```
host> repairbox launch  manybugs:libtiff:2005-12-14-6746b87-0d3d51d \
                        --with genprog daikon
rbox> genprog problem.json
...
```

Figure 3.12: An example use of the `repairbox launch` command.

3.4. Methodology

Using the resources provided by the RepairBox project, together with Pythia, we put forward a methodology for automated program repair experiments that is used throughout the rest of this thesis.

Executable Experiment Designs

Experiment designs are described by simple scripts which interact with containerised bug scenarios and tools through the RepairBox interface. This approach allows experiments to be concisely described, more easily understood, and to be accurately reproduced. By imposing this requirement on the experiment, the benefits of using containerisation, namely performance, extendibility, and isolation, are inherited by the experiment. Since this script contains all of the steps necessary to run the experiment, it essentially becomes an *executable experiment design*.

Host Platform

In the interest of identifying and avoiding rare bugs that only occur when particular kernel versions are used, the experiment must state the kernel version and operating system release used by the host machine. Additionally, where relevant, a description of the hardware specifications of the host machine should also be given.

For the majority of the experiments conducted in this thesis, we used cloud computing facilities to distribute the load across a large number of machines, hosted on Amazon AWS and Microsoft Azure. Directions on expediting the running of experiments within a distributed environment are provided with each experiment's reproduction package.

Justification of Benchmarks

In addition to supplying a simple script capable of running the experiment, this methodology requires that users justify their selection of benchmarks within the context of their research questions. Any weaknesses, trade-offs, or artificiality within the bug scenarios must be stated and mitigated.

Statistical Hypothesis Testing

Where possible, we advocate that statistical hypothesis testing be used to demonstrate the statistical significance of any results. In most cases, we do not have reason to assume that the results will fit to a particular probability distribution, and so we must use nonparametric statistics to describe and test them [Downey, 2011]; nonparametric statistics should also be used when the median is a more appropriate measure of the central tendency of the data. Although nonparametric statistics allow us to describe and test data without assuming a particular distribution, they do so at the cost of decreased statistical power (i.e., the probability of correctly rejecting the null hypothesis). This decreased statistical power means that a greater number of samples are needed to increase the likelihood of detecting a significant effect when one exists.

To test whether the performance of one search algorithm is stochastically greater than that obtained by another, a one-sided test for statistical significance should be used, such as the Mann-Whitney U test [Mann and Whitney, 1947]. To test that two groups are not drawn from the same distribution, one may use a two-sided Kolomogorov-Smirnov (KS-2) test [Downey, 2011].

In addition to determining statistical significance, one should also measure *effect size* (i.e., how big is the difference between the two groups?). Vargha and Delaney [2000]'s A_{12} measure is a popular choice for nonparametric effect-size measurement. Intuitively, A_{12} describes the probability that a randomly selected value from group A is greater than one from group B [Neumann et al., 2015]. Vargha and Delaney [2000] provide guidelines for interpreting the size of the effect: 0.5 indicates no effect, 0.56 is a small effect, 0.64 is a medium effect, and 0.71 is a big effect.

3.5. Conclusion

In this chapter, we identified and discussed the problems of inadequate test suites and incomplete reproduction packages within studies of APR, before introducing Pythia and RepairBox to address them. Using both of these tools, we propose a more robust, efficient experiment methodology, allowing research to be confidently performed for a diversity of research questions using hundreds of bug scenarios, collated and edited from several existing datasets.

RepairBox and Pythia are available to download at:

`https://github.com/squaresLab/RepairBox`

`https://github.com/ChrisTimperley/Pythia`

Over time, we intend to explore further avenues of improvement to RepairBox, allowing it to be used for a wider range of experiments within program testing

and genetic improvement. Additionally, we plan to incorporate further sources of bugs into RepairBox, including the DARPA Security Challenge[13], IntroClass, and CVE[14].

[13] http://archive.darpa.mil/cybergrandchallenge/
[14] https://cve.mitre.org/

CHAPTER 4

Fault Localisation

For automated program repair to become an economical and effective means of fixing bugs, and thus break through into the mainstream, techniques will need to be capable of solving a wide range of bugs within a reasonable resource window. To go beyond the small number of fixes possible with current techniques, search-based repair will need to incorporate a richer repair model, as discussed in Chapter 5. Additionally, techniques will need to be capable of generating patches spanning multiple lines, going beyond the single-line patches currently produced by all but two approaches [Le Goues et al., 2012a; Mechtaev et al., 2016]. As a result of these requirements—a *richer repair model* and the ability to produce *multi-line fixes*—the search space will undergo a combinatorial explosion. As one approach to tackling this growth, more accurate means of localising the source of faults will need to be discovered. At present, all repair techniques make use of relatively simple spectrum-based fault localisation techniques, which often prove ineffective at pinpointing suspicious statements within a basic block (since all statements within a basic block share the same coverage, except in the case of crashing and non-terminating faults).

Previous work by Qi et al. [2013] compared the effectiveness of several spectrum-based fault localisation techniques by measuring the average number of candidate patches (NCP score) that were evaluated before a repair was found (using GenProg). On a sub-set of the ManyBugs benchmarks [Le Goues et al., 2015], Qi et al. [2013] found that the Jaccard metric (discussed in more detail in 4.1) yielded the lowest NCP. Almost all of the solutions that were found during the search were the subject of overfitting, however, and so it is unclear whether Jaccard was more effective at localising the fault or simply better at selecting areas prone to overfitting. Regardless, the accuracy of spectrum-based fault localisation is simply not good enough for the purposes of repair. Often, hundreds of statements will be assigned the same suspiciousness score—a value which determines the probability of a statement being selected as a subject for repair.

Whilst there are a number of complementary methods for fault localisation, discussed in 4.1, almost all of these are performed in an offline, preprocessing step. In this chapter, we turn our attention to the young and promising field of Mutation-Based Fault Localisation (MBFL) [Moon et al., 2014a; Papadakis and Le Traon, 2015] to determine whether information collected over the course of the search, in the form of mutant test suite results, can be harnessed to refine the fault localisation online. More specifically, we ask:

RQ1: Can the results of candidate patch evaluations, gathered over the course of the search, be used to improve the accuracy of the fault localisation, online?

To answer this question, we first perform an exploratory analysis of the test case results of randomly sampled single-edit mutants within GenProg's search space. Based on the findings of this analysis, we propose a number of fault localisation techniques which incorporate knowledge of mutant test suite evaluations. In contrast to the impressive results demonstrated by previous studies on MBFL, we find that the positive contributions of this knowledge to the accuracy of the fault localisation are negligible. In Section 4.4, we speculate on the reasons for this result and discuss alternative means of improving the fault localisation process.

In summary, the primary contributions of this chapter are the following:

- A detailed analysis of the test suite results of mutants sampled from GenProg's search space, covering 28 bugs in six real-world C programs.
- An evaluation of several alternative fault localisation techniques that use the test case outcomes of mutants produced during the search.
- An informed discussion of the limitations of GenProg's statement-level repair actions in identifying faulty locations and how these limitations might be addressed through the use of more granular repair actions.

The rest of this chapter is structured as follows: Section 4.1 presents a brief review of several approaches to automated fault localisation, the majority of which have not been applied to automated program repair. Section 4.2 conducts an analysis of the test case outcomes of mutants randomly sampled from GenProg's search space, in an effort to determine whether test suite behaviour can be used to discriminate faulty statements from correct ones. Section 4.3 proposes and evaluates a number of fault localisation measures based on the findings of the previous section. Finally, Section 4.4 summarises the findings of the study and discusses a number of possible explanations for the lack of noticeable improvement, as well as future directions for research.

4.1. Background

In this section, we provide the reader with a brief introduction to a number of different methods for automated fault localisation.

Spectrum-Based Fault Localisation

Spectrum-Based Fault Localisation (SBFL), also referred to as statistical fault localisation [Landsberg et al., 2015], operates by assigning each statement in the program with a *suspiciousness value* $s \in \mathbb{R}$, encoding the likelihood that the statement is (partly) responsible for the fault. In most cases, SBFL proceeds to rank each statement within the program by its suspiciousness. An exception to this, is in the case of automated repair, where most approaches use the suspiciousness values themselves to better decide exactly how resources should be spent during the search for a repair.

When used to manually identify and repair bugs, suspiciousness information is visualised by highlighting source code lines within a viewer with different brightness and hue components, allowing suspicious statements to be quickly identified, as illustrated in Figure 4.1 [Jones et al., 2002]. Through such visualisation, developers can better decide where to focus their search for the bug, leading to a substantial reduction in the time and effort involved in debugging [Jones et al., 2002].

To generate a set of suspiciousness values, SBFL first generates a *program spectrum*, concisely summarising the coverage data in the form of an n-by-4 array, where n is the number of statements within the program [Yoo, 2012]. Each row in the array encodes a summary of the execution data for a given statement, in the form of four integer variables, (e_p, e_f, n_p, n_f). e_p and e_f state the number of times that the statement was executed by a passing and failing test case, respectively. Conversely, n_p and n_f specify the number of times the statement was not executed by a passing and failing test case. These counters may record the number of times a statement was executed in total (i.e., multiple executions of a statement for a given test case may count), or as is more often the case, the number of test cases that executed the given statement (i.e., only a single execution is recorded for each test case).

Once the program spectrum has been generated, SBFL proceeds to compute suspiciousness values for each statement through the use of a suspiciousness metric, also known as a risk evaluation formula. This metric maps each row in the program spectrum, (e_p, e_f, n_p, n_f), to a suspiciousness value, represented by a real number. By applying this mapping across the entire spectrum, the array is reduced to a list describing the relative suspiciousness values of each statement in the program. An example of a program spectrum can be found in Section 2.1.

Spectrum

The term "spectrum" was introduced to the field by Reps et al. [1997], and later generalised by Harrold et al. [2000]. Generally, the term is taken to refer to the frequency of statement execution, whether the frequency be a measure of the number of test cases that execute a given statement, or the total number of times a statement is executed by all (positive or negative) test cases. Alternatively, when used as a debugging aid for humans, lines may be used as the entities of interest when

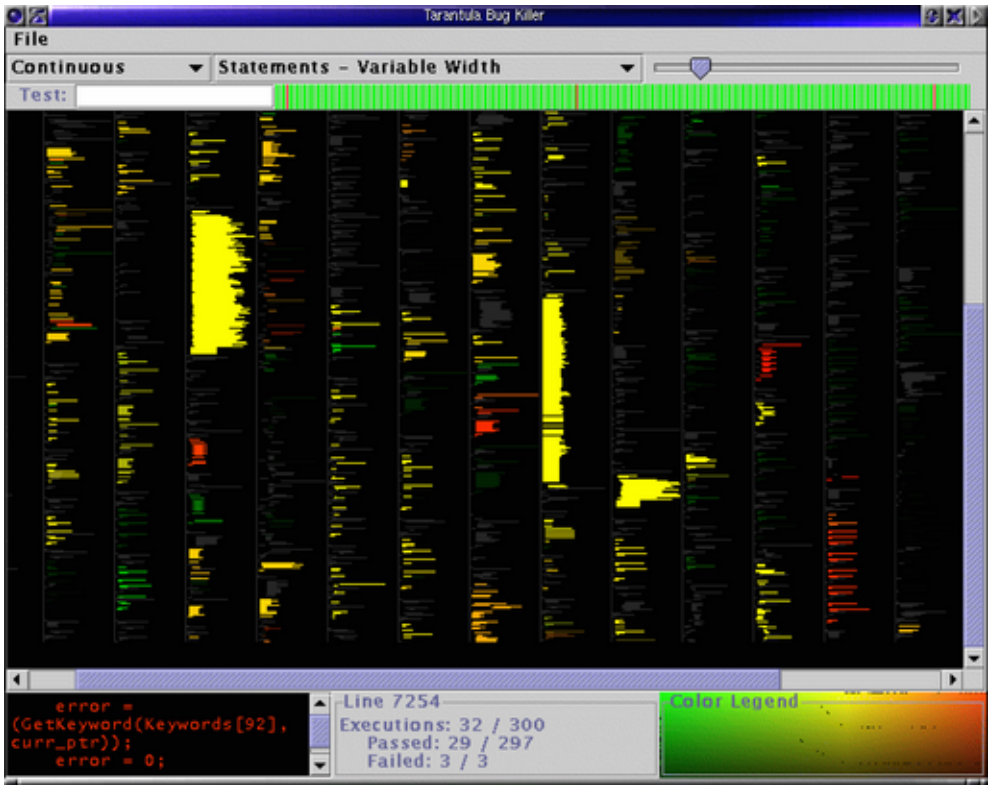

Figure 4.1: A screenshot of the original Tarantula fault localisation visualisation tool. Lines coloured red are executed primarily by the failing test cases, suggesting a high suspiciousness. Lines that are mostly covered by the passing test cases are coloured green. Brightness is used to indicate a form of confidence in the suspiciousness attributed to a given line: The brightness of a line is given by greater of the fraction of failing tests covered by the line, and the fraction of passing tests covered.

Source: http://spideruci.org/fault-localization/ (May 2017).

computing the spectrum, rather than statements. Similarly, one can monitor the execution of function calls, program paths, n-grams, program slices, use-def chains, invariants, and more [Naish et al., 2011; Renieris and Reiss, 2003].

Suspiciousness Metric

As numerous studies have shown [Jones and Harrold, 2005; Qi et al., 2013; Yoo, 2012], the effectiveness of SBFL can be attributed to the choice of suspiciousness metric, of which there are many. Since the publication of Tarantula [Jones et al., 2002], there has been a proliferation of suspiciousness metrics. Over the rest of this section, we shall investigate a handful of the most popular and successful metrics put forth in the literature.

- **Set Union and Intersection:** Two of the simplest suspiciousness metrics are the set union and set intersection metrics [Renieris and Reiss, 2003]. These metrics assign a unitary suspiciousness value to all statements that belong to a given set. The sets for the set union and set intersection metrics are given in Equations 4.1 and 4.2, respectively:

$$f - \bigcup_{S} s \tag{4.1}$$

$$\bigcap_{S} s - f \tag{4.2}$$

  where f is the set of statements covered by a (singular) failing run, each $s \in S$ is the set of statements covered by a given passing run, and $-$ represents the non-symmetric set difference.

  In its standard form, the set union metric assigns a unitary suspiciousness value to each statement within the difference of the set of statements executed by a single failing test case, and the union of the statements executed by the passing test cases. In cases where the faulty statement is executed by both the passing and failing test cases, such as an incorrect conditional, the faulty statement will be missed entirely. The intuition behind the set union metric is that the bug is to be found within the statements uniquely executed by the failing test case.

  The set intersection metric, on the other hand, is built upon the polar opposite intuition: the faulty statement is absent from the failing test case. Whilst the set difference may aid the programmer's understanding of the faulty test case behaviour, it also prevents the program from being repaired automatically, as all changes will go unnoticed (since none of the mutated statements are ever executed by the failing test case).

Pan and Spafford [1992] and Chen and Cheung [1997] propose *soft* extensions to these metrics, which incorporate frequency of execution, similar to the majority of other suspiciousness metrics.

- **Nearest Neighbour:** A slightly more advanced technique than the set union and set intersection methods discussed above is the *nearest neighbour* metric, introduced by Renieris and Reiss [2003].

  This approach first determines the passing test case with the smallest distance to the failing test, where distance is measured as the cardinality of the non-symmetric difference between the failing test spectrum and passing test spectrum, as shown by Equation 4.3:

$$F - P \tag{4.3}$$

  where P is the set of statements covered passing a passing test, and F is the set of statements covered by the failing test.

  Once the passing test with the closest behaviour to the failing test, as measured by the distance metric, has been found, all statements within $F - P$ are assigned a unit suspiciousness value.

- **Tarantula:** A highly popular metric, credited as the first system to apply spectrum-based ranking to software diagnosis for an imperative language [Jones and Harrold, 2005; Naish et al., 2011]. Originally designed as a means of visualising suspicious locations within a faulty program [Jones et al., 2002], but later adapted and more widely used as a suspiciousness metric in its own right [Jones and Harrold, 2005]; an adapted form of this metric is given in Equation 4.4.

$$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p}+\frac{e_f}{e_f+n_f}} \tag{4.4}$$

  Although not optimal, Tarantula has been shown to give good results for automated repair [Qi et al., 2013] and has served as the suspiciousness metric of choice for several automated repair systems [Ke et al., 2015; Kim et al., 2013; Le Goues et al., 2012a; Qi et al., 2014].

- **GenProg:** The default suspiciousness metric in GenProg, given by Equation 4.5, attaches a minimal, non-zero weight to statements that are executed by both the positive and negative test cases, and a maximal weighting to statements that are executed by the negative test exclusively. Statements that are unexecuted by the negative test case are assigned a suspicious of zero, and excluded from consideration.

$$\begin{cases} 1.0 & \text{if } e_f > 0 \wedge e_p = 0 \\ 0.1 & \text{if } e_f > 0 \wedge e_p > 0 \\ 0 & \text{if } e_f = 0 \wedge e_p = 0 \end{cases} \tag{4.5}$$

- **Op1 and Op2:** Op1 and Op2, given in Equations 4.6 and 4.7, respectively, are two suspiciousness metrics introduced by Naish et al. [2011]. Unlike previous metrics, Op1 and Op2 are designed to perform optimally for single fault programs on a model program, ITE2, composed of two consecutive If-Then-Else statements. Beyond theoretical optimality, empirical observations have also demonstrated the strong performance of these metrics on a widely studied sub-set of single fault bugs from the SIR dataset.

$$\begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases} \tag{4.6}$$

$$e_f - \frac{e_p}{e_p + n_p + 1} \tag{4.7}$$

- **Jaccard and Ochiai:** Jaccard [Jaccard, 1901], given in Equation 4.8, and Ochiai [Ochiai, 1957], given in Equation 4.9, are both examples of suspiciousness metrics that have been adapted from existing similarity metrics in other fields [Abreu et al., 2007; Naish et al., 2011].

$$\frac{e_f}{e_f + n_f + e_p} \tag{4.8}$$

$$\frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}} \tag{4.9}$$

- **Evolved:** [Yoo, 2012] transform the problem of deciding a suitable metric into a search problem, using genetic programming to evolve an effective metric for automated repair. The results of this search are a set of metrics that outperformed the most successful human-designed metrics proposed up to that point, including Op1, Op2 and Jaccard.

Comparison

Naish et al. [2011], and later Landsberg et al. [2015], proved that many of the proposed suspiciousness metrics are equivalent for the purposes of ranking, by exploiting their *monotonicity*, i.e., $m_1(x) < m_1(y) \implies m_2(x) < m_2(y)$. Although this is an insightful result and further investigation goes towards explaining the successes of these metrics, for the purposes of automated repair, we are more interested in the absolute difference in scores between statements, so that we may more accurately decide how to divide our efforts.

To compare the effectiveness of different spectrum-based fault localisation techniques, Naish et al. [2012] formally introduce the *Expense* measure, which quantifies accuracy as the fraction of statements within the program that would need to be checked by the programmer before a faulty statement is found were they to examine them in rank order.

Although Parnin and Orso [2011] find that ranking statements by their suspiciousness score allowed programmers to find a bug more quickly, work by Qi et al. [2013] demonstrates that suspiciousness metrics with a low expense are not necessarily the best metrics for automated repair. As an alternative, Qi et al. [2013] propose measuring the number of candidate patches (NCP) produced by GenProg before a repair is found. To account for the stochastic nature of the search, the process must be repeated a fixed number of times and averaged.

Comparing the Expense and NCP measures of a number of proposed suspiciousness metrics across 11 benchmark programs taken from the GenProg ICSE 2012 benchmark suite [Le Goues et al., 2012a], Qi et al. [2013] found Jaccard to be the best performing metric, despite previously being shown to produce sub-optimal ranking by Naish et al. [2012].

Following the proposal of the NCP measure, Moon et al. [2014a] introduced an alternative evaluation metric: Locality Information Loss, or LIL for short. Like NCP, LIL is able to approximate the difficulty of finding a repair for a given bug using APR techniques. Importantly, it does so without the need to perform expensive APR trials.

To determine the accuracy of a given set of suspiciousness values, LIL measures the distance between the probability distribution implicitly defined by those suspiciousness values, P_τ, and an ideal probability distribution, $P_\mathcal{L}$. P_τ and $P_\mathcal{L}$ are used to describe the probability of an APR technique selecting a particular statement as the target location of the repair. Thus, the ideal probability distribution, $P_\mathcal{L}$, is defined as the distribution in which each of the faulty lines is assigned a maximal probability; all other correct lines are given an ϵ probability of selection, for reasons of numerical stability. The distance between P_τ and $P_\mathcal{L}$ is computed using the Kullback-Leibler measure, given in Equation 4.10:

$$D_{KL}(P_\mathcal{L}||P_\tau) = \sum_i \ln\frac{P_\mathcal{L}(s_i)}{P_\tau(s_i)} P_\mathcal{L}(s_i) \tag{4.10}$$

where $P(s_i)$ gives the probability of selection for the statement s_i.

On a single bug from the GenProg TSE 2012 benchmarks (`look utx 4.3`), Moon et al. [2014a] showed a correlation between LIL and NCP for a number of different spectrum-based fault localisation techniques. As previously demonstrated by Qi et al. [2013], Moon et al. [2014a] showed that the EXAM score appeared to have no predictive value.

Program Slicing

Program slicing is a technique for reducing programs to a minimal form, known as a *slice*, which still retains some given semantics of interest, known as the *slicing criterion* [Harman and Hierons, 2001]. Slices are generated by removing all parts of the given program which can be determined to have no effect on the slicing criterion. Slicing has been used for aiding program comprehension, debugging, dead code removal, cohesion analysis, and more [Silva, 2012]. Importantly, slicing may be used as a fault localisation technique by reducing the program to only those parts which have an effect on the location at which the failure was observed. Naturally, this prevents slicing from being used to find memory leaks, since the point at which the program crashes—when it runs out of memory—may have no relation to the fault.

Static Slicing

Program slicing was first introduced in 1979, as part of Mark Weiser's Ph.D. [Weiser, 1979] thesis, in the form of a technique that would later become known as *static program slicing* [Weiser, 1981]. Static program slicing computes the set of statements S, referred to as the *program slice*, that may affect the value of a variable of interest $v \in V$ at a statement x. Collectively, the set of variables of interest V and the statement of interest x are termed the *slicing criterion*, given as $C = (x, V)$. The slice for $C = (x, V)$ can be also be computed from the union of the slices of each of its variables $v \in V$, as shown in Equation 4.11.

$$C = (x, V) = \bigcup_{v \in V} (x, \{v\}) \tag{4.11}$$

There are two types of program slice: the *backward slice* and the *forward slice*. A backward slice contains all statements in a given program P that may have an *effect upon* C, whilst a *forward slice* consists of the statements that C may affect.

The program slice is usually computed by gradually removing statements from the program P which are irrelevant to the slicing criterion, leaving a minimal program which retains the desired properties of interest. By employing backwards slicing, we can exploit this to remove all statements from the program which can be shown to have no (static) effect upon C, thus reducing the number of statements the programmer needs to consider and the debugging effort. An example of a backwards static slice is given in Figure 4.2.

Whilst backwards slicing can help to reduce the program to a smaller form, quite often, the resulting slice can remain a considerable size. This is especially the case for large, well-constructed programs, where statements exhibit a high degree of *cohesion*; in such programs, the value of each variable is dependent on many others.

```
1  read(text);
2  read(n);
3  int lines = 1;
4  int chars = 1;
5  string subtext = "";
6  c = getChar(text);
7  while (c != EOF) {
8    if (c == '\n') {
9      lines = lines + 1;
10     chars = chars + 1;
11   } else {
12     chars = chars + 1;
13     if (n != 0) {
14       subtext = subtext << c;
15       n = n - 1;
16     }
17   }
18   c = getChar(text);
19 }
20 write(lines);
21 write(chars);
22 write(subtext);
```

```
read(text);

int lines = 1;


c = getChar(text);
while (c != EOF) {
  if (c == '\n') {
    lines = lines + 1;

  }









  c = getChar(text);
}
write(lines);
```

Figure 4.2: An example of backwards static slicing on a small program. The source code on the right shows the sliced form of the source code on the right, where $(20, lines)$ is set as the slicing criterion. Adapted from example given in [Silva, 2012].

Dynamic Slicing

Whereas static slicing determines the set of statements that could affect the slicing criterion under all possible conditions, *dynamic slicing* [Korel and Rilling, 1998] produces the set of only those statements that are actually executed when the program is supplied with a certain input sequence [Harman and Hierons, 2001]. This technique builds upon static slicing, constructing a *dynamic slice* instead, according to a *dynamic slicing criterion.* The dynamic slicing criterion is almost identical to its static counterpart, except for the addition of an input sequence i, describing the particular set of inputs to the program for which a slice should be produced; it is described using the form $C(x, V, i)$. In the case of APR, each test case can be used to provide a particular input sequence.

Using a dynamic slice, rather than a static slice, tends to give us a sizeable reduction in the number of statements that need to considered when debugging the program for a particular failing test case. However, this reduction comes at the cost of having to compute the specific control and data-flow dependencies produced by the given input sequence; a problem which requires the resource intensive construction of a complex data structure, such as a trace.

Other Forms of Slicing

A recent survey of program slicing techniques [Silva, 2012] listed 29 variants. Below, we briefly focus on the sub-set of these variants which are immediately applicable to fault localisation.

- **Conditioned Slicing:** Rather than slicing according to a specific execution of the program using a given input sequence, conditioned slicing [Ning et al., 1994] allows the slicing criterion to specify a boolean expression that be must be satisfied by the program inputs. For instance, one may provide the boolean expression F, $x = y - 1$, instructing the slicer to consider only the possible executions of the program when its inputs x and y satisfy F. Using this information, the slicer is able to prune unreachable statements, producing a slice that is at least as small as the static slice.

- **Relevant Slicing:** Whilst dynamic slicing yields the slice of program statements that actually had an effect upon the slicing criterion, the relevant slice is a superset of the dynamic slice, including the set of statements that could have affected the slicing criterion [Agrawal et al., 1993]. The slicing criterion used remains the same as that used by dynamic slicing.

  This technique can be used to find statements whose contamination prevents them from impacting upon the variables of interest within the slicing criterion. In such cases, where a statement does not affect the variables of interest, the dynamic slice will fail to include the faulty statement.

- **Hybrid Slicing:** *Hybrid slicing* [Gupta and Soffa, 1995] is an alternative slicing technique that avoids the need to compute large data structures on the order of gigabytes incurred by dynamic slicing, whilst increasing precision over static slicing by still using dynamic information. To achieve this feat, hybrid slicing incorporates knowledge of which statements have been executed (acquired from the coverage information) into the static analysis, removing from the slice those statements which are in a non-executed possible path of the computation [Silva, 2012].
- **Interprocedural Slicing:** The original form of program slicing has since been reclassified as *intraprocedural slicing*, as static slicing fails to exploit the boundaries of procedure calls, leaving statements within the slice that have no effect upon the criterion [Silva, 2012]. To address the loss of precision, Horwitz et al. [1990] use a *System Dependence Graph* (SDG) to capture dependencies within the program. The SDG allows information about the calling context of procedures to be taken into consideration by the slicer [Silva, 2012].
- **Quasi-Static Slicing:** The quasi-static slicing criterion [Venkatesh, 1991] allows a slice to be produced for a program where some of its inputs are fixed but the rest are unknown.
- **Call-Mark Slicing:** Similar to hybrid slicing, call-mark slicing provides a less expensive alternative to dynamic slicing by using the knowledge of which statements were executed, together with the program's input sequence, to yield a smaller program dependence graph (PDG) by pruning unvisited statements [Nishimatsu et al., 1999].
- **Dependence-Cache Slicing:** Similar to call-mark slicing, dependence cache slicing uses dynamic information to remove irrelevant statements from the slice, via a richer PDG to exploit possible data dependence relationships produced by the input data [Silva, 2012; Takada et al., 2002]. With this information, dependence-cache slicing is able to prune infeasible edges from the PDG, yielding a slice that, on average, tends to be smaller than a call-mark slice.
- **Program Dicing:** Uses the set difference of at least two slices to remove statements which appear to be correct (determined from a set of slices for passing inputs) from the slice of a failing input [Lyle, 1987], exploiting the intuition that statements belonging exclusively to the negative slice are more likely to contain the fault.
- **Stop-List Slicing:** Stop-list slicing [Gallagher et al., 2006] augments the slicing criterion with a stop list, specifying which variables are considered uninteresting and should be omitted from the slice. The stop-list is realised by removing all assignments to variables within the list and all data dependencies stemming from those assignments.
- **Barrier Slicing:** Allows the programmer to control which regions of the program may be considered when building the slice, and which areas may not, allowing the programmer to exclude trusted, correct code from the slice

[Krinke, 2003, 2004]. This ability is made possible through the introduction of *barriers* in the slicing criterion, specifying which nodes or edges of the PDG may not be passed during the graph traversal [Silva, 2012].

- **Concurrent Slicing:** As concurrent programs cannot be represented using the PDG or SDG, concurrent slicing must operate on extended, thread-aware forms of the CFG and the PDG, known as the threaded CFG (tCFG) and threaded PDG (tPDG), respectively [Silva, 2012]. When threads are synchronised or allowed to communicate, a new level of complexity is added to the analysis in the form of the interference dependencies introduced between statements. Such dependencies highlight statements that share a data dependency and may be executed in parallel. Those dependencies are not transitive, however, unlike control and data dependencies, and so concurrent slicing techniques must treat them specially. For a recent overview and evaluation of concurrent slicing techniques, the reader is referred to [Giffhorn and Hammer, 2007].
- **External State Aware Slicing:** Sivagurunathan et al. [1997] demonstrated that standard slicing techniques fail to account for the modification of external state through I/O operations, potentially leading to incorrect slices. To address this shortcoming, they associated each I/O operation with its own special variable, allowing the external state to be observed by the slicer. To achieve this, however, the original program must be subject to a transformation which maps the original program to its equivalent in a special language that includes these external state variables. Later, further revisions to the slicing process were introduced by Tan and Ling [1998], and Willmor et al. [2004], which allowed the slicer to account for database operations.

Delta Debugging

Delta Debugging [Zeller and Hildebrandt, 2002] is an efficient method for determining the cause of regressions (i.e., changes to the program that introduce new bugs) between two software versions in worst-case linear time, with respect to the number of changes introduced between the two versions. By analysing the set of differences C, or *deltas*, between an passing version of the software, referred to as the baseline, and a failing version, the DD algorithm is capable to finding the minimal set of failure inducing changes $c \subseteq C$. This technique allows users to find the minimal set of changes that explain a regression.

DD is capable of finding the minimal set of failure-inducing changes c, where $|c| > 1$ and $\forall c' \subset c(result(c') \neq PASS)$, without having to test each of the 2^n possible sets of changes. To find the minimal sub-set, DD employs a variant of binary search to divide and conquer the set of all changes. At each step, the algorithm halves the set of changes into c_1 and c_2, and recursively searches the half which continues to fail, until c contains only a single edit. In the event where neither c_1 nor c_2 contain the failure inducing changes, known as *interference*, the algorithm continues the binary search on each half without discarding any changes.

A more advanced delta-debugging algorithm, DD^+ [Zeller and Hildebrandt, 2002], removes a number of assumptions within DD to allow the approach to be applied in cases where certain changes may prevent the program from compiling or executing correctly in the absence of particular changes. To achieve this ability, the efficiency of the algorithm is degraded, but steps are taken to ensure that in most cases, the algorithm is capable of finding results in a reasonable amount of time.

Although delta-debugging can be an effective technique for finding the cause of regressions, the associated cost of testing at least $\log n$ variants, and its limited scope prevent it from being more widely used as a means of fault localisation within automated repair. However, through a slight rephrasing, the delta-debugging algorithm can be applied to a wider set of problems beyond fault localisation. Instead, we can use DD to find the minimal sub-set $c \subseteq C$ which retains some arbitrary property of interest, I. Exploiting the wider applicability of this technique, GenProg uses the original delta-debugging algorithm as a post-processing step to minimise the sequence of edit operations within the patch produced by the search [Weimer et al., 2009].

Predicate-Based Fault Localisation

Whereas most fault localisation techniques attempt to localise the fault to a particular statement—especially those applied within the context of program repair—B. Le et al. [2016] propose a novel technique, Savant, for identifying faulty methods within Java programs. To perform localisation, Savant uses Daikon [Ernst et al., 2007], an off-the-shelf likely invariant mining tool, to extract plausible invariants for each of the methods within the program. Using Daikon, Savant produces two sets of invariants: one based on observations of the passing test cases, and another for the negative tests. To determine the set of suspicious methods, Savant uses Daikon's Diff feature to find invariant changes (e.g., "x is less than zero" becomes "x is greater than zero") between the positive and negative invariant sets; methods which exhibit such changes in their invariants are deemed to be suspicious.

Given the prohibitive costs of execution trace collection and invariant mining for large-scale programs, Savant uses method clustering and test case selection heuristics to reduce the costs of these processes. To begin with, Savant excludes from consideration, all methods that are not executed by any of the negative tests. The remaining methods are thereafter grouped into clusters on the basis of the positive tests that cover them; methods that are covered by the same positive tests are more likely to be grouped together. After identifying method clusters, Savant produces execution traces for each—rather than an individual execution trace for each method—across all of the negative test cases and a sub-set of passing tests. This sub-set is selected for coverage-adequacy using a greedy algorithm. Specifically, Savant finds a sub-set of positive test cases where each method within the cluster is covered by at least T test cases (where T is a tunable parameter). Using the generated execution traces for each cluster c, Savant generates the following sets of

invariants across each of the methods contained within the cluster:

- inv(F_c), the set of invariants over the negative test cases that cover c.
- inv(P_c), the set of invariants over the positive test cases that cover c.
- inv($F_c \cup P_c$), the set of invariants over all test cases that cover c.

Using those invariant sets, SAVANT produces a series of *invariant change* features, usable by its learn-to-rank model. To generate these features, SAVANT uses DAIKON's Invariant Diff tool to describe the addition, removal, and modification of invariants between the mined invariant sets. From these differences, SAVANT yields 290,163 features, each specifying the frequency of a particular kind of invariant change (e.g., NonZero to EqualZero) between a given pair of invariant sets, based on its 311 invariant types. Together with these invariant change features, SAVANT also uses a set of ten *suspiciousness score* features, where each represents the computed suspiciousness value computed for the particular method by a state-of-the-art SBFL metric. Before these features are used by learning or ranking, each of them is normalised within the range $[0, 1]$.

To learn a model that ranks methods by their likelihood of being faulty, on the basis of their extracted features, SAVANT applies rankSVM [Joachims, 2002]—a popular, off-the-shelf learning-to-rank algorithm—to a corpus of bug fixes. The learning algorithm is fed the methods modified by the programmer as its ground truth for the location of the fault, together with the extracted feature set. Once the model has been learnt, Savant returns a ranked list of methods for a given program, computed using its extracted feature set.

Out of 357 bugs in 5 programs sourced from the DEFECTS4J dataset [Just et al., 2014], SAVANT correctly localises 63.08, 101.72 and 122 buggy methods on average within the top 1, top 3, and top 5 of the ranked lists it generates, respectively. This corresponds to a 57.73%, 56.69% and 43.13% improvement over the best performing SBFL techniques, at the top 1, top 3 and top 5 positions, respectively.

Although SAVANT provides a significant boost in the accuracy of method-level fault localisation, its rank-based nature prevents it from describing the degree of suspiciousness for each method. Studies by Qi et al. [2013] showed that the best performing fault localisation techniques for automated program repair, as measured by the (average) Number of Candidate Patches till a repair is found, was sub-optimal in terms of its EXAM score. To efficiently find faults, automated program repair needs more detailed localisation information. It may be possible to replace the learning-to-rank algorithm used by SAVANT with a more conventional SVM algorithm, and to train the model to compute suspiciousness values directly. The objective could be to reduce the LIL score of the resulting suspiciousness value distribution—a metric that has been shown to be correlated with NCP [Moon et al., 2014b].

MUSE

Mutation-based Fault Localisation, or MUSE, is a relatively new fault localisation technique introduced by Moon et al. [2014a]. Taking inspiration from mutation testing, MUSE attempts to determine the location of a fault by mutating the program and observing its effect upon the results of the test suite. MUSE assumes that changes made to correct program statements, rather than faulty statements, are more likely to introduce new bugs, which can be detected by the positive test cases. MUSE attempts to exploit this assumption to distinguish between faulty and correct statements using the results of a mutation analysis. Empirical evaluation on a set of 12 bugs from the ICSE 2012 benchmarks showed MUSE to produce 25 times more precise information than the best SBFL technique in terms of Expense, and 2.29 times more precise than Op2 according to the more appropriate LIL metric.

Let P be a faulty program, which passes all positive test cases and fails all negative cases, and m_f and m_c be mutants of P which mutate the faulty statement and a correct statement, respectively. Given these definitions, MUSE is founded on two conjectures.

The first conjecture is that negative test cases are more likely to pass on m_f than on m_c, since faulty programs are usually fixed by modifying the faulty statement (and are more difficult to fix through mutation to correct statements). This reasoning is based on the observation that m_f must satisfy strictly one of the three outcomes:

1. **Equivalent mutant**; the mutant introduces a syntactic change to the program but retains the same semantics, in which case the result of the negative test case execution is unchanged, as the bug still remains.
2. **Non-equivalent, faulty mutant:** the mutant modifies the semantics of the program and continues to fail the negative test cases. The bug may or may not be the same as the original fault. Negative test cases are still considered more likely to fail than to pass on m_f.
3. **Non-equivalent, not faulty mutant:** the mutant no longer fails the negative test cases, and the program is considered to be fixed, with respect to the test suite (the resulting mutant may cheat the test suite or introduce new bugs, but neither of these are detectable using the original test suite alone).

As a result of the above, and the observation that modifications to faulty statements within the program are more likely to yield a repair than changes to correct ones, the number of passing negative test cases should be larger for m_f than m_c.

The second conjecture is that positive test cases are more likely to fail for m_c than for m_f. As with the first conjecture, this conjecture is based on the observation that m_c must exist in one of the two states:

1. **Equivalent (neutral) mutant:** the mutant retains the same semantics as the original program, with respect to the test suite, and should therefore have the

same test case results.

2. **Non-equivalent mutant:** by definition, a non-equivalent mutant must fail at least one of the positive test cases, thereby introducing a fault into the program.

As a result of these two conjectures, and based upon observations, it is believed that mutating the faulty statement is more likely to generate a (partial) fix, and that mutating a correct statement is more likely to fail positive test cases. By generating mutants and observing their test case results, MUSE exploits these conjectures to locate the source of the bug. MUSE incorporates this information into a new suspiciousness metric μ, given in Equations 4.12 and 4.13 (with minor changes in notation):

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \mu_m(s) \tag{4.12}$$

$$\mu_m(s) = \frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \tag{4.13}$$

where s is a statement of P, $f_P(s)$ is the set of negative tests covering s, and $p_P(S)$ is the set of positive tests covering s. $mut(s)$ defines the set of all non-neutral mutants of P with changes at s, $\{m_1 ... m_k\}$; note that neutral mutants are ignored, as they do not provide useful information based on the underlying conjectures. For each mutant $m_i \in mut(s)$, let f_{mi} and p_{mi} be the set of failing and passing tests on m_i, respectively. Importantly, MUSE discards all neutral mutants from consideration, prior to calculating μ; these mutants are deemed to contain no relevant information.

$mu_m(s)$ is used to calculate the contribution of knowledge from a mutant's test case results, based on the conjectures. Its first term describes the proportion of negative test cases that now pass on a mutant m, whilst its second term gives the proportion of positive test cases that now fail on a mutant m. When $mu_m(s)$ is summed and averaged over $mut(s)$, these terms become the probability of a change in test case result (i.e., positive test failure or negative test success), reflecting the intuition behind MUSE. Passing of a negative test case will increase the suspiciousness of s, whereas failing a positive test will decrease its suspiciousness.

As it is more likely that a positive test case will fail for m than a negative test case will succeed, a weighting term α is introduced to balance these probabilities by compensating for the difference in their averages. The equation for α is given in Equation 4.14, where $f2p$ and $p2f$ describe the number of test case results which change across all mutants—note that α does not require *a priori* knowledge of the bug.

$$\alpha = \frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f} \tag{4.14}$$

As a result of this balancing, when the second term is subtracted from the first term, a baseline value of zero is produced. For the faulty statements, the first term is expected to be higher, whilst the second term is expected to be lower than the correct statements in P (w.r.t. the test suite).

Equipped with this suspiciousness metric, MUSE generates suspiciousness values for each of the statements within P by following the steps below.

1. MUSE computes coverage information for P using a given test suite T, finding the set of statements that are executed by the negative test cases, since all of the (manifested) faults are guaranteed to be within this set. To calculate coverage information, MUSE uses GCOV. This decision restricts MUSE from handling crashing programs, such as those with segmentation faults or memory leaks. Note, this limitation is purely a technical one, which could be overcome through the use of an alternative coverage monitoring tool. However, this decision also makes it unclear whether MUSE is equally as effective for crashing bugs.

2. A set of mutants are then generated by mutating each of the candidate statements identified in the previous step. These mutants are created by generating a single mutant at each of its identified mutation points using PROTEUM [Maldonado et al., 2001].

3. From the test case outcomes of all mutants, MUSE proceeds to compute $\mu(s)$ for each statement in the program.

To improve the effectiveness of MUSE in cases where mutants for a given statement are absent, and to use existing spectrum-based fault localisation, Moon et al. [2014b] introduce HYBRIDMUSE. Given a set of suspiciousness values produced by MUSE, μ_{MUSE}, and a set of suspiciousness values generated by an SBFL technique, μ_{SBFL}, HYBRIDMUSE combines these values for each statement, as illustrated by Equation 4.15:

$$\mu_{\text{HybridMUSE}}(s) = \text{norm}(\mu_{\text{MUSE}}, s) + \text{norm}(\mu_{\text{SBFL}}, s) \tag{4.15}$$

where $\text{norm}(\mu, s)$, described in Equation 4.16, is used to normalise a given suspiciousness value such that the minimum value is set to zero, and the maximum is set to one:

$$\text{norm}(\mu, s) = \frac{\mu(s) - \min(\mu)}{\max(\mu) - \min(\mu)} \tag{4.16}$$

Across 51 faulty versions of 5 programs taken from the Software Infrastructure Repository, Moon et al. [2014b] find that HYBRIDMUSE is 4.08 times more precise than MUSE, according to the EXAM metric (the fraction of statements that would have to be considered until a fault is found, if one were to order all statements by their suspiciousness). Moon et al. [2014b] speculate that the relative strength of

Sample	EXAM	Rank	LIL
1%	6.20	123.50	6.26
10%	4.60	95.53	6.11
40%	2.79	61.21	5.96
70%	1.83	45.59	5.90
100%	1.65	41.60	5.86

Table 4.1: The average precision of HybridMUSE as its mutant sampling rate is adjusted. Taken from [Moon et al., 2014b].

HybridMUSE over the standard form of MUSE is observed in cases where all mutants at a faulty statement are discarded, causing those statements to be assigned a suspiciousness of zero.

Interestingly, despite exhibiting a significant improvement in precision—as measured by the EXAM score—Moon et al. [2014b] find that MUSE is a factor of 1.12 more precise than HybridMUSE when the LIL metric is used instead; the authors choose to discount this result, given the improvements in EXAM. For the purposes of automated program repair, however, this result may prove to be of greater importance. Moon et al. [2014b] conduct a preliminary study into the correlation between EXAM and LIL and NCP (average number of candidate patches before a repair was found) on a single automated repair benchmark, "look-utx-4.3", using a randomly generated test suite. On this single benchmark, they find that EXAM bears little relation to NCP, but that LIL appears to correlate well.

Despite impressive gains in precision—as compared to SBFL techniques—both MUSE and HybridMUSE require a vast number of mutants to compute their suspiciousness scores; a lengthy and expensive process. To investigate this further, Moon et al. [2014b] measured the precision of HybridMUSE as the number of mutants sampled at each statement was varied. The results of their investigation, listed in Table 4.1, show a small improvement in LIL as the sample size is increased and a much larger improvement in terms of EXAM and the rank of the faulty statement. At sample sizes below 40%, HybridMUSE performs worse than Jaccard (LIL: 6.04), the best performing SBFL technique. MUSE and HybridMUSE appear to be better suited to aiding human-developers in determining the location of the fault than they are at reducing the difficult of automated program repair.

Metallaxis

Prior to the introduction of MUSE, Papadakis and Le Traon [2015] proposed an alternative approach to Mutation Analysis Fault Localisation: Metallaxis. As with MUSE, Metallaxis assigns a suspiciousness to each statement within the program based on the test suite outcomes for a set of randomly generated mutants. Unlike MUSE, Metallaxis explicitly assigns suspiciousness to individual mutants—

building on the intuition that mutants whose test suite outcomes are closest to those the fault are likely to be co-located. Each *killed mutant* m (i.e., a mutant which fails at least one test)[1] is assigned a suspiciousness using the Ochia similarity metric—a commonly used metric for spectrum-based fault localisation—given in Equation 4.17:[2]

$$\mu(m) = \frac{\#K_n}{\sqrt{\#K \cdot (\#K_p + \#K_n)}} = \frac{\#K_n}{\#K} \tag{4.17}$$

where K is the set of tests that kill m (which may be composed of both positive and negative tests), K_n is the set of (covered) negative tests that kill m, and K_p is the set of (covered) positive tests that kill m.

Although MUSE does not explicitly compute suspiciousness values for its mutants, one can view its summation terms as implicit mutant suspiciousness scores. Comparing MUSE with Metallaxis, one observes that Metallaxis reduces the suspiciousness of mutants that are not killed by negative tests. In contrast, MUSE significantly increases the suspiciousness of such mutants. It seems, therefore, that whilst MUSE and Metallaxis share a similar high-level intuition—that mutants can be used to expose unlocated bugs—they each operate under a fundamentally different set of assumptions: MUSE sees failing-to-passing behaviour as a sign of a partial repair, whereas Metallaxis appears to treat it as a sign of overfitting. This difference in assumptions raises the question: "How should the outcomes of negative tests be treated?"

From the suspiciousness values for each mutant, Metallaxis assigns a suspiciousness value to each statement within the program, using the formula in Equation 4.18.

$$\mu(s; M) = \begin{cases} \max_{m \in M_s} \mu(m) & M_s \neq \emptyset \\ 1.0 & \text{otherwise} \end{cases} \tag{4.18}$$

Rather than computing the average of each of mutant at a given statement, Metallaxis aggregates the suspiciousness of statement-mutants by computing their maximum. An advantage of this approach—unmentioned by its authors—is that the cost of computing suspiciousness values is reduced since values may only increase; there is no need to gather full test suite results for statements which have already been assigned the maximum suspiciousness. Thus, we may forgo the generation of mutants (for the purposes of fault localisation) at statements covered exclusively by negative test cases (since each $\mu(m)$ is guaranteed to be 1.0). Whilst this observation allows us

[1] Provided that one only generates mutants at statements that are covered by at least one negative test case, then the only mutants that are not killed are (partial) repairs.

[2] Notation is changed slightly from its original form, to avoid confusion with similar spectrum-based fault localisation terms. The original form of the equation simplifies to the fraction of killed tests that are negatives (since $\#K_p + \#K_n = \#K$).

to reduce the cost of the fault localisation process, it fails to provide a means of discriminating between such statements, unlike MUSE.

By using the maximum suspiciousness of a statement's mutants, Metallaxis is arguably more prone to the effects of outliers. For example, in the case that a neutral mutant m—one which passes and fails the same test cases as the original, faulty program—is generated at a statement s, $\mu(m)$ will go to 1.0, and by extension, $\mu(s)$ will be assigned the maximum suspiciousness of 1.0. If most statements within the program have neutral mutants, then this behaviour becomes troubling; Metallaxis leaves us unable to discriminate between them. Given that previous research by Schulte et al. [2013] showed that roughly a third of mutants within GenProg's search space are neutral, this problem may manifest in practice.

One can compare the behaviour of Metallaxis to that of MUSE by treating the inner term of μ_{MUSE} as the suspiciousness of a particular mutant. From this perspective, we notice differing behaviours and underlying assumptions in the way that MUSE and Metallaxis aggregate mutant results, and how they treat failing-to-passing test outcomes:

- According to Metallaxis, a statement is as suspicious as its most suspicious mutant. This behaviour assumes that the search landscape is mostly composed of non-neutral mutants. If the search space contains a large number of neutral mutants [Schulte et al., 2013], Metallaxis assigns the maximum suspiciousness value to most statements. In contrast, MUSE actively discards its neutral mutants and computes the suspiciousness of a statement as the average suspiciousness of its mutants, indicating a robustness to sampling noise.
- Whereas MUSE significantly increases the suspiciousness of statements containing mutants which pass previously failing test cases, Metallaxis implicitly decreases the suspiciousness of such statements. These behaviours highlight a contradiction between the underlying assumptions of the techniques: MUSE sees partial solutions as signs of a repair, whilst Metallaxis views them as either irrelevant or the result of overfitting.

Both Metallaxis and MUSE have demonstrated significant improvement to fault localisation accuracy but their evaluation has been limited to manually seeded faults in small-to-medium sized programs, sourced from the Siemens [Hutchins et al., 1994] and SIR [Do et al., 2005] datasets. Furthermore, in the evaluation of Metallaxis, the EXAM metric was used to measure accuracy—a metric which has been shown to be inappropriate for purposes of automated program repair [Qi et al., 2013].

4.2. Analysis

In this section, we explore the feasibility of using results of candidate patch evaluations to improve the accuracy of fault localisation. To determine feasibility, we use

a collection of historical, real-world bug fixes to assess whether mutants at statements where a repair was made exhibit different test suite behaviour to those at statements that were unchanged by the patch. In the context of automated program repair, each mutant can be viewed as a candidate repair. Evidence of differing test suite behaviours between the mutants of modified and non-modified statements would suggest that it is possible to (help to) identify the locations in the program at which a successful fix could be made. If no difference in test suite behaviour is detected, it would suggest that the knowledge of the test outcomes for candidate patches are not useful in improving the accuracy of fault localisation.

To focus our analysis, we use GenProg to generate candidate patches, but anticipate the analysis results can generalise to any search-based repair technique that follows a similar paradigm.

Specifically, this analysis answers the following questions:

- **RQ1A:** Do the solutions found by the search occur at the same statement that was modified by the programmer?

  We ask this question to determine whether APR should restrict its attention to the locations modified by the programmer, rather than pursuing alien repairs at other locations.

- **RQ1B:** Can statements that were modified by the human be discriminated from those that were not, on the basis of the passing-to-failing $p2f$ rates of their mutants?

- **RQ1C:** Can human-modified statements be distinguished from non-modified statements, based on the failing-to-passing rates $f2p$ of their mutants?

  We ask RQ1B and RQ1C to to assess the degree to which $p2f$ and $f2p$ contribute to identifying suitable fix locations, if they do at all; previous work has ignored the individual contributions of these parts [Moon et al., 2014a; Papadakis and Le Traon, 2015].

- **RQ1D:** Are statements covered by the fewest number of positive test cases the most likely to contain the fault?

  We ask this question to determine whether this observation could be exploited to improve offline fault localisation.

To answer these questions, we analyse test case results for a sample of single-edit mutants taken from 28 bugs across 6 real-world C programs. 15 of these bugs are artificial, injected into 3 small-to-medium programs, sourced from the Software Infrastructure Repository [Do et al., 2005]—the same source of bugs used to evaluate MUSE and Metallaxis. We include these bugs to determine whether GenProg's repair operators may be used to perform MBFL, rather than traditional mutation testing operators used by existing approaches [Moon et al., 2014a; Papadakis and Le Traon, 2015]. We supplement this dataset with 13 real-world bugs in 3 large-scale,

Program	# Bugs	KLOC	Tests	Artificial?
gzip	6	6	104	✓
grep	2	10	146	✓
sed	7	14	255	✓
OpenSSL	5	248	77	✗
Python	4	446	344	✗
PHP	4	789	8597	✗

Table 4.2: Details of the subjects we studied for our preliminary mutation analysis. KLOC measures the number of thousands of lines of C code in the program, as calculated by cloc. Tests states the average number of test cases used by bugs for that program.

Instance Type:	c4.large
CPU:	Intel Xeon E5-2666 v3 (2 cores)
Base AMI:	Amazon Linux, 64-bit (ami-0b33d91d)
RAM:	3.75 GB
Storage Type:	io1
Storage Size:	8 GB (400 IOPs)
Cost:	$> \$0.1$/hr

Table 4.3: Specifications for Amazon EC2 instances used to perform mutation analysis on 15 artifical bugs.

real-world programs, to determine whether MBFL remains effective when applied in the wild. We obtained these 13 bugs by manually searching for recent patches within the version control histories of these projects. Table 4.2 provides a summary of the programs we used for the study.

To collect the necessary data for the analysis, we first generated a list of all the single-edit patches within GenProg's search space, before randomly shuffling that list and evaluating as many patches as possible within a 12-hour window. This 12-hour random walk was repeated for each of the bugs within the dataset.

For historical reasons, and given its similarity to traditional mutation testing operators, we also ensured that a deletion was attempted at each statement.[3] In fitting with our general methodology, outlined in Chapter 3, each bug was analysed using RepairBox. We used a C4.Large instance on Amazon EC2, described in Table 4.3, to collect data for the 15 artificial bugs, and a DS1_V2 instance on Microsoft Azure, described in Table 4.4, for the 13 real-world bugs.

To fit as many patch evaluations as possible the time window allotted to each random walk, we performed a number of optimisations to the search process, discussed

[3] In earlier experiments, using publicly available bug scenarios with weak test harnesses, we found that the deletion operator was highly effective at pruning the fault localisation.

Instance Type:	DS1_V2
CPU:	Intel Xeon E5-2673 v3 (1 core)
Host OS:	Ubuntu Server 16.04 LTS (64-bit)
Kernel:	4.4.0-77-generic
Docker:	1.12.6, build 78d1802
RAM:	3.5 GB
Storage:	7 GB (3200 IOPs)
Cost:	$0.249/hr

Table 4.4: Specifications of the Microsoft Azure compute instances used to perform analysis on 13 real-world bugs.

below.

- By limiting our attention to single-edit patches, we were able to exploit the assumption that the fix requires only a single edit by restricting our attention to the sub-set of statements that are covered by all negative test cases as candidates for repair, rather than considering those that are covered by a sub-set of the negative tests.
- For each mutant, we restricted its test suite evaluation to the sub-set of tests whose values may have changed as a result of the mutation. To determine this, we found the sub-set of tests that covered the single statement modified by the mutant. Whilst dynamic analysis techniques could be used to further reduce the evaluation effort required, we believe their associated costs are likely to outweigh any potential gains.
- Additionally, we implemented a form of parallel, asynchronous test case evaluation, which reduces CPU idle time, and improves throughput and efficiency.
- We also used a number of weak equivalency checking techniques introduced by Weimer et al. [2013] in AE, including liveness and scope checking, and the removal of duplicate statements from the donor pool.

Results

A brief summary of the results of the mutation analysis is given in Table 4.5. From inspection of these results, we observe that most (compiling) mutants exhibit an all-or-nothing behaviour: either their test outcomes remain unchanged (shown in the "**% Neutral**" column), or all of their tests are failed (given by the "**% Lethal**" column). We also observe low sample rates (< 1 mutant per suspicious statement) for most Python and PHP bug scenarios. This is due to the substantial overheads involved in compiling mutants, and for statements covered by a large number of tests, the significant costs of evaluating hundreds or thousands of test cases for each mutant.

Scenario	% Compiling	% Neutral	% Lethal	Mutants	Sample Rate
mmb-openssl-0a2dcb6	100.00	14.06	50.66	377	2.48
mmb-openssl-4880672	90.72	22.27	35.51	1971	4.82
mmb-openssl-6979583	99.18	26.62	51.23	1097	13.06
mmb-openssl-8e3854a	93.66	38.14	25.59	3028	4.69
mmb-openssl-eddef30	100.00	55.97	16.84	2898	80.50
mmb-php-01c028a	96.43	7.14	0.00	28	0.06
mmb-php-11bdb85	96.88	25.00	0.00	32	0.07
mmb-php-1d6b3f1	88.66	14.57	25.78	741	8.72
mmb-php-9fb92ee	100.00	28.11	37.33	217	0.51
mmb-python-6c3d527	84.66	64.29	13.76	378	0.46
mmb-python-a93342b	81.51	53.42	0.68	146	0.37
mmb-python-b2f3c23	81.83	45.67	14.33	600	3.03
mmb-python-f584aba	32.74	10.50	5.05	733	0.71
sir-grep-v2-DG_1	98.73	39.17	35.99	628	1.00
sir-grep-v3-DG_3	67.06	31.92	31.19	2353	10.14
sir-gzip-v1-KL_2	91.41	23.60	48.79	1898	10.20
sir-gzip-v1-KP_1	92.09	41.11	36.77	2301	11.22
sir-gzip-v1-TW_3	91.34	27.85	40.57	2068	13.34
sir-gzip-v4-KL_1	91.27	43.17	28.83	2886	13.49
sir-gzip-v5-KL_1	90.26	66.75	16.96	6437	16.98
sir-gzip-v5-KL_8	98.68	24.01	31.26	1062	1.05
sir-sed-v2-AG_17	85.77	23.68	30.67	1630	1.52
sir-sed-v2-AG_19	53.50	8.73	19.93	3011	22.47
sir-sed-v3-AG_11	78.79	24.74	33.81	2579	7.39
sir-sed-v3-AG_15	80.71	24.92	26.02	1545	2.95
sir-sed-v3-AG_17	67.57	16.14	27.40	1332	3.95
sir-sed-v3-AG_18	67.85	16.92	24.13	1235	3.48
sir-sed-v3-AG_6	77.06	23.40	31.17	2615	5.56
	84.94	30.07	26.44	1637	8.72

Table 4.5: A summary of the mutation analysis results for each bug scenario. % Compiling specifies the percentage of mutants that successfully compiled. % Neutral describes the percentage of (compiling) mutants that had no effect on the outcome of the tests. % Lethal describes the percentage of (compiling) mutants (covering at least one positive test) that failed all of their covered tests. Mutants specifies the number of mutants generated within the 12-hour random walk. Sample Rate gives the average number of mutants per suspicious statement.

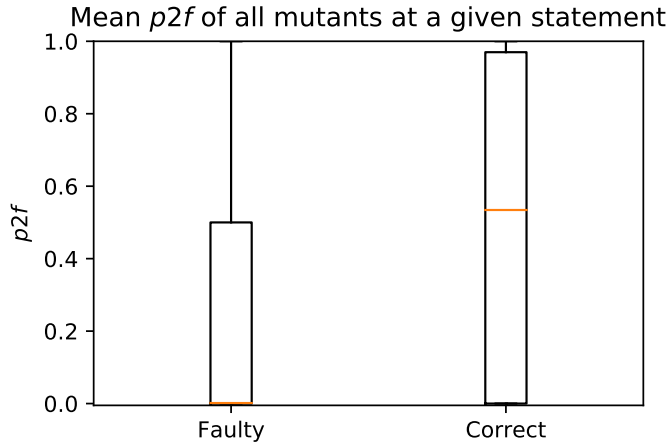

Figure 4.3: Comparing the mean pass-to-failure rate for applicable mutants, we observe different, but overlapping distributions for correct statements and faulty statements ($KS2 = 0.301; p = 0.003, A_{12} = 0.679$ [medium effect]).

RQ1A: Do the solutions found by the search occur at the same statement that was modified by the programmer?

Across the random walk for each of the 28 bugs, we found fixes at a total of 48 different statements. Only 5 of these 48 statements were also modified by the original patch. This finding suggests that GenProg is crafting repairs unlike those that a human would make, supporting arguments made by Monperrus [2014] that automated repair should consider alien repairs, rather than restricting itself to producing human-like repairs.

RQ1B: Can statements that were modified by the human be discriminated from those that were not, on the basis of the passing-to-failing $p2f$ rates of their mutants?

In line with our expectations and previous findings, we observe different mean pass-to-failure rates across all applicable mutants between the statements we assumed to be faulty and those we assumed to be correct, as shown in Figure 4.3. To avoid misleading results, only mutants that successfully compile and cover at least one positive test case are considered applicable mutants. Using a two-way Kolmogorov-Smirnov test, we reject the null hypothesis (with $p < 0.05$) that the samples for the faulty and correct statements are drawn from the same distribution. Between the $p2f$ distributions for correct and faulty statements, we find an effect size of 0.679, indicating that correct statements tend to have higher $p2f$ values than faulty statements. Details on the statistics used in this chapter may be found in Section 3.4.4.

The distributions suggest that faulty statements tend to have a lower pass-to-failure

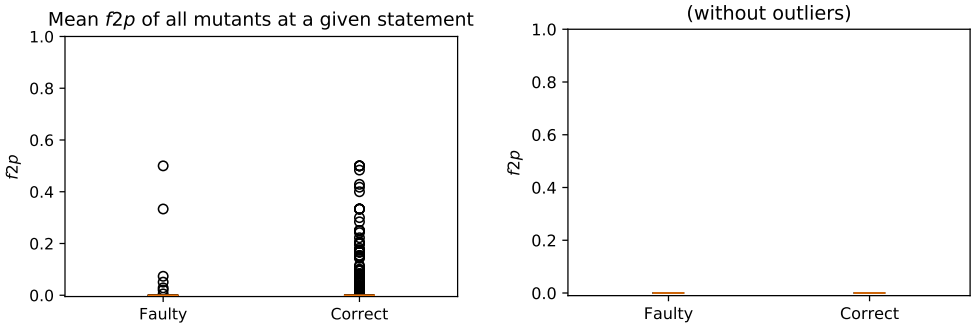

Figure 4.4: We observe similar distributions of mean $f2p$ values for faulty and correct statements ($KS2 = 0.185; p = 0.177$). In both cases, more than half of the mutants at each statement did not pass any of the previously failing tests.

rate than those that are correct, supporting the intuition that modifications to a correct statement are likely to result in a greater degree of functionality loss. In answer to RQ1B, these results suggest that $p2f$ information can be used to partially discriminate between statements that were modified by the programmer and those that were not. Thus, the results provide promise for the use of GenProg's mutation operators to localise the source of faults over the course of the search.

RQ1C: Can human-modified statements be distinguished from non-modified statements, based on the failing-to-passing rates $f2p$ of their mutants?

To answer this question, we first removed all non-compiling mutants from consideration, before also removing all acceptable solutions that were found during the random walk; this puts the $f2p$ values into the context of an ongoing search, allowing us to observe the most complete $f2p$ distributions possible without having yet found a solution. We then compared the mean $f2p$ of all mutants for both statements that were modified by the human and those that were not, illustrated in Figure 4.4.

Results show that mean $f2p$ was zero for the majority of statements, regardless of whether they were modified by the human. On closer inspection, we find that only 2.09% of mutants have any impact on the outcomes of the negative tests (i.e., 97.91% of mutants fail all negative tests). These findings suggest that $f2p$ information may not be particularly effective in distinguishing between faulty and correct statements.

RQ1D: Are statements covered by the fewest number of positive test cases the most likely to contain the fault?

In restricting the search to only those statements covered by all negative test cases, existing spectrum-based fault localisation techniques become partially redundant,

as the contribution of the e_n variable is zero. Instead, it may be preferable to quantify statement suspiciousness using a function of the number of executed and non-executed passing test cases. Thus, we used the results of the analysis to explore how many passing test cases were executed at each statement with an acceptable repair, and whether faulty statements covered fewer positive tests. Accounting for the varying sizes of the test suites, we looked at the fraction of the test suite covered by a faulty statement, or passing test coverage. We computed an adjusted coverage level for each of the bugs, where the minimum number of positive tests covered by any statement under consideration was negated from the number of positive tests covered by a given statement. This allows the levels of coverage to be compared, relative to other candidate statements in the program.

Measuring the adjusted coverage of each bug scenario, we find that 90% are covered by fewer than 2% of the positive test cases, mirroring the intuition behind the original suspiciousness metric used in GenProg.

4.3. Approach

Using the knowledge gained from our mutation analysis, we propose and evaluate two fault localisation strategies for search-based program repair, which may be aggregated. For our purposes, we aggregate localisations by computing product of their suspiciousness values, although more intricate methods may yield better results. To use these layers in a noisy, online context, we ensure that each is numerically stable and that none assigns a suspiciousness of zero to any statement covered by a negative test case. We consider:

Coverage: based on our findings regarding the (adjusted) number of positive test cases covering human-repaired statements, this layer, described by Equation 4.19, creates a localisation that results in a 90% probability that a statement with less than 2% adjusted coverage will be selected, and a 10% probability that a statement with a greater level of coverage will be chosen.

$$\mu_{Cov}(s) = \begin{cases} x & \text{AdjustedCoverage}(s) \leq 0.02 \\ y & \text{otherwise} \end{cases} \tag{4.19}$$

The adjusted coverage of a statement is computed using the formulae given in Equations 4.20 and 4.21:

$$\text{AdjustedCoverage(s)} = \frac{|\text{PositiveTests(s)}| - \text{MinCoverage}}{|\text{PositiveTests}|} \tag{4.20}$$

$$\text{MinCoverage} = \min_{s \in S} |\text{PositiveTests(s)}| \tag{4.21}$$

where |PositiveTests(s)| specifies the number of passing test cases covering a given statement, and |PositiveTests| specifies the number of passing tests overall.

Let the statements that are at or below the adjusted coverage threshold be be given by the set X, and let Y be the set of all other candidate statements, x and y are thus implicitly given by Equations 4.22 and 4.23.

$$\sum_{s \in X} \mu_{Cov}(s) = 0.90 \tag{4.22}$$

$$\sum_{s \in Y} \mu_{Cov}(s) = 0.10 \tag{4.23}$$

Pass-to-Fail: This layer, described in Equation 4.24, assigns values between zero and one to each statement, based on the pass-to-fail rates $p2f$ of its mutants:

$$\mu_{p2f}(s) = \frac{1}{|\text{mutants}(s)| + 1} \cdot \left[1 + \sum_{m \in \text{mutants}(s)} (1 - p2f_m) \right] \tag{4.24}$$

Evaluation

We now investigate effectiveness of a fault localisation that combines the proposed layer. To assess the effectiveness of each candidate fault localisation technique, we use the mutants of the analysis—excluding any solutions, to avoid potential biases—to generate a set of suspiciousness values μ, for each of the programs. We then measure the accuracy of a fault localisation technique by the probability $p(\mu)$ of selecting a statement that contains a fix found during the random walk, as described in Equation 4.25.[4]

$$p(\mu) = \sum_{s \in \text{FixedStmts}} \frac{\mu(s)}{\sum_{s' \in S} \mu(s')} \tag{4.25}$$

Table 4.6 shows results performed across the 11 bugs for which solutions were found during the random walk.

We observe that μ_{Cov} is more accurate than GenProg's standard fault localisation μ_{GP} in 6 out of the 11 cases, marginally worse in 1 case, and considerably less accurate for the remaining 4 cases. When used in isolation, μ_{p2f} is beaten by μ_{GP} for 8 out of the 11 bugs, and for the remaining 3 bugs, μ_{p2f} only provides a marginal improvement in accuracy. This result indicates that passing-to-failing information, used alone, is not particularly effective at identifying suitable fix locations.

[4]Note, we do not measure how well these techniques localise the statement modified in the human repair, since the patterns observed in this data were used to design these techniques.

Scenario	$p(\mu_{GP})$	$p(\mu_{Cov})$	$p(\mu_{p2f})$	$p(\mu_{Cov} \times \mu_{p2f})$
ct-openssl-0a2dcb6	1.227	0.142	0.907	0.046
ct-openssl-6979583	37.681	75.256	15.958	79.753
ct-openssl-8e3854a	0.714	0.106	1.005	0.100
ct-openssl-eddef30	66.667	76.036	45.920	77.583
sir-gzip-v1-KP_1	6.631	4.455	4.673	6.485
sir-gzip-v1-TW_3	9.202	14.211	4.689	15.324
sir-gzip-v4-KL_1	2.685	2.727	1.724	3.083
sir-gzip-v5-KL_1	0.400	0.326	0.360	0.394
sir-gzip-v5-KL_8	2.169	6.585	0.338	4.672
sir-sed-v2-AG_17	0.185	0.019	0.211	0.023
sir-sed-v3-AG_11	0.573	6.953	0.606	3.086

Table 4.6: Comparison of fault localisation accuracies achieved by different approaches, where accuracy is measured by the probability of sampling a statement containing a fix from the resulting distribution. Results are given as percentages.

When μ_{Cov} and μ_{p2f} are naively aggregated by computing their product, the resulting fault localisation is more accurate than μ_{GP} for 6 out of 11 bugs, marginally worse for 1, and considerably worse for 4. If the online modifications to μ_{p2f} are removed, and $1 - p2f(s)$ is used to compute suspiciousness instead, the resulting localisation outperforms GenProg's fault localisation in all cases.

We also experimented with various ways of incorporating $f2p$ information into the fault localisation, but found the approach either attained near-perfect accuracy (since the only mutants to pass any negative tests were at statements where a solution was found), or substantially worse accuracy (since all mutants other than the solutions at the faulty statements failed all of the negative tests). A larger table of results can be found in Appendix C.

4.4. Discussion & Conclusion

From the results of our evaluation, we observe relatively little benefit in incorporating information learned from the evaluation of candidate patches into the fault localisation, in comparison to previous attempts to use mutation analysis to locate faults. We believe there may be a number of reasons for this result:

- **Lack of mutants:** for a number of bug scenarios, we found that excessively long test suite evaluation and compilation times prevented the search from producing an adequate sample of mutants at each statement.
- **Lack of passing test coverage:** in cases where a large of statements are covered by no passing tests, all of these statements will be assigned a suspicious-

ness score of 1.0 by $\mu_{P2F}(s)$. Consequently, this layer will either suppress the fixed statement, if it is covered by any positive test cases, or it will fail to identify it amongst the many statements without positive test coverage.

- **All-or-nothing $f2p$ response:** within GenProg's search space, we observe erratic negative test case behaviour. In some cases, the only statements to pass a negative test case were those that could be repaired. In other cases, negative test case passes were much more common, whilst no mutants at the statement that was repaired (excluding the solutions) passed any negative tests. If one knew which type of $f2p$ response one was dealing with, a more accurate fault localisation might be possible. In the future, we plan to explore whether the rarity of negative test passes might be used to determine whether a given failing-to-passing event is a coincidence, or indicative of a potential repair at that statement.

- **Coarsely-grained mutation operators:** one explanation for the relative lack of success in incorporating the results of the mutation analysis into the fault localisation may be due to the coarsely-grained nature of the repair operators within GenProg's search space. With such actions, it may be difficult to expose subtle bugs within the statement that might otherwise be identified using finer-grained mutation testing operators. In our preliminary analysis, we find that most repairs tend to either have no effect on the outcomes of the test suite, or to cause all of their covering tests to fail; this all-or-nothing behaviour may be a consequence of the granularity of the search operators.

- **Combining information:** to combine each of the proposed layers of fault localisation from our evaluation, we computed the product of each of the layers; a necessarily arbitrary decision. A more meaningful, effective way of combining information from multiple sources and how to deal with conflicting or corroborating suspicious values is not immediately clear.

Going forward, to translate the potential of mutation analysis approaches such as MUSE into efficiency gains in automated program repair, we intend to explore the following:

- **Richer repair models:** in an effort to avoid the all-or-nothing behaviour exhibited by mutants generated using GenProg's coarsely-grained statement-level operators, we intend to explore the utility of lower-level repair operators including, but not limited to, those traditionally used within mutation testing. Beyond the use of mutation testing operators, we intend to explore whether expression-level operators, such as the replacement of the LHS or RHS of an assignment, or the modification of function call parameters, may be used to predict the location of the fault.

- **Shape prediction:** in addition to investigating whether mutants can be used to predict the location of the fault, we are interested to see whether the results of particular types of mutants can be used to refine suspiciousness beyond the level of the statement, and if they can be used to predict the type of repair that

might be needed (for instance, if replacing an if condition appears to have no effect, that might suggest that a replacement condition is needed).

- **Ensemble learning:** rather than combining fault localisation layers by taking their product, or using a simple weighted average, we may see better results—when using a different model—if ensemble learning techniques are used to find more effective ways of combining these multiple sources of information.

In conclusion, we find that mutation analysis appears to offer little potential for online fault localisation within GenProg's search space. From observation of the mutants, we see that GenProg exhibits an all-or-nothing landscape, where most edits are either neutral or fail all of their covering tests. Given the previous success of Metallaxis and MUSE, we believe that this all-or-nothing behaviour may be partly responsible for the lack of improvement. Alternatively, it may be that the results observed by Metallaxis and MUSE fail to scale to large real-world programs. In future work, we intend to investigate the assumptions behind these techniques more deeply.

To benefit from the knowledge of its mutants test case results, we believe a set of more finely grained mutation operators are required; a requirement that will most likely allow a larger number of bugs to be solved at the same time.

CHAPTER 5

Repair Model

Motivated by our findings and the findings of others—that the statement-level repair model used by GenProg is ineffective at finding repairs—in this chapter we conduct an empirical study of bugs in real-world C programs to determine a more effective repair model. Specifically, we explore the viability of extending the ideas of plastic surgery—using code from existing sources to craft the materials necessary for a repair—beyond the level of statements, and to a larger set of richer, more granular changes, capable of fixing a greater number of bugs.

Using a new bug fix mining tool, BugHunter, we automatically identify bug fixing commits within Git repositories, before extracting instances of AST-level repair actions and collecting a pool of donor code snippets from the program. Equipped with this information, we determine the fraction of bug fixes which involve a particular repair action, and the fraction of repair action instances that can be *grafted* [Barr et al., 2014] from existing code within the program. In an effort to reduce the search space, and backed by the findings of previous studies regarding the effectiveness of plastic surgery, we limit the membership of the donor code pool to snippets taken from the file where the fix occurred. To avoid the rejection of potential snippets due to differently labelled variables, we also explore the effectiveness of plastic surgery when such labels are removed; we refer to the labelled and unlabelled forms of the donor pool as the *concrete* and *abstract* donor pools, respectively.

From analysis of the results, we find that more granular repair actions, at and below the level of statements, are better suited to plastic surgery than more block-level changes. By removing labels from donor code snippets and treating them as templates, we observe a substantial increase in graftability, rising from 0–58% to 16–94%. To fix a larger number of bugs, we suggest incorporating a sub-set of the most frequent repair actions into the repair model, and to use an abstract donor pool to craft repairs.

The contributions of this chapter are as follows:

- We present BugHunter, a repair action mining tool, capable of identifying bug fixing commits within Git repositories and discovering potential instances of automated repair actions at the AST-level.
- We build upon previous definitions of *repair models* [Martinez and Monperrus, 2013] and provide inference rules for a set of repair actions, used to perform the study.
- We examine the *frequency* of 23 repair actions, inspired by repair models used within existing repair techniques, across 10,000 identified bug fixes in 200

open-source C projects.

- We determine the *graftability* [Barr et al., 2014] of each of these repair actions within a set of labelled and unlabelled donor code pools.
- From the observed frequencies and graftabilities of repair actions and donor code pools, we make a number of suggestions for constructing a future repair model, capable of addressing a greater number of bugs.

The rest of this chapter is structured as follows: Section 5.1 gives a brief review of the related related literature. Section 5.2 elaborates on the motivation of this study and outlines our research questions. Section 5.3 discusses our methodology; Section 5.4 expands upon the definition of *repair models* and provides descriptions, in the form of inference rules, for each of the repair actions studied. Section 5.5 outlines our approach to mining repair action instances from real-world software projects. Section 5.6 presents and discusses the results of our study. Finally, Section 5.7 summarises our findings, provides suggestions for the construction of more effective repair models, and outlines future directions for research.

5.1. Related Work

In this section, we briefly discuss previous research of relevance to repair models and plastic surgery, as well as highlighting where our study differs and builds upon this body of work.

The Plastic Surgery Hypothesis

Barr et al. [2014] summarise the "plastic surgery hypothesis" as follows:

> Changes to a codebase contain snippets that *already exist in the codebase at the time* of the change, and these snippets can be *efficiently found* and exploited.

The plastic surgery hypothesis underlies various genetic-programming-based approaches to automated program repair, optimisation, and improvement, all of which use existing code to find solutions.

Given this definition, Barr et al. [2014] break down this hypothesis into two assumptions: (1) changes to the program are repetitive, relative to their parent, and that (2) this repetitiveness may be efficiently exploited to construct those changes. In particular, techniques such as GenProg rely on the existence of donor snippets within the current, buggy form of the program. To test the first assumption, they measure the *graftability* of 15,273 commits, taken from several large-scale Java projects. The graftability of a change is defined as the number of its snippets for which a matching snippet can be found within the search space. For the purposes of the study, source

code lines (with whitespace removed), rather than AST-level entities, are treated as snippets. Barr et al. [2014] measure this quantity across three different search spaces, each containing snippets taken from the following sources, respectively:

- The *parent* of the change
- All *non-parental ancestors* of the change
- The *most recent version of a foreign software project*

To test the validity of the second assumption—that the space of donor snippets can be efficiently explored—Barr et al. [2014] also measure the density of matching snippets within each space.

From the results of their analysis, Barr et al. [2014] found that, in most cases, donor grafts could be found within the current version of the program and that rarely was it necessary to search non-parental ancestors for the graft. Moreover, 30% of grafts could be found within the same file at which the human-written change occurred. These results support both the plastic surgery hypothesis and GenProg's interpretation of that hypothesis: restricting the attention of the search to snippets within the current versions of the faulty versions of the file allows a large number of grafts to be found much more efficiently.

Our study builds upon the work by Barr et al. [2014] by investigating redundancy at the level of program repair actions, rather than at the level of source code lines. We choose to study source code redundancy within the context of particular repair actions since this allows us to more accurately determine the effectiveness of plastic surgery when applied to program repair.

Empirical Inquiry into Redundancy Assumptions of APR

Martinez et al. [2014] investigate the underlying assumption of plastic-surgery driven APR techniques, such as GenProg, PAR, and SearchRepair: that the ingredients used by a fix already exist within the program. To investigate this assumption, the authors examine six open-source Java projects and determine the fraction of (version control) commits—including those which do not pertain to bug fixes—that are "temporally redundant". A commit is deemed temporally redundant if it can be composed in its entirety from code introduced by previous commits. The authors measure temporal redundancy at both the line-level and token-level, and within the same file (termed *local temporal redundancy*) and across all files (termed *global temporal redundancy*).

The results of the study demonstrate a stark contrast in temporal redundancy at the line-level and token-level: Between 2–17% of commits are temporally redundant at the line-level, whereas 8–52% are temporally redundant at the token-level. This finding suggests that more granular changes to the program are easier to compose from previous versions of the program, although this trade-off comes at the cost of a significantly increased search space.

From analysis of global and local redundancy, the authors find that between 8–29% of tokens can be found within previous versions of the same file, compared to 31–52% across previous versions all files. Importantly, the size of the local pool was between two-to-three orders of magnitude smaller than the global pool. The cost effectiveness of searching the donor pool lends support to GenProg's decision to restrict the composition of its donor statements to those within the files under repair.

Our study differs from that conducted by Martinez et al. [2014] in two important aspects: Firstly, we focus our attention on the effectiveness of plastic surgery within C programs, rather than Java programs. Secondly, we measure redundancy within the context of concrete repair actions, rather than generically measuring it at the line- or statement-level. For instance, we determine the graftability of a "Replace If Guard" action by measuring redundancy at an expression level.

Mining Software Repair Models

Martinez and Monperrus [2013] conduct an empirical study of the frequency of 41 different AST-level repair actions within real-world bug fixes in Java programs, and argue that not all probabilistic repair models are equally effective. As the dataset for their analysis, the authors automatically identify the sub-set of bug fixes in the CVS-Vintage dataset [Monperrus and Martinez, 2012]: a dataset containing 89,993 source-code versioning transactions across 14 open-source Java programs. To find AST-level changes for each fix, they employ ChangeDistiller [Gall et al., 2009], an AST differencing tool which describes modifications to ASTs using 41 different types of changes (e.g., "Statement Insertion", "Statement Update", "Statement Deletion", "Addition of final to class declaration").

Martinez and Monperrus [2013] look at the different outcomes produced by using different methods of bug identification. They find that the number of source code changes is a good predictor of whether the changes within that transaction differ to those of normal software evolution; transactions with fewer changes tend to be the most different. In contrast, when transactions were selected purely based on the presence of indicators (e.g., "bug" or "fix") within their associated messages, the observed distribution of repair action frequencies was almost identical to the distribution across all transactions, suggesting such measures are ineffective at isolating fixes to the source code.

When considering the set of transactions that contain only a single change—as reported by ChangeDistiller—the top five change types were as follows: Statement Update (38%), Add Function (14%), Condition Change (13%), Statement Insertion (12%), Statement Deletion (6%). Between them, these change types account for 83% of all changes.

The focus of Martinez and Monperrus [2013]'s study is on the frequency of particular repair actions within Java projects. In contrast, although we also measure the

frequency of repair actions—albeit it for C—our study is primarily focused on the effectiveness of plastic surgery in the context of APR.

Critical Review of PAR

As an alternative to GenProg's coarsely-grained repair model, Kim et al. [2013] propose a hand-crafted repair model, based on frequently observed bug fix patterns (e.g., insertion of a null check). They incorporate this repair model into PAR, an evolutionary program repair technique aimed at repairing Java bugs. Compared to an implementation of GenProg for Java, PAR was able repair more bugs (27 out of 119, vs. 16). The authors also demonstrate the acceptability of PAR's patches through the use of a human study.

Despite achieving impressive results—and an *ACM SIGSOFT Distinguished Paper Award*—both PAR and the methodology used in its evaluation have since been the subject of criticism by Monperrus [2014]. This criticism has focused on both the unbalanced composition of the bug scenario benchmark used in the evaluation, and the methodology used to conduct the human study. The latter is of relevance to this study. Principally, Monperrus [2014] highlights potential biases within the human study, which may lead to participants to judge the correctness of a patch based on its visual similarity to human-repaired bug fixes, rather than its semantics. He argues that conflating correctness with appearance may unfairly lead to the dismissal of more alien-looking but otherwise correct patches.

In this study, we estimate the utility of a repair action (i.e., its ability to generate repairs) by its frequency within a corpus of mined human-written bug fixes. In practice, this decision may not accurately reflect the utility of some repair actions. (e.g., it may be possible to fix a bug using a certain repair action, but such an action is unlikely to be performed by a human-programmer, owing to its aesthetics or other such non-functional properties.) Nonetheless, our results provide a reasonable approximation of utility. For a more detail discussion of both PAR and its critique, see Section 2.2.5.

Bug Fix Patterns in Java

In an effort towards realising a more effective repair model for Java programs, Soto et al. [2016] use software repository mining to determine the frequency—and to some extent, the composition—of certain bug fix patterns within human-written repairs for Java. As a corpus for their study, the authors use the publicly available *September 2015/GitHub* dataset, provided by Boa [Dyer et al., 2013], containing 4.5 million identified bug fixes in over 500,000 Java projects.

As the first part of their study, Soto et al. [2016] assess how many files are changed by each fix, where file insertions, deletions and modifications are all considered to be changes. Across all files types—including non-source code files—the authors observe

a median of 2 changes, and a surprisingly high mean of 11.3 changes, suggesting a long-tailed distribution. When only Java source code files are considered, the authors find that each bug fix changes a mean of 4.47 files—the median number of changes is omitted from the paper. Although this adjusted mean is still high, the relatively low—and more relevant—median suggests that most bug fixes involve changes to two files or fewer.

After investigating the number of files changed by each bug fix, Soto et al. [2016] look into the mean number of class, method, field, and variable introductions—high-level changes that are currently beyond the scope of all APR repair models. They find that, on average, each fix introduces 0.16 classes, 0.69 methods, 1.32 fields, and 0.20 variables (these figures fail to account for test fixtures, and should be considered an overestimate). Based on these findings, Soto et al. [2016] advocate for the inclusion of property introduction into (Java) repair models. For two reasons, we do not agree with this conclusion, nor with the methodology used to reach it:

- Within the context of learning suitable models for automated program repair, it does not make sense to base one's decisions on the mean number of introductions. This statistic is more susceptible to noise within the dataset—stemming from a risk of misclassification—and the effects of large-scale refactorings; Soto et al. [2016] do not report the standard deviation or interquartile range to determine this risk. Instead, one should investigate the more statistically robust median, which is better suited to asking the (more relevant) question: "How many properties do repairs tend to introduce?"
- Not all modifications made by developers are strictly necessary to fix the bug. In the process of bug fixing, developers will sometimes refactor the original code to improve its maintainability and reduce its complexity.

Following an investigation of these source-level feature introductions, the authors attempt to determine the frequency of PAR's repair templates within the corpus. Using Boa, Soto et al. [2016] look for instances of eight of the ten templates used by PAR: neither "Class Cast Checker" or "Expression Changer" patterns are investigated, due to technical limitations in Boa.

Under the most lenient assumption, that no two of PAR's patterns co-occur within a bug fix, these templates are found to cover 14.78% of buggy files. In contrast, in the dataset used to evaluate the effectiveness of PAR, these repair templates cover 30% of the corpus. This discrepancy suggests that the dataset used to evaluate PAR was unrepresentative; bugs that are fixable by PAR are encountered twice as often in PAR's dataset than they are in general.

Finally, the authors turn their attention to the statement-level repair model used by GenProg. After mining instances of GenProg's operators within the corpus, the authors determine the likelihood of a statement of a given type being replaced by that of another type. The results of this analysis demonstrate a non-uniform distribution, and that certain kinds of statement are highly unlikely to be used as a replacement (e.g., TypeDecl, Label, Do, Assert). The analysis exclusively considers

cases where a statement is replaced by a statement of another type. It may be more informative to the construction of a repair model to determine how often a statement is replaced by another of the same type.

From observations of the mined Delete and Insert operators, for each kind of statement, the authors determine the fraction of fixes that involve deleting, inserting or otherwise leave statements of that kind unchanged. The most frequently modified kind of statement, in terms of both insertion and deletion, was found to be Expression statements (e.g., function calls, assignments, casts). If, Return and For statements insertions and deletions were all found to occur in at least 15% of bug fixes. TypeDeclaration and Assert statements were found to have the lowest rates of insertion and deletion.

Although our study shares a similar motivation to that of Soto et al. [2016]'s—to find a more effective repair model to reduce the cost of APR—it aims to increase our understanding of repair models by considering the frequency of repair actions and their associated graftabilities.

Repair Quality

Recent studies by Qi et al. [2015] and Smith et al. [2015] demonstrate the tendency of APR techniques to yield plausible but incorrect patches when used with inadequate test suites. Qi et al. [2015] showed that the all of the fixes generated by GenProg that were reported in [Le Goues et al., 2012a] were either the result of overfitting, or could be realised through code deletion alone. This finding raises questions regarding the true effectiveness of GenProg's repair model. Firstly, it causes us to ask: "Are GenProg's statement-level repair actions sufficiently expressive to repair most bugs?" Secondly, it leads us to the question: "Do GenProg's assumptions regarding redundancy—i.e., that the materials required for the repair can be found within the same file as the fault—hold in practice?" The questions raised by this study serve as one of the driving motivations behind this study. By observing a large number of historical bug fixes, we hope to answer both of them.

For a more in-depth discussion of the issues of repair quality raised by Qi et al. [2015] and Smith et al. [2015], see Section 3.2.1.

Anti-Pattern Avoidance

To improve the quality of patches produced by existing repair techniques (specifically, GenProg and SPR), Tan et al. [2016] take an orthogonal approach to the problem by introducing the notion of *anti-patterns* into the search space. Anti-patterns are used to actively prune patches whose resulting changes to the control-flow graph of the program are deemed likely to yield an incorrect or incomplete program according to a set of heuristics. Each of the seven hand-crafted anti-patterns are briefly outlined below:

- **A1 - Anti-delete CFG exit node:** prevents the removal of return statements, exit calls, assertions, and calls to functions containing the word "error".
- **A2 - Anti-delete Control Statement:** prevents the deletion of control-flow statements (e.g., if-statements, switch-statements and loops).
- **A3 - Anti-delete Single-statement CFG:** prevents statement deletion within CFG nodes that contain only a single statement.
- **A4 - Anti-delete Set-Before-If:** prohibits the deletion of a variable definition if that definition is immediately followed by an if-statement using the defined variable.
- **A5 - Anti-delete Loop-Counter Update:** forbids the deletion of an assignment statement contained within a loop if the set of variables on the LHS of the assignment intersect with the variables referenced by the loop invariant.
- **A6 - Anti-append Early Exit:** prevents the insertion of return and goto statements at all locations within the program, except for immediately after the last statement within a CFG node.
- **A7 - Anti-append Trivial Conditions:** prevents the insertion of if-conditions that can be shown to be either tautological or fallacious following static analysis of the program.

Motivated by a manual inspection of the patches generated by GenProg, AE, and SPR on the ManyBugs dataset, these anti-patterns are designed to address specific weaknesses that are exploited in producing incorrect patches:

- **A1 and A6** are designed to overcome *susceptibility to weak test oracles* (i.e., those which only check the exit status of the program execution).
- **A2, A3 and A4** are designed to compensate for *inadequate test coverage*, which may otherwise cause important, but untested functionality to be destroyed.
- **A4** is intended to prevent *existing vulnerabilities, exposed by the test suite, from being masked*, instead of being directly addressed.
- **A5** is intended to prevent the *occurrence of infinite loops* which significantly hinder the efficiency of the search, and in the worst case, may be accepted as correct patches by test suites that test for the absence of a particular bug.
- **A7** aims to inhibit the implicit removal of functionality (observed when using tools such as SPR) through the insertion of a tautological or fallacious condition.

To demonstrate the effectiveness of these anti-patterns, Tan et al. [2016] produced modified versions of GenProg and SPR incorporating these rules, termed mGenProg and mSPR, respectively, and evaluated them against a sub-set of the CoREBench and ManyBugs datasets, using 86 bugs taken from 12 programs.

- Results demonstrated a 41% reduction of the search space for mGenProg, and a 27% reduction for mSPR, yielding 1.39x and 1.78x speed-up, as measured by the average wall-clock time taken to find a plausible repair.
- mSPR yielded fewer plausible repairs than SPR in most cases, however, the omitted repairs were those (correctly) identified as erroneous by the anti-patterns. By producing fewer repairs, mSPR reduces the burden of manually inspection placed upon the user.
- Both mGenProg and mSPR removed fewer lines of code than their unmodified variants, demonstrating a lower rate of functionality deletion.
- In cases where the repair produced was incorrect, mGenProg and mSPR were able to localise the fault to a single statement within the program, substantially reducing the complexity of debugging for the user.

Although the incorporation of these anti-patterns into both GenProg and SPR improved both the efficiency of the tool and its ability to localise the fault, neither saw a considerable increase in the number of correct patches that were produced. For mSPR, a single additional, correct repair was found, increasing the total number of correct repairs up from 12 to 13 (out of 86). In the case of GenProg, mGenProg found one fewer repairs, reducing the number of correct repairs found from 3 to 2. Given the significiant space reduction achieved by mGenProg, these results seem to suggest that GenProg's repair model lacks the complexity required to tackle a greater number of bugs.

Furthermore, although the results from this study are encouraging, one may question whether the techniques generalise beyond the relatively small set of benchmarks used. Since the dataset used to perform the experiment was the same from which the anti-patterns draw their motivation, the results may be subject to bias.

Despite questions over its generality, and the lack of any significant improvement in the number of correct repairs found (which is arguably a reflection of the limitations of the underlying repair model), anti-patterns provide a cost-effective means of reducing the search space, increasing efficiency, and improving fault localisation. As repair models grow in size and complexity, such techniques may prove crucial in tackling the inevitable explosion of the search space.

5.2. Motivation for Study

To avoid producing low quality patches, and to increase the number of correct repairs produced by search-based repair techniques, we must use alternative, richer repair models. Innovations from previous works can be used to increase the likelihood of selecting a correct repair over a feasible, but incorrect one—specifically, the fix prediction model used by Prophet [Long and Rinard, 2016]—in order to bene-

fit from these additions, we must possess a repair model capable of constructing a viable repair to begin with.

One way to tackle this problem is through the introduction of human repair templates, as proposed in PAR [Kim et al., 2013]. The sourcing of these templates remains a problem, however. It is also unclear how well these templates would apply to bugs beyond those encountered in the benchmark. From observation, one sees that the majority of (publicly visible) repairs for large-scale, open-source programs, do not represent the common programmer errors encoded by these templates; rather, although certain bug fixes may share certain similarities in their semantics, most are syntactically unique, and do not represent instances of common programmer error.

Alternatively, one could take a similar approach to most genetic programming systems beyond the domain of automated repair, and generate code for insertion from scratch, rather than copying it from the program under repair. However, whilst generating fragments of code from scratch may be acceptable for synthesis-based approaches, the resulting search space would almost certainly prove intractable when using search-based repair.

Therefore, plastic surgery presents a promising way of reducing the size of the search space to a more tractable one, and gives us a more general means of constructing repairs, than relying on the utility of manually constructed repair templates. Despite previous studies demonstrating the effectiveness of the plastic surgery [Barr et al., 2014; Martinez et al., 2014], its particular realisation within the repair model of GenProg has been shown to struggle to discover a correct repair for most problems to which it has been applied, as both ourselves and others have found [Qi et al., 2015]. (For more details, see Section 3.2.1.)

For the remainder of this chapter, the effectiveness of the plastic surgery within the domain of automated repair is explored at finer levels of granularity, beyond the coarsely-grained statement-level repair model used by GenProg. Additionally, the benefit of incorporating a number of different repair actions into the repair model is explored in tandem.

Specifically, we ask the following research questions:

- **RQ2:** Is plastic surgery equally effective for all repair actions?
- **RQ3:** Can the effectiveness of plastic surgery be increased through the use of unlabelled code snippets?

5.3. Methodology

Traditionally, when comparing repair techniques, one would make use of an existing set of pre-processed bugs from the literature, such as ManyBugs or GenProg's

TSE 2012 benchmarks. The efficacy of repair actions would then be evaluated by determining the number of bugs for which a repair can be found within a fixed resource window [Le Goues et al., 2012a; Long and Rinard, 2015; Mechtaev et al., 2016]. However, such an approach is inappropriate when assessing repair models, for a number of reasons:

1. **Diversity:** To produce a fair comparison of techniques, one must subject the repair models to a large sample of bugs, taken from a diversity of programs, in order to minimise the effects of sampling errors, and to mitigate the potential for any bias. At present time, the only publicly available benchmark suites for automated program repair presented in the literature (ManyBugs, IntroClass, GenProg TSE 2012, Defects4J, SPR) consist of at most several hundred bugs, across fewer than 15 different programs. Furthermore, the composition of these benchmarks is unlikely to be truly random, and may be subject to unintentional selection biases; for example, one may omit bug scenarios that are particularly expensive or difficult to replicate, such as bugs within the Linux kernel.

2. **Overfitting:** Operating on a relatively small set of bugs allows one to unintentionally overfit to the specifics of that sample. By analysing the composition of the corpus, one may quite simply introduce a number of specific repair actions, tailored to fix certain bugs. Such criticism may be levelled at the approach taken by PAR to the design and evaluation of its repair actions, each of which bear the watermarks of solving particular bugs from the benchmark it was tested on. By evaluating its repair model on such a small and well-studied set of benchmarks, we are unable to determine whether the repair model continues to hold the same utility in a more general context.

   A potential solution to this problem of overfitting would be to use separate datasets for training (i.e., learning a suitable repair model) and testing (i.e., assessing the generality of the learned repair model). To avoid misleading results, such a dataset would need to represent a truly random, uniform sampling of real-world bug scenarios. To our knowledge, no such dataset is publicly available. Existing real-world datasets are either hand-picked (e.g., ManyBugs), and thus subject to selection bias, or represent a narrow category of bugs (e.g., IntroClass contains bugs in simple programming assignments). Given the costs and complexities associated with sourcing and evaluating a suitable dataset, the only feasible way to assess the effectiveness of a repair model, *in general*, is through the use of bug fix mining.

3. **Repair Quality:** Instead of determining the ability of repair actions to aid in the construction of high-quality fixes, one may unintentionally end up finding the repair actions which best exploit weaknesses in the test suites used by each of the bug scenarios. For example, one may trivially introduce an "Append Exit" repair action, which appends a statement containing `exit(0);` after the selected statement. For many of the bugs within the TSE benchmarks, and a sub-set of the ManyBugs scenarios, this would yield an acceptable fix, as only

the exit status of the program is checked, rather than its outputs. (For more details, see Section 3.2.1).

4. **Feasibility:** Even if one were to possess a sufficiently rich variety of bugs, evaluating the effectiveness of each repair action by using each of them to perform search may take prohibitively long. One could sample a sub-set of the candidate fixes generated by each repair action to reduce the running time, but doing so will degrade the accuracy of the evaluation, and ties the performance of the repair actions to the underlying search technique.

Given the difficulties involved in this approach, we opt to assess the prevalence and graftability of repair actions through software repository mining instead, allowing a far larger corpus to be analysed, without the need for expensive test suite evaluations. The steps of our analysis are as follows:

1. A corpus of human bug fixes is mined from over 200 of the most popular repositories on GitHub containing C source files, using a custom-written, open-source repository mining tool, BugHunter.
2. A set of abstract syntax trees (ASTs) and AST differences is computed for each of the files modified by each fix, using GumTree [Falleri et al., 2014].
3. From these ASTs and their associated differences, a set of repair action instances are mined using the detection rules.
4. For each modified AST, a series of abstract and concrete donor pools are generated from its contents. Using these repair pools, we determine the graftability of each of the proposed repair actions.

Trade-Offs

Although this approach overcomes the identified problems involved in evaluating repair actions by incorporating them into the search procedure, it also comes with its own set of trade-offs. These trade-offs, and the steps taken to minimise their effects upon the results of the analysis, are as follows:

- **Precision:** Although this approach allows us to evaluate repair actions across a much larger corpus of bugs, it comes with the drawback that it excludes correct repairs that are not syntactically equivalent to the human bug fix. To partially mitigate this problem, we check whether the human repair is of the same kind as the action (e.g., does it modify an `if` guard?) in addition to checking whether the *exact* repair can be crafted from the elements of the program. This step also allows us to reason about the effectiveness of synthesis-driven automated repair approaches, and generative repair models, more like those from traditional applications of genetic programming.
- **Irrelevant Changes:** Human repairs may often contain modifications to the source code that are irrelevant to the bug fix. These changes might include

aesthetic or structural changes, or they may have been bundled together with the bug fix commit. In theory, one could alleviate this problem by using delta-debugging to minimise the AST difference. In practice, however, there may be no (readily accessible) test suite, and so this technique is unusable. Instead, in this study, only bug fixes pertaining to a single file and function are used to perform the analysis of repair actions.

Although the results of this analysis are likely to underestimate the utility of certain repair actions, the overall results help us to understand the composition of human repairs, the effectiveness of the plastic surgery hypothesis, and how we might go about designing a more effective repair model.

5.4. Repair Model

In this section, we provide a more rigorous definition of "repair models" building upon a previous, informal definition given by Martinez and Monperrus [2013]. Following this definition and a brief discussion of its related terms, we propose a series of 23 repair actions, inspired by the plastic surgery hypothesis and existing repair models.

Definitions

The term "repair model" appears to have been introduced by Martinez and Monperrus [2013] in an empirical study of search spaces within APR problems. They provide the following informal definitions:

- a *repair action* is defined as a "kind of modification on source code that is made to fix bugs", such as "changing the initialisation of a variable" or "adding a method call".
- a *repair model* is defined as a set of repair actions that are used to construct repairs.
- a *repair* is defined as a concrete application (or instance) of a particular repair action to source code, such as "adding the method call *fun(x)*."

We further refine the definition provided by Martinez and Monperrus [2013]:

Host Program, P: is the program subject to repair, whose behaviour is defined to be incorrect (faulty) with respect to a given test suite T. P is represented by the set of abstract syntax trees belonging to the source code files under repair. For the sake of brevity, $x \in P$ is taken to refer to the existence of an AST node x within P.

Editable Locations, L: the locations within the program at which a repair action may be applied. For the majority of repair systems, L is comprised of the sub-set of statements within P that are covered by at least one failing test case.

Fault Localisation, $\mu : L \to \mathbb{R}$: the *suspiciousness* of each editable location, $\mu_l \geq 0$, where larger values of μ_l imply a higher degree of suspicion, and $\mu_l = 0$ indicates the belief that l does not contain a fault.

Donor Pool, $\mathbb{D}$: the set of AST sub-trees that may be combined with a repair action to construct an edit.

Repair Actions, $a \in A$: are used to describe a kind of AST transformation that may be performed. Each repair action is described using the form described in Equation 5.1.

$$\frac{statement(s_1) \qquad s_1 \neq [] \qquad s_2 \neq [] \qquad s_2 \in \mathbb{D}}{s_1 \to [s_1; s_2]} \tag{5.1}$$

To the left of the *transformation arrow*, below the horizontal line, is the *matching signature*; this uses the BNF for the language of the host program to specify the shape of AST nodes that may be subject to this repair action, and to capture their contents. In the example above, the matching signature is simply s_1, allowing the action to be applied to any AST node (before consideration of the rest of the rule).

The right-hand side of the transformation arrow, is used to specify the shape of the node that the left-hand side should be transformed into. In the given example, the right-hand side $[s_1; s_2]$ states that the matching statement should be replaced by a block containing the matching statement and another statement, s_2.

Finally, the clauses above the horizontal line describe the *transformation criteria* that must be satisfied by all instances of this repair action. In the example above, the transformation criteria states that the matching node should not be an empty block, and that the second statement of the right-hand-side s_2 should be a non-empty block, contained within the donor pool.

Edit, e: represents a particular application of a repair action by specifying the matching node within the host program, together with the values of each of the free variables (e.g., s_2 in the previous example).

Edit Space, E: defines the set of all edits that may be applied to P.

Edit Localisation, $\omega : E \mapsto \mathbb{R}$: is used to assign a weight to each edit, $\omega_e \geq 0$, in a similar fashion to fault localisation. In this case, larger weights are used to encode a belief that a particular edit is more likely to be correct.

Repair, R: represents a candidate repair as a set of non-overlapping edits, which yield the program P' when applied to P.

For the sake of our analysis, and in the interest of bounding the size of the repair space, the non-overlapping constraint is strengthened, such that no two edits within a given patch may be applied at the same statement:

$$\forall e_1, e_2 \in R \mid e_1 \neq e_2 \implies \text{nearestStmt}(e_1) \neq \text{nearestStmt}(e_2)$$

From observation, most human repairs may still be expressed when this constraint is applied.

Donor Pool

Associated with each repair model is a *donor pool*, containing the fragments of code (i.e., AST subtrees) from which repairs may be crafted. In the case of synthesis-based or generative repair models, such as those used by SPR, PAR, and NOPOL, this donor pool is implicit, and is lazily generated during the synthesis (or generation) procedure. For tools relying on the plastic surgery hypothesis, such as GenProg and SearchRepair, the donor pool explicitly describes the fragments that may be used to form repairs.

Typically, the contents of the donor pool are sourced from the program under repair, as is the case with AE, GenProg, Prophet, and others; however, foreign programs may also be used to construct the donor pool, as demonstrated by SearchRepair [Ke et al., 2015].

In either case, one must usually perform optimisations in order to reduce the size of the donor pool to a more scalable one. Whilst these techniques, described below, may lead to efficiency gains, they do so at the cost of potentially solving fewer bugs.

- In the case of GenProg (and its variants), the donor pool is composed from the sub-set of files that are subject to the repair, rather than the entire program. Previous studies [Barr et al., 2014; Martinez et al., 2014] have shown that this restriction causes a minimal reduction in the number of solvable problems. However, these studies failed to account for the particulars of any repair model, potentially reporting a higher number of successes than would otherwise be possible.
- GenProg further restricts the donor pool to only contain code fragments that are executed by at least one positive test case.
- AE reduces the number of redundant entries in the donor pool by treating it as a set, rather than a collection, removing all duplicates of syntactically equivalent statements.
- Rather than restricting the donor pool to the contents of the file under repair, Prophet retains the n-most executed statements from the entire program

within its donor pool.

One may also reduce the donor pool further still, through the introduction of compiler optimisation techniques—similar to those used by AE to prune edits from the search space—such as canonicalisation and constant folding.

For each of the reviewed repair techniques which rely upon plastic-surgery, the donor pool is used only at the statement-level, and so, it solely consists of statements from the program. Here, we extend the donor pool, allowing it to operate with a number of finer-grained repair actions, by introducing sub-pools:

- $\mathbb{D}_{stmt}$ containing all the statements in the program.
- $\mathbb{D}_{block}$ containing all control-flow blocks within the program, with the exception of `Switch` blocks.
- $\mathbb{D}_{exp}$ containing all expressions within the program. In practice, one could further divide this sub-pool by type, allowing for more efficient repair action queries.
- $\mathbb{D}_{guard}$ containing all boolean expressions that are used as guards for all if-, while-, do-while- and for-statements within the program.
- $\mathbb{D}_{lhs}$ contains all of the left-hand side expressions of each assignment within the program.
- $\mathbb{D}_{rhs}$ contains all of the right-hand side expressions of each assignment within the program.
- $\mathbb{D}_{arg}$ contains all of the function call arguments within the program.

Repair Actions

Having provided a definition of *repair models*, *repair actions*, and their associated terms, in this section we define a number of repair actions, inspired by existing repair techniques. Each repair action is tailored to exploit the plastic surgery hypothesis, in order to maintain a tractable edit space, through the incorporation of the extended donor pool, described in Section 5.4.2.

Below, we describe each of the 23 repair actions, using the repair action notation introduced in Section 5.4.1. To aid our descriptions, we introduce the following definitions:

- $[s_1;\ \ldots\ ; s_n]$ denotes a block of statements.
- $[]$ denotes the empty block.
- HasKind$(node, kind)$ determines whether a given AST node belongs to a specified kind (e.g., statement, expression, function call). A node may have multiple kinds.

Generic Statement Actions

This first group of repair actions is identical to the original actions used in the GenProg repair model, with the addition of empty statement checking, a prepend action, and the removal of the `Swap` action, present in earlier versions of GenProg.

- **Delete Statement:** removes a non-empty statement from the program.

$$\frac{s \neq []}{s \rightarrow []}$$

- **Append Statement:** appends a statement from the donor pool immediately after an existing statement.

$$\frac{s_1 \neq [] \qquad s_2 \neq [] \qquad s_2 \in \mathbb{D}}{s_1 \rightarrow [s_1; s_2]}$$

- **Prepend Statement:** clones a statement from the donor pool at random and prepends it immediately before an existing statement in the program.

$$\frac{s_1 \neq [] \qquad s_2 \neq [] \qquad s_2 \in \mathbb{D}}{s_1 \rightarrow [s_2; s_1]}$$

- **Replace Statement:** replaces an existing statement within the program with a randomly selected clone, taken from the donor pool of statements.

$$\frac{s \neq [] \qquad s' \neq [] \qquad s \neq s' \qquad s' \in \mathbb{D}}{s \rightarrow s'}$$

If-Statement-Related Actions

- **Wrap Statement:** replaces a compatible statement with an if-statement whose body contains the original statement, and whose condition is taken from the donor pool.

$$\frac{\begin{array}{c} s \in S \\ NC = \{\text{ForLoop}, \text{WhileLoop}, \text{DoWhileLoop}, \text{Switch}, \text{IfThen}\} \\ \forall k \in NC \mid \neg\text{HasKind}(s, k) \\ c \in \mathbb{D} \end{array}}{s \rightarrow \text{IfThen}(c, [s], [])}$$

- **Unwrap Statement:** replaces an if-statement that contains no else-branch by its then-branch.

$$\frac{else = [] \qquad then \neq []}{\text{IfThen}(c, then, else) \rightarrow then}$$

- **Replace If-Condition:** replaces the condition of an existing if-statement within the program with a boolean expression taken from the donor pool.

$$\frac{c' \in \mathbb{D} \qquad c \neq c'}{\text{IfThen}(c, then, else) \rightarrow \text{IfThen}(c', then, else)}$$

- **Replace Then-Branch:** replaces the then-branch of an existing if-statement with a (compatible) block taken from the donor pool.

$$\frac{then' \in \mathbb{D} \qquad then \neq then' \qquad then' \neq [] \qquad then' \neq else}{\text{If}(guard, then, else) \rightarrow \text{If}(guard, then', else)}$$

- **Replace Else-Branch:** replaces the else-branch of an existing if-statement with a (compatible) block taken from the donor pool.

$$\frac{else' \in \mathbb{D} \qquad else \neq [] \qquad else' \neq [] \qquad else' \neq then}{\text{If}(guard, then, else) \rightarrow \text{If}(guard, then, else')}$$

- **Remove Else-Branch:** removes the else-branch of an existing if-statement, provided that the else-branch does not contain an if-statement, and is not empty.

$$\frac{else \neq [] \qquad kind(else) = Block}{\text{If}(guard, then, else) \rightarrow \text{If}(guard, then, [])}$$

- **Insert Else-Branch:** clones an existing block from the donor pool, and inserts it into a given if-statement (which lacks an else-branch) as its else-branch.

$$\frac{else \in \mathbb{D} \qquad else \neq [] \qquad else \neq then}{\text{If}(guard, then, []) \rightarrow \text{If}(guard, then, else)}$$

- **Insert Else-If-Branch:** selects both a randomly-selected block and guard from the donor pool, before joining them into an if-statement, and replacing the empty else-branch of an if-statement with the created statement.

$$\frac{\begin{array}{c} g_2 \in \mathbb{D} \qquad b_2 \in \mathbb{D} \\ b_2 \neq [] \qquad b_2 \neq b_1 \qquad g_2 \neq g_1 \\ elif = \text{If}(g_2, b_2, []) \end{array}}{\text{If}(g_1, b_1, []) \rightarrow \text{If}(g_1, b_1, elif)}$$

- **Add Guard to Else-Branch:** adds a guard to an unguarded else-branch within the program.

$$\frac{else \neq [] \qquad \neg\text{HasKind}(else, \text{If}) \qquad g_2 \in \mathbb{D} \qquad else' = \text{If}(g_2, else, [])}{\text{If}(g_1, then, else) \rightarrow \text{If}(g_1, then, else')}$$

Switch-Related Actions

- **Replace Switch-Expression:** replaces the expression of a switch statement with compatible expression from elsewhere in the program.

$$\frac{exp' \in \mathbb{D} \qquad exp \neq exp'}{\text{Switch}(exp, block) \rightarrow \text{Switch}(exp', block)}$$

Loop-Related Actions

- **Replace Loop-Guard:** replaces the guard of an existing for-, while- or do-while-loop with a guard cloned from the donor pool.

$$\frac{g' \in \mathbb{D} \qquad g \neq g'}{\text{While}(g, do) \rightarrow \text{While}(g', do)}$$

$$\frac{g' \in \mathbb{D} \qquad g \neq g'}{\text{DoWhile}(g, do) \rightarrow \text{DoWhile}(g', do)}$$

$$\frac{g' \in \mathbb{D} \qquad g \neq g'}{\text{For}(init, g, incr, do) \rightarrow (init, g', incr, do)}$$

- **Replace Loop-Body:** replaces the body of an existing for-, while- or do-while-loop with a block cloned from the donor pool.

$$\frac{do' \in \mathbb{D} \qquad do \neq do'}{\text{While}(g, do) \rightarrow \text{While}(g, do')}$$

$$\frac{do' \in \mathbb{D} \qquad do \neq do'}{\text{DoWhile}(g, do) \rightarrow \text{DoWhile}(g, do')}$$

$$\frac{do' \in \mathbb{D} \qquad do \neq do'}{\text{For}(init, g, incr, do) \rightarrow \text{For}(init, g, incr, do')}$$

Assignment-Related Actions

- **Replace RHS of Assignment:** replaces the right-hand side of an assignment statement with a compatible expression, cloned from the donor pool.

$$\frac{rhs' \in \mathbb{D} \qquad rhs \neq rhs'}{\text{Assign}(lhs, op, rhs) \rightarrow \text{Assign}(lhs, op, rhs')}$$

- **Replace LHS of Assignment:** replaces the left-hand side of an assignment with a compatible left-hand side, cloned from the donor pool.

$$\frac{lhs' \in \mathbb{D} \qquad lhs \neq lhs'}{\text{Assign}(lhs, op, rhs) \rightarrow \text{Assign}(lhs', op, rhs)}$$

Function-Call-Related Actions

- **Replace Target of Function Call:**

$$\frac{lhs' \in \mathbb{D} \qquad lhs \neq lhs'}{\text{FunCall}(target, args) \rightarrow \text{FunCall}(target', args)}$$

- **Modify Arguments of Function Call:**

$$\frac{args' \in \mathbb{D} \qquad args \neq args'}{\text{FunCall}(target, args) \rightarrow \text{FunCall}(target, args')}$$

- **Insert Argument into Function Call:**

$$\frac{arg' \in \mathbb{D} \qquad args = l \oplus r \qquad args' = l \oplus (arg') \oplus r}{\text{FunCall}(target, args) \rightarrow \text{FunCall}(target, args')}$$

- **Replace Argument of Function Call:**

$$\frac{arg' \in \mathbb{D} \qquad args = l \oplus (arg) \oplus r \qquad args' = l \oplus (arg') \oplus r}{\text{FunCall}(target, args) \rightarrow \text{FunCall}(target, args')}$$

- **Remove Argument of Function Call:**

$$\frac{args = l \oplus (arg) \oplus r \qquad args' = l \oplus r}{\text{FunCall}(target, args) \rightarrow \text{FunCall}(target, args')}$$

5.5. Approach

Below, we briefly describe the steps used by our tool, BugHunter, to mine repair action instances in Git repositories:

1. **Repository Sourcing:** We use GitHub's[1] API to identify and download the top 200 most starred projects whose source code includes C.[2]. Notable projects include the Linux kernel, Redis, Git, PHP, Vim, tmux, CCV, and Curl.

2. **Bug Fix Detection:** We discover commits that are suspected to be bug fixes across all of the 200 Git repositories. Possible bug fixes are found by iterating over commit histories,[3] and identifying commits that satisfy all of the following criteria:

   (a) The commit message must contain at least one of the following words: bug(s), fix, fixed, fixes, fault(y), defect, repair(ed), patch(ed).

   (b) To avoid compilation-related source changes, or artefacts from unrelated version control actions, the commit message for a potential bug fix should not contain any of the following words: compile, compilation, merge, revert.

   (c) The commit for a potential bug fix must modify at least one `.c` source file, and should not modify any header files (i.e., `.h` files), nor should it modify the source code of files written in another language (that can be identified; e.g., `.java, .py, .cc`).

[1] https://github.com

[2] We use stars as a crowd-sourced indicator of the popularity of a project

[3] Using the open-source GitPython API: https://github.com/gitpython-developers/GitPython

(d) Any modified `.c` source code files within the commit of a potential bug fix must also exist in the previous version of the program (i.e., no new `.c` source files may be introduced).

(e) The commit does not delete any `.c` files.

Although there may be more sophisticated ways to detect bug fixing commits, we believe our approach is suitable for this study. Our relatively strict criteria is likely to result in a high number of false negatives, which should have little to no effect on its outcome. Importantly, these criteria should maintain a low false positive rate. That is, the commits within our dataset are likely to be bug fixes, but the size of the dataset may be much smaller than the set of all bug fixing commits. We address concerns of size by examining commits from 200 projects.

Rather than dealing with the overheads and limitations of the GitHub API, BugHunter downloads given Git repositories—which may be hosted at locations other than GitHub, if one so wishes—to the host machine and interacts with the repositories directly. This avoids the need to re-download the source code for a particular pair of commits, as the faulty and patched files can be quickly acquired using the `git branch` command.

In total, we collected 372,463 bug fixing commits. To reduce the costs of mining, we use a random sample of 10,000 commits for the subsequent stages of the mining process.

3. **AST and Edit Script Generation:** We use GumTree [Falleri et al., 2014] to generate the abstract syntax trees (ASTs) of the buggy and fixed versions of each modified file. We also use GumTree to compute an edit script between the two trees, describing a set of primitive operations required to transform the buggy AST into the fixed AST. Finally, we apply post-processing steps to these ASTs and edit scripts to ease their integration into later stages of the mining process. See Appendix B.1 for more details on this stage of the mining process.

4. **Donor Pool Extraction:** We compute the set of donor pools, outlined in Section 5.4.2, for each of the buggy ASTs. To reduce the memory burden of maintaining multiple donor pools across thousands of fixes, we hash code snippets and save them to disk.

5. **Repair Action Detection:** We use inference rules, described in Appendix B.2 to mine instances of the repair actions described in Section 5.4.3.

6. **Graft Detection:** We determine which of the mined repair action instances can be grafted from the contents of each donor pool.

5.6. Results

In this section we present the findings of our analysis. Firstly, we look at the frequency of each of the proposed actions, before determining the extent to which instances of these repair actions can be grafted from existing source code within the same file.

Repair Action Frequency

To determine the utility of each repair action outside of the context of plastic surgery, Table 5.1 shows the number of instances of each repair action that were found across each of the mined bug fixes, together with the number of bug fixes for which a repair action of that kind could have been used.

We find that statement modification is by far the most frequently-encountered repair action, used by over 73% of bug fixes. This result, and the relatively low rate of statement deletion, indicates that the majority of changes to the program occur *below* the statement-level. This finding confirms our hypothesis, that the majority of changes to the program occur below the statement-level (used by existing techniques, such as SPR and GenProg). To fix more bugs, we can either restrict the consideration of replacement statements to those that are structurally similar to the ones being replaced, or we can extend the repair model with actions that operate below the level of statements.

We also find that statement deletion occurs in almost a third of bug fixes, suggesting that attempts to remove this repair action in the hope of avoiding the low-quality repairs discovered within GenProg may be misguided and detrimental to the effectiveness of the search technique. The high frequency of statement deletion may also be (partly) due to the way that statement replacement events are handled (as separate deletion and insertion actions). Nevertheless, we find that statement insertion occurs in over half of bug fixes, lending further support to the statement-level repair model.

Upon first glance, the fraction of bug fixes that involve replacement of large blocks of code—such as if-branches and loop bodies—may seem particularly high. This number is likely due to the fact that we attempt to find all possible ways of interpreting a repair; a modification to a particular statement within a block may also be achieved by replacing that entire block with another that contains the necessary changes.

Amongst the least frequent repair actions ($< 2\%$), we find "Replace Assignment Operator", "Guard Else Branch", "Remove Call Argument", "Replace Switch Expression", "Insert Else-If Branch", "Unwrap Statement", and "Insert Else Branch". Although "Remove Call Argument" and "Unwrap Statement" are relatively rare, neither of these repair actions requires any donor code, and so they may be added to

Repair Action	Instances	Usage	%
Modify Statement	27,989	2,027	73.07
Insert Statement	9,749	1,510	54.43
Replace Then Branch	9,835	1,136	40.95
Modify Call	6,473	934	33.67
Delete Statement	6,702	915	32.98
Modify Assignment	5,869	843	30.39
Replace If Condition	1,946	648	23.36
Replace Call Argument	2,898	546	19.68
Replace Assignment RHS	1,668	535	19.29
Replace Loop Body	986	473	17.05
Replace Else Branch	3,481	408	14.71
Replace Call Target	828	201	7.25
Replace Assignment LHS	429	115	4.15
Wrap Statement	148	105	3.79
Insert Call Argument	388	82	2.96
Replace Loop Guard	134	69	2.49
Remove Else Branch	76	62	2.24
Insert Else Branch	54	45	1.62
Unwrap Statement	38	28	1.01
Insert Else-If Branch	19	17	0.61
Replace Switch Expression	21	16	0.58
Remove Call Argument	23	8	0.29
Guard Else Branch	2	2	0.07
Replace Assignment Operator	0	0	0.00

Table 5.1: The number of instances of each repair action discovered across each of the mined bugs, together the number (and percentage) of bugs that involve at least one repair action of that type.

a repair model to solve a small number of bugs at a relatively low additional cost. In contrast, each of the other repair actions may potentially have a large number of applicable potential snippets, and thus they may represent a poor value proposition within a context where compute resources prevent a large number of candidate repairs from being tested.

Repair Action Graftability

Having determined the frequency of repair actions, we now turn our attention to whether the snippets used within those repair actions can be discovered within the (buggy version of the) source code of the same file. Specifically, we measure the *graftability* of each of the different types of repair action (i.e., the fraction of the repair action instances whose materials can be found in the donor pool). Since the

	Graftability	
Type	**Concrete Pool**	**Abstract Pool**
Delete Statement	100.00%	100.00%
Guard Else Branch	0.00%	50.00%
Insert Call Argument	44.85%	88.92%
Insert Else Branch	20.37%	25.93%
Insert Else-If Branch	5.26%	15.79%
Insert Statement	27.17%	37.90%
Modify Assignment	58.00%	71.63%
Modify Call	4.55%	45.54%
Modify Call Arguments	4.72%	37.22%
Modify Statement	47.40%	57.54%
Remove Call Argument	100.00%	100.00%
Remove Else Branch	100.00%	100.00%
Replace Assignment LHS	17.25%	49.18%
Replace Assignment RHS	10.07%	45.50%
Replace Call Arg	22.60%	52.45%
Replace Call Target	18.84%	94.08%
Replace Else Branch	0.92%	37.92%
Replace If Condition	34.33%	55.65%
Replace Loop Body	1.11%	15.52%
Replace Loop Guard	12.69%	38.06%
Replace Switch Expression	9.52%	42.86%
Replace Then Branch	47.25%	51.46%
Unwrap Statement	100.00%	100.00%
Wrap Statement	35.81%	54.05%

Table 5.2: The graftability of each repair action in the contexts of the concrete pool, containing the unchanged snippets from the file under repair, and the abstract pool, containing the unlabelled forms of the snippets from the file under repair.

graftability of a repair action is dependent upon the contents of the donor pool, we measure graftability for both the *concrete* and *abstract* donor pools. The results of this analysis can be found in Table 5.2. Note, actions that purely involve deletion, such as "Delete Statement" and "Remove Else Branch", are completely graftable regardless of donor pool since they do not introduce code.

Concrete Donor Pool

Using the contents of the concrete pool, we find that between 0–58% of instances of particular repair action types can be grafted in their entirety. The large degree of variance in graftability between different kinds of repair action suggests that plastic surgery is more effective within the context of some kinds of repair action, and less

so in others.

At the statement level, we find that 27% of statement insertions may be grafted; a similar rate to previous studies [Barr et al., 2014; Martinez et al., 2014]. Encouragingly, we find that statement modification—the most frequently encountered repair action, at 73%—is the second-most graftable repair action with a graftability of 47%; this result suggests that the modified form of a statement are likely to already exist within the donor pool. We find that 58% of "Modify Assignment" actions are graftable, hinting that varying degrees of graftability may be seen between different kinds of statements; we leave investigation of this possibility, and how it may be harnessed by the repair model, to future work.

In general, we find that repair actions that accept a block of statements as their input are amongst the least graftable: "Replace Loop Body" (1%), "Replace Else Branch" (1%) and "Insert Else-If Branch" (5%). Two surprising exceptions to this rule are "Insert Else Branch" (20%) and "Replace Then Branch" (47%).

Below the statement level, we observe a graftability between 5% and 45%. Again, we observe substantial variance between the graftability of actions at this level. Amongst the most graftable repair actions are "Replace If Condition" (34%), "Wrap Statement" (36%), and "Insert Call Argument" (45%).

Abstract Donor Pool

Comparing the abstract pool to the concrete pool, we see a substantial increase in graftability, rising from 0–58% to 16–94%. These results show that snippets structurally identical to those used by the repair are likely to already exist, and as such, serve as an effective pool of donor code from which to automatically craft repairs. This increase in graftability is most prominent in block-level actions, that were amongst the most difficult to graft using the concrete donor pool: "Replace Loop Body" (1% to 16%), "Replace Else Branch" (1% to 38%), "Insert Else-If Branch" (5% to 16%). Likewise, "Modify Call" and "Modify Call Arguments" rise from 5% each to 46% and 37%, respectively. When one removes block-level repair actions from consideration, one finds that graftability increases from 16–94% to 37–94%—a reasonable figure when attempting to automatically generate repairs.

5.7. Discussion & Conclusion

Building upon previous work on plastic surgery and the redundancy assumptions of automated program repair, we find that the effectiveness of plastic surgery varies according to the particular repair actions to which it is applied. In general, we find that plastic surgery is most effective in the context of more granular repair actions, at and below the statement level, and less so at the block level. We find that our

odds of discovering a graft within the previous version of the file under repair are significantly enhanced when labels (i.e., variable and function names) are removed from consideration.

In Table 5.3, we summarise the findings of our study by giving the frequency and graftability of each of the proposed repair actions, together with a measure of its effectiveness, computed as the product of frequency and graftability. To achieve a balance between search efficiency and the number of bugs that can be solved by a given tool, we suggest that the search should devote the bulk of its attention towards repair actions that are both frequent and highly graftable. We also suggest swapping GenProg's "Replace Statement" operator with a "Modify Statement" operator, or otherwise tuning the "Replace Statement" operator to heavily favour the replacement of a statement with a structurally similar one.

Type	Frequency	Graftability	Effectiveness
Modify Statement	73.00%	57.54%	42.00%
Delete Statement	32.98%	100.00%	32.98%
Modify Assignment	30.39%	71.63%	21.77%
Replace Then Branch	40.95%	51.46%	21.07%
Insert Statement	54.43%	37.90%	20.63%
Remove Call Argument	19.68%	100.00%	19.68%
Modify Call	33.67%	45.54%	15.33%
Remove Else Branch	14.71%	100.00%	14.71%
Replace If Condition	23.36%	55.65%	13.00%
Replace Call Arg	19.68%	52.45%	10.32%
Replace Assignment RHS	19.29%	45.50%	8.78%
Modify Call Arguments	19.68%	37.22%	7.32%
Replace Call Target	7.25%	94.08%	6.82%
Replace Else Branch	14.71%	37.92%	5.58%
Replace Loop Body	17.05%	15.52%	2.65%
Insert Call Argument	2.96%	88.92%	2.63%
Wrap Statement	3.79%	54.05%	2.05%
Replace Assignment LHS	4.15%	49.18%	2.04%
Unwrap Statement	1.01%	100.00%	1.01%
Replace Loop Guard	2.49%	38.06%	0.95%
Insert Else Branch	1.62%	25.93%	0.42%
Replace Switch Expression	0.58%	42.86%	0.25%
Insert Else-If Branch	0.61%	15.79%	0.10%
Guard Else Branch	0.07%	50.00%	0.04%

Table 5.3: A summary of the frequency of each of the proposed repair actions, measured by the percentage of bugs in which it is encountered, together with the graftability of that repair action when the abstract pool is used. *Effectiveness*, computed as the product of *frequency* and *graftability*, estimates the fraction of bugs for which a given repair action may graft a repair.

RQ2: Is plastic surgery equally effective for all repair actions?

We discovered that plastic surgery is less effective when applied at the block level. This suggests that excluding such edits from the search space may increase efficiency with minimal compromise to repairability (i.e., the bugs that can be fixed).

We found that statement deletion is the second most common repair action, occurring in roughly a third of bug fixes. This result suggests that excluding statement deletion from the repair model to avoid overfitting [Long and Rinard, 2015; Qi et al., 2015] may be misguided and likely to reduce the ability to fix bugs.

Given the success of repair techniques which use repair actions below the statement-level, it is crucial that such actions be incorporated into scalable, search-based repair.

Our results demonstrate that plastic surgery can be successfully applied at this level of granularity—by doing so, we can allow search-based repair to address a greater number of bugs.

We also found that the majority of bug fixing commits occur below the statement-level. In 58% of cases, the modified variant of the statement could be found somewhere else in the program. This result could be used to increase the efficiency of program repair by focusing statement replacement on structurally similar snippets. This result builds on previous work by Soto et al. [2016], showing that certain kinds of statement are more likely to be replaced by certain kinds of donor statements (i.e., not all statement replacements are equally likely).

RQ3: Can the effectiveness of plastic surgery be increased through the use of unlabelled code snippets?

We found that the effectiveness of plastic surgery can be boosted significantly through the use of abstract donor pools. Our results suggest that incorporating abstract donor pools into repair techniques could provide a substantial increase in the number of fixable bugs.

Future Work

Below, we discuss a number of ways in which this analysis could be extended to gain knowledge about the likelihood and composition of repair actions; knowledge that may be leveraged to produce a more predictive, effective, and efficient repair model, allowing more bugs to solved in fewer candidate evaluations.

- **Limitations of Plastic Surgery:** We found that success of plastic surgery is dependent upon both the repair actions to which it is applied, and their associated level of granularity. Building on this, one could further explore a more detailed set of repair actions to determine whether type, or other attributes associated with code fragments affect graftability. In particular, it may be fruitful to explore the graftability of particular kinds of statements and expressions, and whether certain fragments are unlikely to be found within existing code (e.g., print statements and strings). By identifying the strengths and weaknesses of plastic surgery, we can tune our repair model accordingly, and prune certain portions of the search landscape, leading to a more efficient search process.

- **Domain-Specific Languages:** Although we use inference rules to formalise the semantics of repair actions within this analysis, further work could be performed to allow a compact domain-specific language to be used to describe (and implement) repair actions (in terms of constrained source code transformations). Beyond improvement to software quality, this addition would allow potential repair actions to be machine generated, rather than being translated

by hand, opening up the possibility of using evolutionary computation and knowledge discovery to construct and evaluate novel repair actions.

- **Graft Localisation:** Although we may use the frequency of repair actions to generate a more effective probabilistic repair model, this step only allows us to differentiate between repair actions, rather than candidate grafts. One solution to this would be to incorporate an additional layer into the probabilistic model, describing the likely type of the donor code snippet. Soto et al. [2016] achieve this by looking at which kind of statement is most likely to replace a statement of a given kind. Soto et al. [2016]'s approach could be further extending to statement insertions, or it could use other attributes of a snippet, such as its size or complexity.

  More generally, it may be productive to apply machine learning to the problem of graft selection—for both automated repair and otherwise. Ideally, such an approach would allow us to rank grafts by their likelihood. In addition to potentially improving the quality of repairs and the efficiency of the search process, this step would allow us to remove the non-deterministic aspects of the search with minimal impact to results, eliminating the costly need to gauge performance over a number of repeated costs.

- **Discovery of Repair Actions:** Instead of mining a pre-specified set of repair actions, one may attempt to discover common, predictive repair actions within the corpus, through the application of machine learning and graph mining techniques. In particular, one may combine a variant of the LASE [Meng et al., 2013], a method for discovering context-aware, systematic edits, with BugHunter to find (and constrain) common repair actions across a wide and diverse corpus of programs.

- **Bug-Fix Detection and Categorisation:** To identify a larger number of bug-fixing commits, and to reduce the number of false positives, one could explore the use of more sophisticated bug fix identification criteria. Whilst incorporating such measures would be unlikely to significantly alter the outcome of the research questions posed in this chapter, it may allow BugHunter to be used to probe the composition of specific repair actions, allowing the findings to be exploited to produce a more effective repair technique.

  By improving bug fix detection accuracy, one may use existing techniques [Thung et al., 2012] to predict the category of bug under repair, potentially allowing the exploitation of that information to suggest which repair actions are most likely to be used and where the fix is most likely to reside.

CHAPTER 6

Search

Where search-based program repair techniques such as GenProg, HDRepair, and PAR are designed to be capable of multiple-edit repair, rarely is this ability used in practice. Underlying all of these techniques is a common search algorithm based on genetic programming. HDRepair and PAR can be viewed as extensions to the original GenProg system. Despite possessing the ability to construct multiple-line patches, in practice, the majority of patches generated by GenProg can be reduced to a single edit [Qi et al., 2015]. Moreover, almost all of the *plausible* patches generated by GenProg on the ManyBugs dataset have been shown to be equivalent to functionality deletion [Qi et al., 2015]; only 2 of the 55 reported bug fixes are correct, with respect to the intended program semantics. Whilst these results suggest that GenProg, as a whole, struggles to craft repairs, it is unclear whether this is due to an inadequate repair model, a poor search algorithm, or both.

Since the introduction of GenProg, work on search-based program repair has focused primarily on the efficiency of the repair process, to the detriment of scalability (i.e., the ability to repair multiple-line bugs). We briefly discuss three subsequent search based repair techniques with alternative search algorithms:

- AE, proposed by a sub-set of GenProg's authors, replaces GenProg's search algorithm with an exhaustive search. By default, this search is conducted within a single-edit search space. This search space is obtained by discovering, and discarding, a sub-set of provably equivalent patches, through the use of various compiler optimisations and program analysis techniques. Furthermore, by treating the search process as a decision problem (i.e., *is this patch correct?*), rather than an optimisation one (i.e., *how close is this patch to being correct?*), AE uses test case prioritisation, and regression test selection techniques to reduce the expected number of test case evaluations.[1]

- RSRepair, a variant of GenProg, showed that a form of random search attained a higher efficiency and success rate than GenProg's genetic algorithm [Qi et al., 2014]. However, RSRepair constrains itself to a single-edit search space, and the ManyBugs dataset was used to conduct its evaluation. Thus, it may be that RSRepair simply converges to a plausible but incorrect solution faster than GenProg does. Authors have incorrectly cited this study as evidence that GenProg is merely equivalent to a less efficient form of random search [Long and Rinard, 2015, 2016]. Whilst this *may* be the case, the evaluation of RSRepair does nothing to prove this, nor does it claim to. Rather,

[1] Note, regression test selection, unlike test prioritisation, could be used by GenProg to reduce the test suite for a candidate patch.

RSRepair shows the effectiveness of using test case prioritisation to reduce the cost of finding single-edit patches.

- SPR introduces both a new, substantially larger repair model, intentionally lacking a *statement deletion* repair action, and a new search algorithm, used for a sub-set of its patches, known as *value search*. SPR manages to find correct patches for 16 bug scenarios within the ManyBugs dataset, compared to 2 found by GenProg. Despite the intentional exclusion of explicit functionality deletion, Mechtaev et al. [2016] highlight that SPR implicitly generates many such patches through the introduction of tautological and contradictory branch conditions [Mechtaev et al., 2016]. For the one of the `libtiff` subjects within ManyBugs, 80% of SPR's reported patches were found to be functionality deleting.

  To accommodate the increased search space accompanying its enriched repair model, SPR uses value search to soundly reduce the number of patch evaluations. Instead of validating concrete patches, value search first determines whether a set of outcomes exists for a given branch condition that would cause the test suite to pass. If, and when such a set of outcomes is found, SPR utilises value search to synthesise a suitable branch condition, using its knowledge of the program state, and intended outcome at each evaluation of the branch condition.

Although all of these techniques outperform GenProg in terms of efficiency, and in some cases, effectiveness (i.e., the number of bugs for which it can find a repair), none is capable, by default, of producing multiple-edit patches. In other words, since GenProg, researchers have proposed significantly more efficient techniques for finding single-edit repairs, but none have proposed better (search-based) techniques for generating multiple-edit patches.

SPR, RSRepair, and AE demonstrate that treating program repair as a decision problem, rather than an optimisation problem, is an effective strategy for decreasing the cost of finding single-edit patches. This treatment, however, is not scalable; the size of the search space increases exponentially with respect to the size of the patch. Even if decision techniques can substantially reduce the cost of evaluating a single candidate, the combinatorial number of patches renders this advantage, more or less, useless. If search-based program repair is to scale to multiple lines, active search algorithms capable of discovering partial fixes, and promising repair locations will be needed.

Motivated by this need for better search algorithms for multiple-edit repair, in this chapter, we conduct a theoretical and empirical analysis of GenProg's search technique—since GenProg is the only search based technique capable of multiple-edit repair—in order to gain a clearer understanding of its strengths and weaknesses. Based on our findings, we propose and demonstrate an alternative search technique better suited to the nuances of program repair, inspired by a *greedy algorithm*.

In summary, the main contributions of this chapter are as follows:

- We conduct a theoretical analysis of GenProg's search algorithm, identifying its underlying assumptions and a number of potential weaknesses.
- Based on our theoretical analysis, we conduct an empirical analysis to test our theories and to determine the contribution of fitness to the success of the search.
- Motivated by the findings of these analyses, we propose and implement an alternative search technique, capable of multi-edit repair, based on a greedy algorithm.
- We demonstrate that our proposed technique outperforms GenProg's search algorithm across a number of performance metrics.

The remainder of the chapter is structured as follows: Section 6.1 provides a review of the related literature. Sections 6.2 and 6.3 conduct theoretical and empirical analyses of GenProg's search algorithm, respectively. Section 6.4 introduces and evaluates an alternative search algorithm for program repair, inspired by greedy algorithms. Section 6.5 outlines directions for future work. Finally, Section 6.6 summarises the findings of this chapter and provides concluding remarks.

6.1. Related Work

In this section, we briefly discuss previous and related work concerning the performance of genetic algorithms for automated program repair, as well as relevant studies on the matter of search landscapes for programs.

Random Search for Program Repair—RSRepair

In an effort to gauge the effectiveness of GenProg's search algorithm, Qi et al. [2014] explored the effects of replacing GenProg's genetic algorithm with a random search. The authors found that, over 100 runs on a sub-set of the ManyBugs benchmarks, their variant of random search, RSRepair, achieved both a higher success rate, and a lower mean number of test case evaluations than GenProg on 24 out of 25 problems. For more technical details on RSRepair, see Section 2.2.3.

Although these results demonstrate the (relative) effectiveness of random search compared to genetic algorithms for the purposes of program repair, they do not necessarily show that genetic algorithms are detrimental to the search. Importantly, RSRepair is able to use test case prioritisation and to terminate on the first instance of failure within the test suite, whereas GenProg must evaluate a fixed number of tests in order to produce a fitness value, regardless of their outcomes. As such, RSRepair is able to achieve a far higher test case efficiency than GenProg; in most cases, the candidate repair fails the first negative test case. In addition to evaluating

candidates in a different way, RSRepair also operates in a single-edit search space, giving it a distinct advantage over GenProg, which considers patches of an arbitary length.

For repairs that require only a single edit, there is little room for genetic algorithms to compose a solution, thus leading to a partial explanation of the strength of RSRepair. Conversely, problems requiring multiple edits cause us to question whether genetic algorithms are capable of identifying and combining partial solutions—no studies have investigated this assumption.

Representation and Operators

Le Goues et al. [2012c] conducted a study of the impact of various parameter choices on the efficiency of GenProg and the total number of bugs it is able to repair (within a fixed window of time). Specifically, the authors considered the representation used, the choice of crossover operator, the probabilities associated with the selection of each mutation operator, and the weighting factor used by the fault localisation to bias the search towards mutation of statements executed solely by the failing test cases. Through their investigation, the authors discovered:

- The use of the `Patch` representation, in which individuals are represented as a sequence of edit operators, yielded a higher success rate than the original `AST/WP` representation.
- The time taken to find a repair (when successful) was lowest when no crossover operator was used, although this resulted in the worst success rate (54.4%) of the options studied. Of the two remaining options, one-point crossover and sub-set crossover, one-point crossover was found to have the higher success rate (65.2% vs. 61.1%), and the lower number of on-average fitness evaluations (118.20 vs. 163.05).
- From observation of the composition of (Insert, Delete, Replace) operations within the minimised form of (plausible) repairs generated by GenProg, it was found that each of these edit types appeared in 13%, 51%, and 57% of repairs, respectively. In contrast, the associated weightings for each mutation operator cause the different types of edit to appear with equal frequency, despite the prevalence of deletion and replacement edits.
- The authors also measured the proportion of repairs that exclusively modified statements executed by the negative test cases to the repairs that modified statements executed by both the positive and negative test cases. Only 35% of modified statements were executed exclusively by the negative tests, despite the fact that GenProg's default fault localisation weightings attribute 90% of modifications to statements executed solely by the negative tests.

Based on the results of the investigation, Le Goues et al. [2012c] changed the parameters used by GenProg, and evaluated the resulting algorithm on a set of 105

bugs (which now forms a sub-set of the ManyBugs benchmarks). These changes allowed GenProg to repair 5 additional bugs, and decreased the time taken to find a repair by 17-43% for some of the more difficult bug scenarios. Qi et al. [2015] found that, of the 55 bugs that were reported to have been repaired by GenProg, only 2 were correctly repaired. In most cases, patch evaluation only considered the exit status of the program and not its outputs, leading to patches that inserted code such as `exit(0);` to be accepted. The findings of that study cast uncertainty on any conclusions made regarding the effectiveness and efficiency of GenProg's search algorithm on the basis of inadequate test suites. Results reported to increase the success rate or reduce the cost of finding a repair may in fact be pushing the search towards areas likely to contain a plausible but incorrect repair.

Predicate-based Fitness Functions

Fast et al. [2010] propose an alternative fitness function for GenProg based on the similarity of learned program invariants—logical assertions at points within a program that hold true over an associated set of executions, also referred to as predicates—between the original program and candidate solutions. For example, an invariant may specify that an integer argument x, belonging to some given function, is always non-negative (i.e., $x \geq 0$ holds over all observed executions).

Fast et al. [2010] use Daikon [Ernst et al., 2007], a popular, off-the-shelf invariant mining tool, to infer these predicates. This process works by first instrumenting the program to monitor invariants over observed program values, before executing the program and observing which predicates hold. Sets of observed predicates are collected for executions of each of the tests cases within the test suite. Building on work by Liblit et al. [2005] on the use of invariants for statistical fault localisation, the following statistics are calculated for each observed predicate P:

- *Failure*(P): probability the program will fail, given P.
- *Context*(P): probability the program will fail, given the line on which P is tested is reached.
- *Increase*(P): the amount by which P being true increases the probability of failure.

The resulting statistics describe a set of observations of program behaviour that are associated with failure (e.g., the branch condition at line 5 only evaluates to 0 when the program fails). Using these statistics, the following two predicate sets are determined:

- *Increase Set:* all P such that *Increase*$(P) > 0$.
- *Context Set:* all P such that *Context*$(P) > 0$.

The following *universal sets* of predicates are also constructed:

- all P that hold for every execution,

- all P that fail to hold for every execution,
- all P that hold only for the positive tests,
- all P that fail to hold only for the negative tests.

Together, these six sets represent a baseline B, characterising the behaviour of the buggy program. To compare the behaviour of a candidate patch i to the original program, its six invariant sets V_i are computed and compared against B. This comparison is performed by computing a number of statistics:

- The number of predicates that once predicted failure (i.e., the set of predicates that hold only for the negative tests) that no longer hold on failing runs in variant i.
- The number of predicates that did not hold on failing runs for the original program that hold on failing runs for variant i.
- The cardinality of set differences, for each of the six predicate sets.
- The weighted sum of the *Context* scores of predicates in the context set. (This weighting is learnt using linear regression, discussed next.)
- The weighted sum of the *Increase* scores of predicates in the context set

These statistics are combined with the outcomes of the test suite for the candidate patch to form a characteristic vector of scalar features f_i. To transform these vectors into scalar fitness values, linear regression is used to learn a suitable global intercept c_0, and a linear weighting of features, given in Equation 6.1.

$$\text{predicate\_fitness}(i) = c_0 + \sum_j c_j f_{i,j} \tag{6.1}$$

As its training set, the learner requires a large number of evaluated patches for the bug under repair—in the evaluation performed in the [Fast et al., 2010], 1772 such patches were generated—together with their test suite outcomes, and the state of their invariants. Assuming that at least one solution to the bug is found amongst these patches, D_{opt}, the shortest edit distance from a given patch to a known repair, is calculated for each entry in the training set. Linear regression is then used to calculate a global intercept and a vector of feature weights that approximates the edit distance from a given edit to a potential solution.

Although the idea of using learned invariants as part of an alternative, more amenable fitness function sounds promising, the practical utility of the fitness function proposed by Fast et al. [2010] is unclear. The technique requires that a highly expensive process of learning be carried out before the search process begins; for larger programs, such as PHP and Python, gathering the necessary number of patches would likely take longer than 12 hours. More importantly, however, this approach *requires*

that at least one repair to the bug is known before beginning the search. This requirement raises the question: "If a repair to the bug is already known, then why are we searching?"

To evaluate the quality of the fitness function, the authors measure its *fitness–distance* correlation [Jones and Forrest, 1995] for a single bug scenario, taken from a precursor to the GenProg TSE 2012 dataset. The fitness–distance metric approximates the difficulty of a search problem as the correlation between (scalar) fitness values and the edit distance to the nearest (known) solution. For a single bug, the authors found a moderately strong correlation between the predicate-based fitness function and D_{opt}. In contrast, GenProg's original fitness function demonstrated little to no correlation. We do not believe that these results should come as a surprise, nor do they demonstrate the superiority of the predicated-based fitness function; the results are a reflection of the fact that D_{opt} was used to train the fitness function.

Test Case Sampling

In addition to conducting an initial study into the feasibility of using learned program invariants as part of an alternative fitness function, Fast et al. [2010] investigated the use of test suite sampling to reduce the cost of fitness evaluations. Instead of running all tests within the suite for each fitness evaluation, the authors observed the performance of the search—measured by wall-clock time taken to find a repair—when a random sample of tests is used to calculate fitness.[2] Fast et al. [2010] found that the use of sampling, combined with a safe-impact analysis responsible for determining whether the outcomes of a particular test would be affected by a given patch, resulted in an 81% speed-up. Notably, the benchmarks used to conduct this study are amongst those that we found to have weak oracles (e.g., only checking the exit status of the program; see Section 3.2.1). As such, it is unclear whether test case sampling would produce the same gains when used with stronger test suites. The authors found little difference in performance between the use of random selection and Walcott et al. [2006]'s time-aware test case prioritisation technique when selecting the tests within the sample.

Crossover

Oliveira et al. [2016] argue that GenProg's implicit conflation of operator type, fault location, and fix statement into a single discrete unit within the genome results in a landscape that is more difficult to traverse. With the goal of allowing more granular building blocks to be identified and exploited, the authors propose explicitly separating these aspects into their own sub-spaces. To this end, six new crossover operators are proposed, all of which act upon an intermediate representation that

[2] No mention of the size of this sample is made in the paper. Both subsequent experiments [Le Goues et al., 2012a] and the default settings within GenProg use a 10% sample.

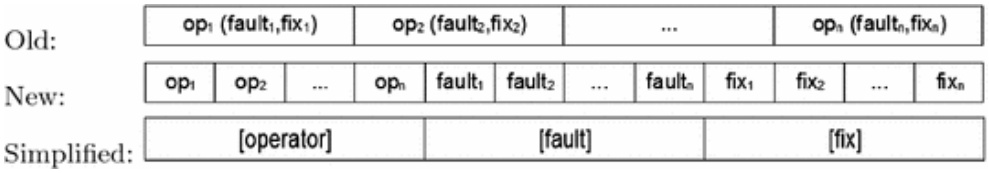

Figure 6.1: An illustration of the implicit search space defined by GenProg's representation and a more granular alternative proposed by Oliveira et al. [2016]. Whereas the type of operation, location, and donor statement used by an edit are all considered to be part of a discrete, evolvable unit within GenProg's representation, Oliveira et al. [2016]'s intermediate representation allows each of these attributes to be treated separately by a set of purpose-built crossover operators.

explicitly separates these sub-spaces, illustrated in Figure 6.1. These six crossover operators are composed from three base operators, with and without a "auxiliary memorisation component":

- *One-Point Crossover on a Single Subspace* (OP1Space) applies variable-length one-point crossover to a single subspace, selected at random, leaving the rest of the intermediate representations of the parents unchanged.
- *Uniform Single Subspace* (Unif1Space) selects a subspace at random and swaps entries between components according to a randomly-generated mask.
- *One-Point Across All Subspaces* (OPAllS) performs a variable-length crossover across the entire intermediate representations of both parents, allowing all subspaces to be mixed simultaneously.

To allow this intermediate representation to be used in tandem with the original `Patch` representation, individuals are subject to an *encoding* and subsequent *decoding* phase. During the encoding phase, the edits within each patch are exploded into each of the three subspaces of the intermediate representation. After crossover is applied across the three subspaces, each intermediate representation is decoded to form a valid patch. As a consequence of allowing individual subspaces to be blindly modified without consideration of the well-formedness of the patch, invalid edits may be introduced. The role of decoding, therefore, is to identify and remove these invalid edits.

Optionally, a memorisation component may be employed to avoid removing invalid edits produced during crossover, thus limiting the destruction of search information. This component caches the details of each edit encountered over the course of the search, allowing recovery queries to be made to find valid edits at a particular location and/or with a given donor statement. When enabled, memorisation will attempt to use part of the information provided by the invalid edit to find an edit that shares some of its details within the cache.

To assess the effectiveness of these operators, the authors compared the resulting performance of each operator against the performance achieved through the

use of GenProg's original one-point crossover operator. Across 43 bug scenarios, taken from small programs in the IntroClass and GenProg TSE 2012 benchmarks, Unif1Space without memorisation yielded a 34% improvement in the success rate of the search. Accompanying this improvement in the success rate, however, was a significant degradation of efficiency, as measured by the average number of test suite evaluations (15.59 vs. 29.32).

Mutational Robustness

Schulte et al. [2013] investigated the widely-held view that the majority of software is highly fragile and that small changes are likely to lead to substantial and undesirable changes in behaviour. The authors found that over 30% of (single-edit) modifications—generated using GenProg's mutation operators—resulted in no change to the outcomes of the test suite. The results were found to hold across 22 programs at both the source code and assembly instruction level, as well as across multiple programming languages, including C, C++, Haskell, and OCaml.

Based on their findings, the authors proposed the use of neutral variants as a means of proactively generating software diversity, allowing techniques such as N-version programming [Avizienis, 1985] to be used to supply an alternative version of a function when it ceases to work under some conditions. To demonstrate the practical utility of such an ability, the authors seeded five latent faults into the original versions of 11 different programs, and evaluated whether a fix to the fault could be found from amongst a sample of 5,000 of its single-edit neutral variants. With respect to an extended test suite, containing tests that were unused in the generation of neutral variants, plausible repairs for 12 out of 55 faults were found.

6.2. Theoretical Analysis

In this section, we identify and qualitatively discuss a number of phenomena exhibited by GenProg's search algorithm, which we believe to be detrimental to its repair process. As part of this discussion, we speculate on both the immediate and wider ramifications of these phenomena to the efficiency and success of the search. Following this discussion, we investigate the role of fitness information within the search. The discussions in this section are supported by a number of graphs, generated over 20 repeats of a sub-set of the GenProg TSE 2012 dataset, with improvements to its weak output checking.

Bloat

The first, and perhaps, most apparent indicator of unintended behaviour in GenProg is the continual growth of its patches over time—a widespread phenomenon within

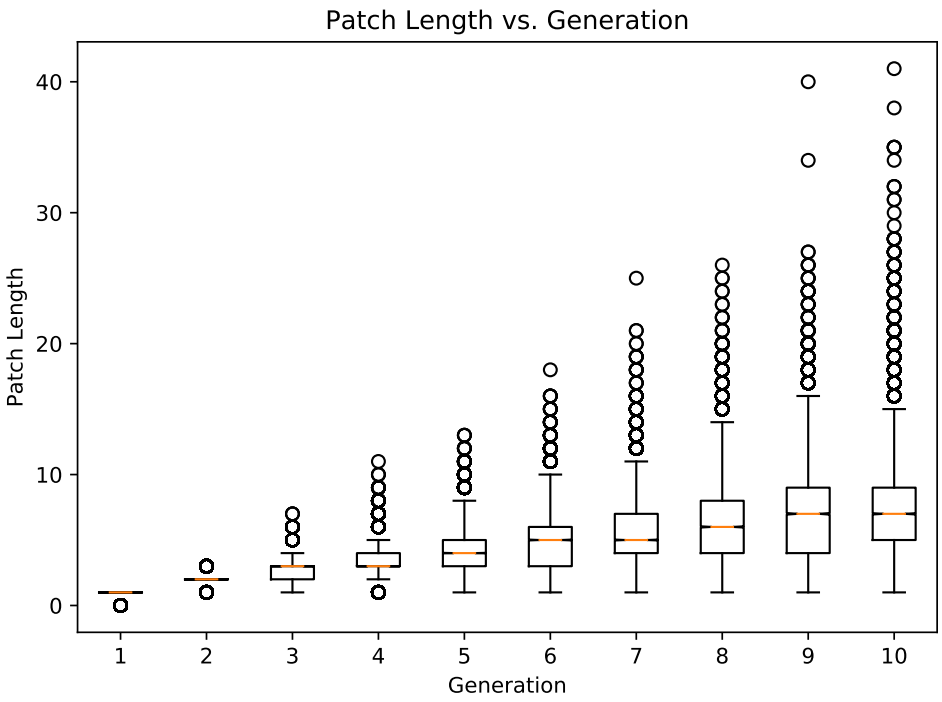

Figure 6.2: Patches tend to become longer over time.

the field of genetic programming, known as *bloat* [Langdon and Poli, 1998]. As with other applications of genetic programming, we observe a steady and sustained increase the size of individuals with each generation, measured by their number of edit operations, Figure 6.2. More specifically, we observed a monotonic increase in the total number of edit operations within the population with each generation, Figure 6.3.[3]

Upon closer inspection of GenProg's search algorithm, this phenomena is not particularly surprising, since the mutation operator appends a newly-sampled edit to the end of each child with a probability equal to the mutation rate; by default, the mutation rate is set to 1.0, causing all children to grow by a single edit.

In practice, GenProg partially overcomes this problem through the use of delta-debugging as a minimisation technique. For more details, see Section 4.1.3. However, this process takes the form of a post-processing step, occurring only once an acceptable solution has been found by the search. In most cases, this minimisation stage reduces the size of the patch to a single edit. For the rest of the search process, however, GenProg takes no active steps to control or reduce bloating within the population.

[3]Note, that this phenomena persists regardless of whether crossover is enabled. In fact, this phenomena is further exacerbated when crossover is disabled, since all individuals within the population are guaranteed to be the same size and to be one edit longer than their predecessor.

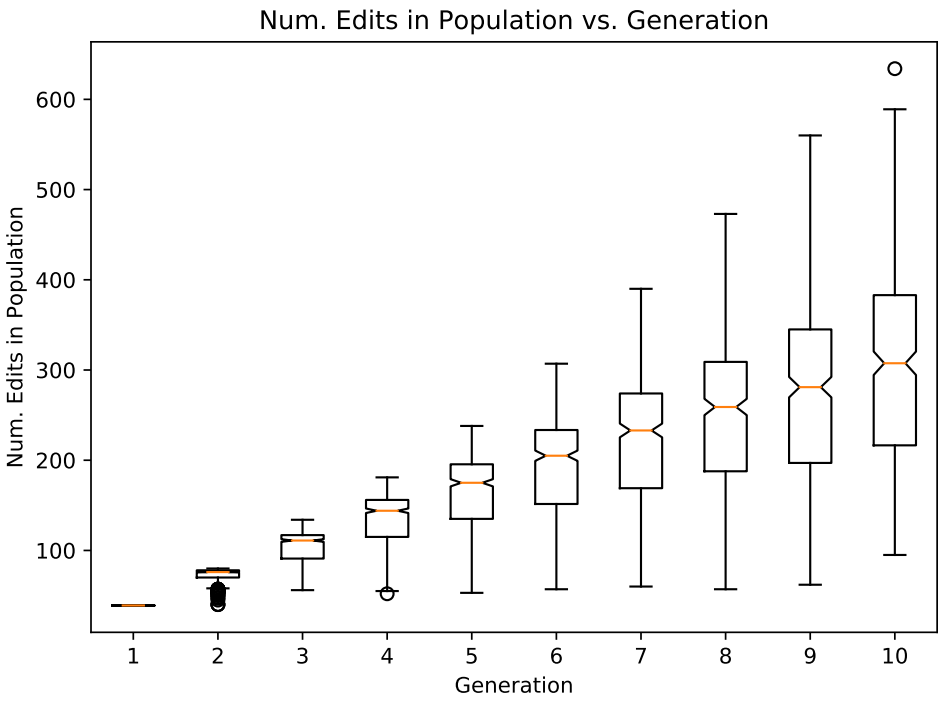

Figure 6.3: Across all problems, we observe a monotonic increase in the total number of edit operations contained within the population despite the ability of crossover to generate smaller individuals.

One may argue that the phenomenon of bloating is irrelevant to the ability of the search to generate and discover correct solutions, and that the search remains unhindered in its presence. Based on observation and analysis of GenProg's search algorithm, we believe this is not the case, and that bloat is detrimental to the search in multiple ways, for the following reasons:

1. Although the edits within the patch may have no effect on the outcomes of the positive test cases, this does not mean that the patched program retains its original or intended semantics. When combined with a weak testing suite, this phenomenon allows destructive changes to be introduced into the population without being detected, and subsequently retained and propagated (since all edits are conserved).

   Over time, as the search finds and exploits more of the weaknesses within the testing suite, the individuals within the population inevitably end up accruing more of these silently destructive edits, and thus, end up further away from the semantics of the original program. This behaviour places the search at odds with its implicit assumption, that the semantics of the repaired program are a short distance away from the faulty program, which allows automated program repair to be a tractable problem.

2. Even in the few cases where the testing suite is perfect at identifying destructive mutants, the search ends up performing redundant work by evaluating edits that appear to be neutral. In these cases, the search is unable to use the results within the cache and so, it must evaluate the mutant as if it were any other. As a result, we believe the search to be spending most of its limited resources in evaluating larger, semantically-equivalent patches.

3. Finally, a human study by Fry et al. [2012] demonstrates that the likelihood of a patch being accepted by a human-developer sharply diminishes as the size of the patch increases and that such increases in patch length are also associated with higher maintenance costs.

Accumulation of Neutral Edits

As bloat causes candidate solutions to move further away from the original program syntactically, the introduction of new edits also drives the program further away from its original and intended semantics as a result. Moreover, this drift is likely to be felt most in the bug-affected regions of the program, covered by few, if any, positive test cases, where almost all changes have no effect on the outcomes of the test suite. In such areas, the search will reduce to a random walk, and as a result of bloat, will continue to accumulate random edits within this region. Consequently, the majority of the search is spent evaluating either bloated but semantically-equivalent candidate patches, or patches that involve so many changes that they are likely to be far away from the intended semantics of the program.

In the absence of effective bloat controls, an implicit selective advantage is created towards larger, more bloated, neutral variants, whose neutral edits are more likely to survive a crossover event. Consequently, these neutral edits will tend to be selected together with other such genes, to the effect that the genome as a while becomes selected for bloat. Whilst the minimal tournament sizes ($k = 2$) used by default in GenProg's tournament selection operator should weaken this pressure towards bloat, it does so at the cost of allowing overtly destructive edits to enter the population.

In summary, without effective bloat controls, patches will tend to accumulate edits that are unhelpful within the context of repair, in the best case. In the worst case, patches will accumulate detrimental edits that silently destroy functionality within the program and prevent solutions from being found.

Operator Biases

In addition to the problems produced by bloat, we believe that the way in which GenProg allows multiple edits to be performed at a single statement within the program leads to subtle biases that ultimately harm its ability to find repairs. Since

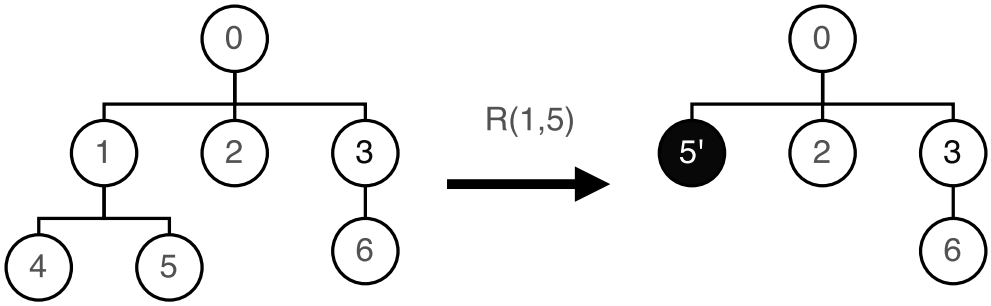

Figure 6.4: An example of the *destructive edit bias*. In this example, a single-edit patch is applied to the original AST, given on the left. This patch replaces the node at location 1 with the node at location 5. When the child tree is subsequently mutated, any changes to locations 1, 4 or 5 will have no effect. Note, the donor node, coloured black, may not be the subject of a future mutation operation. Thus, the effects of this replacement operation are permanent on all of its descendants (except in cases where crossover moves this operation to another child).

the search permits multiple edits to a given statement, a somewhat arbitrary decision must be made as to how these edits should be interpreted when applying the patch.

GenProg deals with this dilemma in perhaps the most intuitive way—applying each edit within the patch in the order in which it appears. However, since the mutation operator may only introduce new edits at the end of the patch, the set of programs that may be reached from a given point becomes constrained by the existing contents of that patch.

The most destructive bias introduced by this behaviour is observed when a deletion or replacement operation is applied to a given statement. In this case, the statement is no longer addressable by future edits, causing the modification to become frozen. As a result, the search is allowed to accumulate non-coding edits at the affected site, since all edits after the `Delete` or `Replace` are ignored. An example of this behaviour is illustrated in Figure 6.4. Since all edits are conserved during crossover and mutation, and given the relatively low selective pressure, this can also result in functionality loss, particularly within regions lacking positive test case coverage. This functionality loss may only be recovered through an increasingly-unlikely crossover event.

Another less destructive, but nonetheless hindering, consequence of this behaviour stems from the treatment of append operations. In the event that a donor statement X is appended before a selected statement S, then a successive append of statement Y at S will result in the statement sequence $S; Y; X$ at the original location of S. Whilst the semantics of this operation seem acceptable, since edits can only be introduced at the end of the patch, the set of reachable programs becomes increasingly constrained with each append. In the case of the example above, if X and Y are appended in the wrong order, or an incorrect statement is appended instead, that

mistake becomes frozen, save for a unlikely re-arrangement of the genome through a series of unlikely crossover events. An illustration of this bias is given in Figure 6.5.

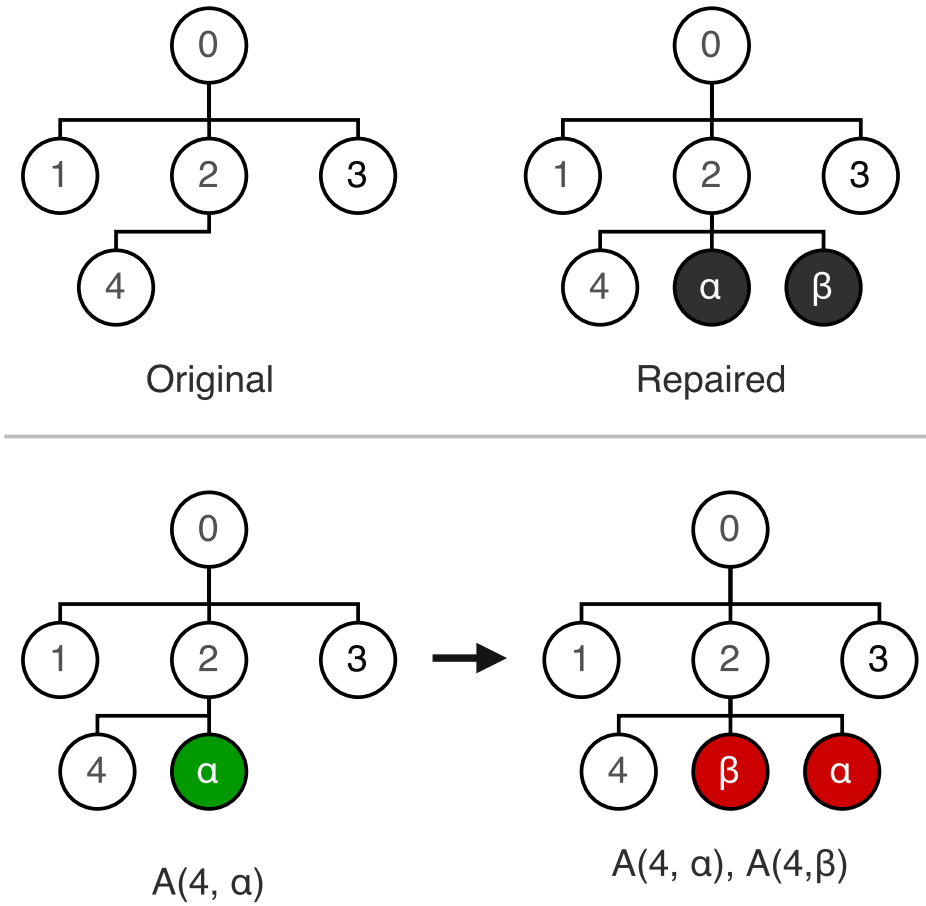

Figure 6.5: An example of the *append bias*. The AST in the top left shows the state of original, faulty program. The AST in the top right shows the repaired version of that AST, containing two missing statements α and β. On the bottom half of the diagram, we show a repair scenario in which this bias is encountered. In the first generation, α is appended after the statement at location 4, matching its intended position in the repaired program. In the following generation, β is appended, yielding the incorrect sequence $\beta;\alpha$. From the first mistake in the order of append operations, the search is unable to move backwards to find a correct order; incorrect edits will continue to be accumulate at statement 4.

In summary, by allowing multiple edits at a single site, a number of subtle biases are introduced into the search, limiting its ability to reach solutions as generations pass.

Limited, Short-Term Memory

As a consequence of the relatively small size of its population, the ability of the search to implicitly retain useful information about the problem and the possible nature of a solution is fundamentally and severely restricted. It is therefore the responsibility of the selection operator to manage the storage of this information, and to hold onto information useful for solving the problem whilst discarding misleading or irrelevant information.

For most optimisation problems where population-based algorithms are used, the role of selection is to identify and exploit high-quality individuals, based on the assumption that such individuals tend to be co-located with other such high quality solutions. In these cases, selection serves primarily as a positive feedback mechanism aimed at promoting higher-quality solutions.

In the case of automated repair, however, we are purely interested in finding a *correct* solution, and from the perspective of the end-user, there is little to no concept of a partial solution. Here, fitness purely serves as a means of more efficiently generating potential solutions, rather than assessing the quality of a solution. To help the search craft potential solutions that are more likely to be correct, we believe that fitness combines elements of both positive and negative feedback:

- **Positive Feedback:** in theory, fitness helps the search to identify and exploit partial solutions, identified by their passing of a sub-set of the negative test cases. In practice, when adequate test suites are used, patches that pass any of the negative test cases are very rarely encountered.
- **Negative Feedback:** whilst positive feedback remains largely unused for the majority of the search, the primary feedback mechanism comes from the avoidance of destructive patches, which cause previously passing test cases to fail.

Despite the fact that the search largely makes use of negative feedback in selecting parents, the population-based nature of the algorithm is almost entirely oriented towards positive feedback, which is rarely used. Populations serve relatively well as retainers of positive information (i.e., potential partial solutions) since they are very likely to persist from one generation to the next, and thus be exploited to find promising solutions. However, populations also make for a woefully-ineffective means of storing and exploiting useful information for problems that are largely driven by negative feedback mechanisms. The reason for this ineffectiveness is that negative feedback is only used to influence what is not contained in the population. Given the vast size of the search space, the very small size of the population (40 individuals, by default), and the substantial cost of evaluating candidate solutions, all of

this negative feedback information—indicating that certain edits are destructive—is lost between generations. One approach to tackling this problem would be to avoid solutions that are known to be destructive, in a similar manner to Tabu search [Glover, 1986]. For reasons given in Section 6.4, we decided not to introduce such functionality into GenProg.

The consequences of this lack of memory are most obvious during mutation, whereupon previously encountered destructive edits are given equal consideration when appending a new edit to the patch as those that have yet to be explored. As a result, the algorithm reintroduces known (but unrecorded) destructive edits into the population, causing potentially useful information about partial solutions to be corrupted and discarded. Even in cases where unexplored edits are added to the patch during mutation, if those edits are destructive, which approximately two thirds of edits are [Schulte et al., 2013], then information will also be corrupted.

In summary, given the nature of the automated program repair search landscape and its heavy emphasis on the avoidance of destructive patches—rather than the pursuit of promising patches—using a population to record problem information appears to be misguided.

Summary

In combination, we believe that these phenomena result in a number of unintended effects, all of which combine to significantly harm the effectiveness of the search algorithm in numerous ways:

- **Inability to reach certain repairs:** as a consequence of operator biases, and further compounded by the problem of bloat, it becomes increasingly difficult for the search to reach certain repairs with each passing generation, since the presence of certain edits within a patch preclude the existence of others. Futhermore, mutations at areas within of the program that lack positive test case coverage, which also happen to be the most likely to be selected, may silently destroy functionality, preventing a repair from being found.
- **Sensitivity to initial conditions:** as a side effect of the search's inability to reach certain repairs over time, the success of the search becomes highly dependent on the outcome of a series of chance events in the early generations of the search. If destructive and restrictive edits successfully invade the population, the search is unlikely to be able to recover.
- **Inefficient usage of test case evaluations:** the inability of the search to explicitly remember and avoid destructive edits, combined with the highly limited memory capacity of its population, leads to corruption of partial solutions and discarding of useful negative-space information.
- **Majority of resources are spent on unlikely repairs:** due to bloat, the search spends the majority of its time evaluating long and highly unlikely

patches, rather than focusing its efforts on patches that are closer to the original program.

- **Poor reliability:** as a result of the above symptoms of these phenomena, we expect to see a decreasing (marginal) probability of success as the search progresses, leading to lower levels of reliability.

Ultimately, these symptoms stem from certain design decisions and trade-offs within the design of GenProg's algorithm, including its ability to perform multiple edits at a given site, and its treatment of the problem as an optimisation problem, rather than a decision problem.

6.3. Empirical Study

Having highlighted a number of potential issues and inefficiencies associated with GenProg's search algorithm—an algorithm similar in all important aspects to most other evolutionary APR techniques—here, we conduct an empirical study to explore whether these claims hold in practice. In doing so, we seek to produce a better understanding of the workings (and shortcomings) of the evolutionary repair process.

To investigate the effects, if any, of bloat and operator biases on the performance of the search, we answer following questions:

- **RQ4A:** Does bloat hinder the ability of the search to efficiently find repairs?
- **RQ4B:** Does removing the ability to perform multiple edits at a given site improve the efficiency and reliability of the search?

After exploring these effects, we turn our attention to the role of fitness information within the search for multiple-edit repairs. To conduct this investigation, we deconstruct the notion of fitness in a systematic, stepwise manner by answering each of the research questions below:

- **RQ5A:** Does selecting individuals to serve as parents at random, rather than in accordance to their fitness, have a measurable impact on the performance of the search?
- **RQ5B:** Does restricting parent selection to non-destructive individuals (i.e., those which pass all of the positive tests), and choosing from amongst them at random affect the performance of the search?
- **RQ5C:** Does selecting non-destructive individuals as parents on the basis of the number of negative tests that they pass—instead of choosing between them at random—affect the performance of the search?

Methodology

To answer each of these research questions, we extended GenProg, and produced a suitable configuration for each question. To reduce the cost of answering these questions, we used asynchronous patch evaluation, and weak equivalency checking, described in Section 4.2. Using these configurations, we collected performance data for each research question by a number of repeat repair trials over a set of 21 bug scenarios.

To balance the accuracy of the results against the staggering time and computational resources required to conduct extensive program repair trials, we performed 10 repeat trials for each bug-configuration; in total, we conducted 1260 individual trials.

For each research question, we measure the performance of the associated configuration against GenProg's standard configuration, in terms of the metrics described below:

- We measure the *effectiveness* of the configuration by the fraction of bug scenarios for which it was able to find a repair.
- For bug-configurations where a repair was found, we measure the *reliability* as the fraction of runs for that bug-configuration that yielded a repair.
- Additionally, we measure a modified version of the previously used NCP metric, a measure of the average number of patch evaluations required to find a repair. The first of these metrics, NCP_U, measures the average number of *unique* candidate patch evaluations. By measuring unique evaluations, rather than all evaluations, we gain a more representative measure of performance for configurations that are more likely to revisit previously evaluated patches. The second, ER, measures the efficiency of the search by the number of unique patch evaluations per run (including runs for which a repair was not found). This measure gives a more reliable estimate of the efficiency of a configuration over multiple runs.

Except where explicitly stated, we used identical configurations and termination criteria for each experiment. An outline of this configuration is provided in Table 6.1. The settings within this configuration are based on the default settings for GenProg, which have been previously used to conduct research [Le Goues et al., 2012a,c]. For the purposes of this study, we chose to enforce an explicit limit on the number of unique candidate patch evaluations—rather than restrict the number of generations, as is *de facto* practice [Le Goues et al., 2012a,b,c; Weimer et al., 2009],—in order to more fairly represent the performance of algorithms that revisit solutions.

As with all of the experiments conducted within this thesis, a complete replication package is provided using RepairBox. Details on replicating the results of this study, along with the other studies within the thesis, can be found in Appendix A. By using RepairBox, we are able to provide a transparent and controlled execution environment without compromising performance.

Population Size	40
Fault Localisation	GenProg (10/1)
Selection	Tournament ($k = 2$)
Crossover Rate	100%
Mutation Rate	100%
Termination Criteria	
Num. Unique Candidates	400

Table 6.1: The baseline parameters of the algorithm used within each run.

CPU:	Intel Xeon E5-2673 v3 (4 cores)
OS:	Ubuntu Server 16.04 LTS (64-bit)
Kernel:	4.4.0-77-generic
Docker:	1.12.6, build 78d1802
RAM:	8 GB
Storage:	64 GB, SSD
Cost:	$0.249/hr

Table 6.2: Specifications of the Microsoft Azure F4 compute instances used to collect the data for this study.

To run these experiments, we used a large number of parallel F4 instances cloud computing instances on the Microsoft Azure cloud computing platform. Specifications for these instances are given in Table 6.2.

Benchmarks

In selecting a suitable set of bug scenarios for this study, we primarily considered three factors. The first is that performing 1260 repeat repair trials over the bug scenarios should be neither prohibitively time-consuming, nor exorbitantly expensive. The second is that the test suite should meet the minimal set of requirements laid out in Chapter 3; bug scenarios that fail to thoroughly check the outputs and resulting state of the program should not be considered. From earlier experiments, we found that including such benchmarks—as have been used to conduct previous studies on the efficiency of GenProg [Le Goues et al., 2012a,c; Oliveira et al., 2016; Weimer et al., 2009]—can produce misleading results. In such cases, variants of GenProg which push the search towards regions of the space likely to contain plausible, but incorrect, solutions are assumed to be better performing variants. When one controls for test suite quality, one finds that these variants lead to a lower performance. The third and final factor is that the bug scenarios should include, but not necessarily be restricted to, bugs whose patches require multiple edits. Without the inclusion of such bugs, one cannot assess the effectiveness of the search technique when it comes to multi-edit bugs—bugs which have been largely been beyond the reach of

search-based repair.

With these considerations in mind, we first assessed a number of publicly available datasets of real-world bugs: ManyBugs [Le Goues et al., 2015], Codeflaws [Tan et al., 2017], and the GenProg TSE 2012 [Le Goues et al., 2012b] benchmarks. We found that the bug scenarios within the ManyBugs dataset exhibited excessively-long compilation and test suite evaluation times, and that the majority of test suites failed to meet our minimum quality requirements (many exclusively consider the exit status of the program). We encountered similar concerns of quality in the GenProg TSE 2012 benchmarks. On inspection of the subjects within the Codeflaws dataset, we determined that GenProg would be highly unlikely to find repairs for any of its bugs, owing to the small size of the programs (mostly less than 20 lines of code).

Given the extensive difficulties involved in using naturally-occurring bugs, we turned our attention to synthetic bugs. Since the research questions asked in this study are purely concerned with the process of searching for patches, rather than the ability to solve real bugs, we believe this decision to be sound. We first explored using a sub-set of the C bug scenarios within the Software Infrastructure Repository, but found that their vast search space reduced the probability of repair to such a point that any significant difference in performance between search techniques would be impossible to gauge. Although the programs used by these bug scenarios are smaller than those used in previous APR research, it is important to note that in these cases, the search was restricted to a single file within the project—usually containing a few hundred lines of code (and not the millions of lines of code used by the entire project). In contrast, the programs within the SIR are collapsed into a single file containing thousands of lines of code.

Algorithm 5: Test Suite Reduction

```
ReduceTestSuite(tests, lines, line_cov, test_cov) begin
    selection ← ∅ ;
    line_cov ← {(l, 0) | l ∈ lines} ;
    for i in 1..k do
        for l_1 in lines do
            candidates ← CoveringTests(l_1) ;
            poor_cov ← {t | t ∈ tests ∧ line_cov[t] < i} ;
            if line_cov[l_1] ≥ i or candidates = ∅ then
                continue
            end
            best_candidates ← ∅ ;
            best_score ← 0 ;
            for t in candidates do
                score ← #{l | l_2 ∈ poor_cov ∧ Covers(t, l_2) } ;
                if score > max_score then
                    best_score ← score ;
                    best_candidates ← {t} ;
                else if score = max_score then
                    best_candidates ← best_candidates ∪ {t} ;
                end
            end
            t ← RandomChoice(best_candidates) ;
            UpdateCoverage(line_cov, test_cov[t]) ;
            selection ← selection ∪ {t} ;
            tests ← tests \ {t} ;
        end
    end
    return selection
end
```

Ultimately, we found that when combined with Pythia (see Section 3.2) to produce a high-quality oracle, the programs within the Siemens benchmarks provide all of the desired characteristics. Although each program is only several-hundred lines of code in length, the corresponding size of the search space matches those used in previous studies [Le Goues et al., 2012a, 2015; Long and Rinard, 2015; Mechtaev et al., 2016]. The test suites used by these programs, containing thousands of tests, are unlike those used in previous work, however. To reduce the burden involved in evaluating these test suites, and to produce a test suite closer to those encountered in real-world programs, we devised and used a simple test suite reduction heuristic. This algorithm, Algorithm 5, uses a fast, stochastic greedy search to ensure that, where possible, each line within the program is covered by at least $k = 5$ tests. Using this algorithm, we were able to reduce the size of the test suites by roughly a factor of 100. This substantially diminished level of coverage appeared to have

Program	# LOC	# Stmts	# Tests	# Tests′
TCAS	135	136	1608	20
TOTINFO	230	271	1052	25
REPLACE	494	372	5542	42
PRINTTOKENS	342	292	4073	14
PRINTTOKENS2	388	340	4115	18
SCHEDULE	249	194	2650	15
SCHEDULE2	273	217	2710	19

Table 6.3: A table of the subject programs used to conduct this study. **# LOC** describes the number of source code lines in the original program, as measured by CLOC. **# Stmts** specifies the number of statements within the GENPROG's pre-processed AST representation of the program. **# Tests** gives the size of the original test suite for the program.

no discernible effect on the quality of patches produced during the study—a phenomenon we intend to investigate more closely in future work. Details of the seven programs used as subjects for this study, together with the sizes of their original and reduced test suites, and their number of lines of code, are given in Table 6.3.

Upon inspection of original bug scenarios within the Siemens subject programs, we found that GENPROG was unable to fix most of them—owing to the granularity of its repair model—and that most fixes required only a single edit. To gain an understanding of the ability of GENPROG's algorithm to compose multiple-edit patches, and to ensure our understanding was based on bugs that are solvable within GENPROG's search space, we manually seeded bugs into these programs. For each of the seven programs, we seeded a one-line bug, a two-line bug, and a three-line bug, giving a total of 21 bug scenarios, balanced across the different bug sizes.

To ensure that the seeded bugs could be solved, we ensured that a fix could be generated from the source code in the faulty file. For example, to seed a bug scenario that required the insertion of a missing statement, we deleted a statement in the program, and ensured that a redundant copy of that statement could be found elsewhere (and that the redundant copy was covered by the test suite). That is, for each seeded bug scenario, we devised a repair within GenProg's search space, and then applied the inverse of the fix to the program. We also ensured that all mutations to the program resulted in the failure of at least one test within the reduced test suite.

Instructions on finding and replicating these bug scenarios are given in Appendix A.

Results

A summary of the results across all of the studied configurations is aggregated into the following tables: Table 6.4 describes which bugs were successfully repaired at least once for each configuration. We find that the baseline configuration is unable to repair any bugs that contain more than a single fault. Across all configurations, only one bug containing two faults was repaired, and no bugs containing three faults were repaired. Table 6.5 shows the observed reliability of each configuration. Table 6.6 gives the median number of unique candidate patches required to find a repair for each bug-configuration. Table 6.7 specifies the cost of finding a repair, measured by the number of unique candidate patch evaluations across all runs, including those which failed to yield a patch, divided by the number of successful runs.

For the remainder of this section, we use these results to provide answers to each of our research questions.

Bug Scenario	Baseline	Bloat Control	Restricted	Shadow	Neutral	Neutral Negative
REPLACE-ONE	✓	✓	✓	✓	✓	✓
REPLACE-TWO						
REPLACE-THREE						
PRINTTOKENS-ONE	✓	✓	✓	✓	✓	✓
PRINTTOKENS-TWO						
PRINTTOKENS-THREE						
PRINTTOKENS2-ONE					✓	✓
PRINTTOKENS2-TWO						
PRINTTOKENS2-THREE						
SCHEDULE-ONE	✓	✓	✓	✓	✓	✓
SCHEDULE-TWO						
SCHEDULE-THREE						
SCHEDULE2-ONE	✓	✓	✓	✓	✓	✓
SCHEDULE2-TWO						
SCHEDULE2-THREE						
TOTINFO-ONE		✓			✓	✓
TOTINFO-TWO						
TOTINFO-THREE						
TCAS-ONE	✓	✓	✓	✓	✓	✓
TCAS-TWO				✓		
TCAS-THREE						
Successful	5	6	5	6	7	7

Table 6.4: A summary of the bugs for which a repair was found at least once over the ten runs, for each configuration. ✓indicates that at least one patch was found for the given bug-configuration.

Bug Scenario	Baseline	Bloat Control	Restricted	Shadow	Neutral	Neutral Negative
REPLACE-ONE	30%	30%	20%	20%	100%	100%
REPLACE-TWO	—	—	—	—	—	—
REPLACE-THREE	—	—	—	—	—	—
PRINTTOKENS-ONE	60%	60%	70%	60%	70%	80%
PRINTTOKENS-TWO	—	—	—	—	—	—
PRINTTOKENS-THREE	—	—	—	—	—	—
PRINTTOKENS2-ONE	—	—	—	—	10%	10%
PRINTTOKENS2-TWO	—	—	—	—	—	—
PRINTTOKENS2-THREE	—	—	—	—	—	—
SCHEDULE-ONE	20%	20%	20%	20%	20%	20%
SCHEDULE-TWO	—	—	—	—	—	—
SCHEDULE-THREE	—	—	—	—	—	—
SCHEDULE2-ONE	30%	20%	30%	50%	30%	30%
SCHEDULE2-TWO	—	—	—	—	—	—
SCHEDULE2-THREE	—	—	—	—	—	—
TOTINFO-ONE	—	10%	—	—	10%	10%
TOTINFO-TWO	—	—	—	—	—	—
TOTINFO-THREE	—	—	—	—	—	—
TCAS-ONE	100%	100%	90%	100%	100%	90%
TCAS-TWO	—	—	—	10%	—	—
TCAS-THREE	—	—	—	—	—	—

Table 6.5: A comparison of the reliability of different configurations for each bug scenario. The presence of an "—" symbol indicates that no patches were found for that given bug-configuration.

Bug Scenario	Baseline	Bloat Control	Restricted	Shadow	Neutral	Neutral Negative
REPLACE-ONE	166.0	85.0	63.5	93.5	144.0	81.5
REPLACE-TWO	—	—	—	—	—	—
REPLACE-THREE	—	—	—	—	—	—
PRINTTOKENS-ONE	13.5	13.5	14.0	13.5	14.0	37.0
PRINTTOKENS-TWO	—	—	—	—	—	—
PRINTTOKENS-THREE	—	—	—	—	—	—
PRINTTOKENS2-ONE	—	—	—	—	109.0	296.0
PRINTTOKENS2-TWO	—	—	—	—	—	—
PRINTTOKENS2-THREE	—	—	—	—	—	—
SCHEDULE-ONE	31.0	31.0	31.0	31.0	31.0	31.0
SCHEDULE-TWO	—	—	—	—	—	—
SCHEDULE-THREE	—	—	—	—	—	—
SCHEDULE2-ONE	39.0	34.5	39.0	60.0	39.0	39.0
SCHEDULE2-TWO	—	—	—	—	—	—
SCHEDULE2-THREE	—	—	—	—	—	—
TOTINFO-ONE	—	325.0	—	—	198.0	183.0
TOTINFO-TWO	—	—	—	—	—	—
TOTINFO-THREE	—	—	—	—	—	—
TCAS-ONE	26.5	26.5	20.0	26.5	26.5	20.0
TCAS-TWO	—	—	—	83.0	—	—
TCAS-THREE	—	—	—	—	—	—

Table 6.6: A summary of the median number of unique candidate patch evaluations required to find a repair, for each bug-scenario. The presence of an "—" symbol indicates that no patches were found for that bug-configuration.

Bug Scenario	Standard	Bloat Control	Restricted	Shadow	Neutral	Neutral Negative
REPLACE-ONE	1087.33	1089.00	1663.50	1693.50	167.40	115.00
REPLACE-TWO	—	—	—	—	—	—
REPLACE-THREE	—	—	—	—	—	—
PRINTTOKENS-ONE	331.50	336.33	277.86	310.67	264.00	173.13
PRINTTOKENS-TWO	—	—	—	—	—	—
PRINTTOKENS-THREE	—	—	—	—	—	—
PRINTTOKENS2-ONE	—	—	—	—	3709.00	3896.00
PRINTTOKENS2-TWO	—	—	—	—	—	—
PRINTTOKENS2-THREE	—	—	—	—	—	—
SCHEDULE-ONE	1631.00	1631.00	1631.00	1631.00	1631.00	1631.00
SCHEDULE-TWO	—	—	—	—	—	—
SCHEDULE-THREE	—	—	—	—	—	—
SCHEDULE2-ONE	972.67	1634.50	972.67	477.80	976.67	972.67
SCHEDULE2-TWO	—	—	—	—	—	—
SCHEDULE2-THREE	—	—	—	—	—	—
TOTINFO-ONE	—	3925.00	—	—	3798.00	3783.00
TOTINFO-TWO	—	—	—	—	—	—
TOTINFO-THREE	—	—	—	—	—	—
TCAS-ONE	58.00	40.70	61.33	36.40	45.30	35.33
TCAS-TWO	—	—	—	3683.00	—	—
TCAS-THREE	—	—	—	—	—	—

Table 6.7: An overview of the cost of finding a patch for each bug-configuration, measured by the total number of unique candidate patch evaluations, across all runs, divided by the number of runs that were successful. Bug-configurations with an "—" symbol indicate that no patches were found during any of the runs.

RQ4A: Does bloat hinder the ability of the search to efficiently find repairs?

We first introduce a variant of the search algorithm, fitted with active bloat controls. One way of doing this is to break the conservation of edits within the mutation operator by allowing edits to be removed, through the introduction of an UNDO operator. To add such an operator, however, we would need to arbitrarily determine a suitable probability of its application, which opens the door to overfitting the particular benchmarks being studied. Since the majority of problems that GENPROG has been applied to can be solved within a single edit, one could engineer an UNDO probability such that all patches within the population remain at this particular size.

A simpler method to achieve bloat control, less prone to overfitting, is to replace the selection operator with a *double-tournament* selection operator, based on the non-parametric parsimony pressure techniques introduced by Luke and Panait [2002]. Rather than drawing k individuals at random from the population and choosing the best amongst them to act as a parent for the next generation—as in standard tournament selection—individuals are also compared on fitness, through a series of "qualifier" tournaments.

- k "qualifier" tournaments are carried out, in order to select the k individuals that will compete in the final "fitness" tournament.
- Each "qualifier" tournament selects two individuals at random from the population, and chooses the smallest as the winner, with probability $S_p/2$, where S_p is a real number between 1.0 and 2.0. Setting S_p to 2.0 ensures the smallest individual is always picked, whilst 1.0 chooses between the individuals at random.
- The "fitness" tournament selects the individual with the highest fitness as the winner, and uses random selection as a tie-breaker.

After introducing double-tournament selection, we compared the performance metrics of the resulting algorithm against the baseline. We found that double-tournament selection was able to fix an additional bug scenario compared to the baseline (5 vs. 6), as shown in Table 6.4. Furthermore, all bugs that are repaired by the baseline are also repaired when double-tournament selection is used; double-tournament selection dominates the baseline, in terms of effectiveness.

We observe little to no difference in reliability between double-tournament selection and the baseline, Table 6.5. For scenarios where reliability can be compared, we see a minimal 10 percentage point difference.

In terms of efficiency, as measured by NCP_U, double-tournament selection dominates the baseline. Double-tournament selection exhibits a lower NCP_U for two of the five comparable bugs, and an identical NCP_U for the remaining three. When measured by expense, the results are less clear. Double-tournament selection is marginally worse in two cases (1087.33 vs. 1089.00, and 331.50 vs. 336.33; Table

0	1	2	...	N - 1
R 12	NULL	NULL	...	D

Figure 6.6: Our restricted representation implicitly encodes individuals as a fixed length list of optional edits P. Each entry of the list, P_i, describes the edit, if any, that should be applied to the statement with number i.

6.7), substantially worse in one (927.67 vs. 1634.50), identical in one, and improved in one (58.00 vs. 40.70).

Taken together, these results demonstrate little to no improvement in performance, suggesting that traditional bloat control measures may be ineffective within GenProg's search space. This may be down to the need for more careful, but expensive, parameter tuning. Alternatively, it may be that bloat controls further hinder the memory capabilities of the population. As the population contains fewer edits, it also retains less information about the solution space. In summary, although the search appears to be most effective during its earliest stages, the use of traditional bloat control measures appear to be unsuccessful in increasing reliability or efficiency.

RQ4B: Does removing the ability to perform multiple edits at a given site improve the efficiency and reliability of the search?

To determine the effects of removing the ability to perform multiple edits at a single location, we needed to introduce a slightly-modified representation, and an associated set of operators, all ensuring that no more than a single edit could be made at a given site. Instead of encoding the patch as an arbitrary-length sequence of edit operations, we effectively represent the patch as a fixed length list of edits P, as illustrated in Figure 6.6. Each entry of the list, P_i, describes the edit e_i that should be applied at particular location i. The absence of an edit at a particular site is denoted by a `NULL` edit. In practice, to avoid high memory costs, this representation is realised by placing certain constraints on variable-length edit sequences.

To accommodate these changes, we replace the standard mutation operator used by GenProg with a uniform mutation operator, which replaces any existing edit at a given statement with a newly sampled one. Similarly, we replace GenProg's standard, variable-length one-point crossover with a symmetrical uniform-crossover operator, outlined in Figure 6.7. Each crossover event takes a pair of parent patches as input (x, y), and generates a pair of children (a, b), by iterating through each modifiable statement s, and assigning the edit at s from either parent to x, and the other to y, by selecting at random.

Compared against the baseline effectiveness, we find that the *restricted* configuration solved the same set of bugs as the baseline. We observe little to no difference in reliability when the restricted configuration is used: for two out of five bugs, reliability is the same; for one out of five, the restricted configuration is minimally

A	B
R 12	NULL
NULL	A 13
NULL	NULL
NULL	NULL
....	...
D	NULL

C	D
R 12	NULL
A 13	NULL
NULL	NULL
NULL	NULL
...	...
NULL	D

Figure 6.7: Uniform crossover takes two parents, A and B, and generates a pair of symmetrical children, C and D.

better; for the remaining two bugs, the baseline configuration is minimally better. In terms of efficiency, when measured by NCP_U, we observe an identical efficiency for two bugs, an improvement for two bugs (166.0 vs. 63.5, and 20.0 vs. 26.5; Table 6.6), and a slight reduction for one bug (13.5 vs. 14.0). When efficiency is measured using the expense metric, we see a worse performance for two bugs (1087.33 vs. 1663.50, and 58.00 vs. 61.33; Table 6.7), an identical performance for two bugs, and an improvement for one bug (331.50 vs. 277.86).

In summary, whilst restricting the search to only a single edit at each site fails to produce any sizeable improvements in performance, it also appears to have little negative effect. This small change substantially reduces the size of the search landscape, and in practice, prevents few bugs from being patched. This observation can be better exploited by search techniques which are not population based.

RQ5A: Does selecting individuals to serve as parents at random, rather than in accordance to their fitness, have a measurable impact on the performance of the search?

Building on previous studies showing that random search outperform genetic algorithms, here we determine whether this result was due to any exploitable information provided by fitness values, or if it was largely due to differing search spaces—in the previous study of random search [Qi et al., 2014], RSRepair only considered single edit patches—and the ability to terminate on the first instance of failure.

To perform this experiment, we simply ran an otherwise identical version of GenProg's search algorithm, except with its parent selection replaced by random selection.

From the results in Table 6.4, we see that random search dominates the baseline in terms of effectiveness; random search fixes all the bugs fixed by the baseline, plus tcas-two. Furthermore, of all the configurations we investigated, random search

was the only technique that found a solution to a multi-edit bug. It may be the case that the bug in question, TCAS-TWO, exhibits a particularly deceptive search space, ill-suited to optimisation techniques.

In terms of reliability, we observe little difference between random search and the baseline: random search was minimally better for one bug, minimally worse for two, and identical for the remaining two bugs on which it can be compared.

When comparing the efficiency of random search and the baseline, we see mixed results. When measured by NCP_U, random search is better in one case (166.0 vs. 93.5; Table 6.6), worse in one (39.0 vs. 60.0), and identical for the rest. In terms of the expense metric, the baseline is outperformed in three cases (331.50 vs. 310.67, 972.67 vs. 477.80, and 58.00 vs. 36.40; Table 6.7), better in one case (1087.33 vs. 1693.50), and identical in one case.

In summary, these results suggest that GENPROG's search algorithm is no more effective than selecting patches at random. Overall, random search appears to exhibit a stronger performance than the baseline.

RQ5B: Does restricting parent selection to non-destructive individuals (i.e., those which pass all of the positive tests), and choosing from amongst them at random affect the performance of the search?

Having shown that fitness information either has no effect on the search, or that it actively misleads it, we explore two variants of GENPROG's fitness function, designed to more aggressively prune and exploit the search space. In this experiment, we examine whether the algorithm would be better served by strictly selecting neutral edits. Or, put another way, we ask whether tolerating destructive patches (i.e., patches which cause passing tests to fail) is beneficial to the search. To do so, we introduce a variant of GENPROG wherein the default parent selection method is replaced by a pair of successive selection operators: firstly, the population is reduced to those candidates which pass all of the positive test cases; from this restricted pool, n individuals are sampled at random, with replacement.

When compared to the baseline, we find that this *neutral* configuration finds patches for a greater number of bugs (5 vs. 7; Table 6.4). Specifically, the neutral configuration successfully finds patches for all of the single-edit bugs, which includes the set of all bugs repaired by the baseline. In all cases, the reliability of the neutral configuration is higher (30% vs. 100%, and 60% vs. 70%; 6.5) than or equal to that of the baseline configuration. In terms of NCP_U, we observe little difference in efficiency, but a more pronounced improvement is observed when efficiency is measured by expense: for the five bugs on which efficiency can be compared, the neutral configuration is better for three (1087.33 vs. 167.40, 331.50 vs. 264.00, 972.67 vs. 477.80; Table 6.7), and identical for the remaining two.

The substantial performance improvements stemming from restricting selection to only those solutions which do not fail any positive test cases suggests that detailed

positive test case information is not particularly useful to the search. This finding, that the search should only pursue non-lethal changes, offers room for significant optimisation: one can use test case prioritisation to order the positive test cases, thus minimising the number of test case evaluations (i.e., regret) necessary to identify a *lethal* edit.

RQ5C: Does selecting non-destructive individuals as parents on the basis of the number of negative tests that they pass—instead of choosing between them at random—affect the performance of the search?

Building on our findings from the previous experiment, showing that the search is more performant when restricted to selecting from amongst patches which do not fail any of the positive tests, we explore whether negative test case outcomes contribute to the success of the search. To determine the contribution of negative test outcomes, if any, we modified the configuration introduced in the previous experiment to perform a tournament selection between non-destructive patches, based on the number of passing negative tests.

In terms of effectiveness, we find no difference between the *neutral negative* and neutral configurations: each technique solves the same set of bugs. Similarly, we find little difference between the reliability of the two configurations: in one case, the neutral negative configuration is minimally better; in another case, the neutral configuration is superior, and for the remaining cases, both configurations are identical. We observe differences in the efficiency of the two configurations, as measured by the expense metric. For five out of seven bugs, the neutral negative configuration is superior, in one case it is worse, and for the remaining case, it is identical.

In summary, these results suggest that knowledge of negative test outcomes is useful and that it may be used to enhance the efficiency of the search. When the search is used with a higher resource limit, beyond the default limit used by GenProg, this increased efficiency may also translate into a higher effectiveness.

6.4. Greedy Algorithm

As we demonstrated in our empirical study, the performance characteristics of the search can be improved through certain modifications to its various parameters and components. However, such parameter tweaks are a cheap way of addressing the deficiencies of GenProg's genetic algorithm—they ignore the deeper cause of those deficiencies. Improving the genetic algorithm along one dimension often degrades it along another. For instance, we can bolster the memory of the search by increasing the size of the population, but doing so is likely to reduce the rate of convergence. Similarly, we can control bloat by introducing controls such as double-tournament selection, but doing so also causes information about the problem, implicitly encoded in the population, to be lost.

Crucially, our study shows that, for the most part, the search performs better when it only considers whether a candidate solution passes or fails any of its positive test cases, and how many of its negative test cases it passes. Restricting the search to patches which do not degrade existing functionality—as measured by the positive test cases—allows it to operate with greater effectiveness and efficiency. We also found that selecting from amongst potential parents on the basis of the number of negative test cases that they pass can lead to further gains in effectiveness and efficiency.

Motivated by these findings, here we propose the use of a form of greedy algorithm to conduct search-based program repair. Since we found little to no degradation to performance by restricting the search to considering only a single edit at each location, our algorithm exploits this observation to reduce the size of its search space.[4] Our technique allows the space to be explored more efficiently, with the bulk of its efforts spent looking at the smallest possible patches—in which a solution is more likely to be encountered—rather than evaluating increasingly large patches over time. In doing so, we find ourselves closer to the driving motivation behind search-based program repair: patches are only a short distance away from the buggy program. At the same time, to allow the technique to scale to multiple-edit patches and to speed up convergence, our algorithm exploits promising combinations of edits as soon as they are encountered, as explained next.

To achieve these behaviours, our algorithm divides into search space into two parts: an *exploration space* and an *exploitation space*. The former is composed, initially, of the set of all possible single-edit modifications to indicted statements within the program. As the search progresses, edits are sampled without replacement from this space, in an effort to discover partial solutions. Over time, this space continues to shrink, monotonically, until no edits remain. The exploitation space contains the set of all partial solutions discovered over the course of the search. When a partial repair is discovered, it is added to this space. The algorithm then determines whether any other partial repairs may be combined with this newly added partial, and if so, pushes their concatenations to the exploitation stack. Whilst this stack is non-empty, the algorithm evaluates each of its combinations of partial solutions. If the combination passes all of the negative test cases passed by each of its parts, the combination is added to the exploitation space. If not, this is treated as a sign that at least one of the two partial solutions involved in the combination is incorrect, and so their combination is discarded. This process continues until either an acceptable solution is found, a resource limit is reached, or both the exploration space and exploitation stack are empty.

[4] GenProg, and to some degree, MintHint, are the only two repair techniques to allow multiple edits to a single program location. From observation of GenProg's patches, we find no cases where this ability is used in an acceptable, minimised repair.

Algorithm 6: Greedy Search Program Repair

```
explore ← GenerateExplorationSpace() ;
exploit ← ∅ ;
combinations ← [] ;
while resources not exhausted do
    /* perform exploitation when possible                          */
    if combinations ≠ [] then
        xy ← Pop(combinations) ;
        x, y ← combo ;
        /* xy should pass all tests passed by x and y              */
        Evaluate(xy, T_N) ;
        foreach t : PassedTests(x) ∪ PassedTests(y) do
            if not PassesTest(xy, t) then
                break
            end
        end
        /* xy should pass all positive tests                       */
        if not PassesAll(xy, T_P) then
            break
        else if PassesAll(xy, T_N) then
            return xy
        else
            ReportPartialSolution(xy, exploit, combinations) ;
        end
    /* check if exploration space is empty                         */
    else if explore = ∅ then
        return ⊥
    /* explore single-edit space                                   */
    else
        cand = Sample(explore) ;
        if PassesAny(cand, T_N) and PassesAll(cand, T_P) then
            if PassesAll(cand, T_N) then
                return cand
            end
            ReportPartialSolution(cand, exploit, combinations)
        end
    end
end
```

Pseudocode for our greedy algorithm is given in Algorithm 6. As an initialisation step, the GenerateExplorationSpace populates a memory- and look-up-efficient data structure with the set of all possible single edit patches to the program, at locations implicated by the fault localisation. The main loop of the algorithm continues to iterate until a solution is found, a resource limit is hit, or the search space has been exhausted.

At each iteration, the algorithm checks to see whether there is a promising combination of edits waiting to be evaluated. If so, the algorithm evaluates the combination on the (covering) negative tests. The combination xy is saved to the exploit space, using ReportPartialSolution!, provided it passes all of the tests that are passed by each of its parts, x and y. For example, if x passes $\{N_1\}$, and y passes $\{N_2\}$, xy should pass $\{N_1, N_2\}$, as well as all of the positive tests. As an optimisation, the combination is only evaluated against the positive tests if it satisfies this invariant over the negative test cases. A further optimisation, left for future work, would be to evaluate the union of the negative tests passed by x and y, using prioritisation and terminating on the first instance of failure.

In cases where the *combinations* stack is empty, the algorithm proceeds to sample an edit from its *explore* space via the Sample! function. This process of sampling first selects a subject statement from the program, according to its fault localisation. For the sake of allowing a fair comparison to GenProg's existing search algorithm, we used GenProg's default 10:1 fault localisation. Once a statement has been selected, the algorithm proceeds to choose the type of edit at random. In accordance with GenProg's default mutation operator probabilities, we give each operator an equal probability of selection. Finally, the algorithm samples an edit of the chosen type at the selected location at random without replacement (i.e., the edit is removed from the exploration space). Upon sampling an edit, the algorithm updates the state of its *explore* space, removing the chosen statement from consideration by the fault localisation if no edits remain at that statement.

Once an edit has been sampled, the algorithm evaluates that edit on all of its covering negative tests, in parallel. If the edit fails all of the negative tests, it is discarded. If, and only if, the edit passes at least one negative test, is it evaluated against the covering positive tests. To increase the efficiency of the search, the set of covering positive tests are ordered by their observed likelihood of failure, prior to evaluation; this minimises the expected number of test evaluations required to reject an edit. Once sorted, tests are then evaluated in parallel. If at any point a positive test is failed, the search ceases to evaluate the rest of the covering positive tests and discards that edit. In the event that the edit passes all of its covering positive tests and at least one of its covering negative tests, that edit is recorded as a partial solution by ReportPartialSolution.

When a partial solution is discovered by the search, ReportPartialSolution traverses its *exploit* space to find promising combinations of edits that include the edits within the reported partial solution. Combinations, each formed of two partial solutions within the *exploit* space which satisfy certain criteria, are pushed to the

Algorithm 7: Greedy Search, Partial Solution Reporting

```
ReportPartialSolution(x, exploit, combinations) begin
    stmts_x ← CoveredStmts(x) ;
    passes_x ← PassedTests(x) ;
    /* Look for promising combinations including this patch      */
    foreach y : exploit do
        /* Find any overlap in edit statements                   */
        stmts_y ← CoveredStmts(y) ;
        stmts_xy ← stmts_x ∩ stmts_y ;
        /* Look for mutually exclusively test passes             */
        passes_y ← PassedTests(y) ;
        passes_Δ ← (passes_x \ passes_y ≠ ∅) ∧ (passes_y \ passes_x ≠ ∅)
        /* Determine the length of the resulting patch           */
        length ← Length(x) + Length(y) ;
        if (stmts_xy = ∅) ∧ passes_Δ ∧ (length ≤ length_max) then
            xy ← Combine(x, y) ;
            Push(combinations, xy) ;
        end
    end
    Add(exploit, partial) ;
end
```

combinations stack: This behaviour causes the algorithm to pause its exploration of the single-edit space and to immediately exploit the combination. Pseudocode for the ReportPartialSolution procedure can be found in Algorithm 7. A partial solution x can be combined with partial solution y to form xy, provided that the following criteria are satisfied:

1. The set of statements modified by x and y do not overlap.
2. Let T_x and T_y be the set of tests passed by x and y, respectively. Ensure that the following holds:

$$(T_x \setminus T_y \neq \emptyset) \ \wedge \ (T_y \setminus T_x \neq \emptyset)$$

   In other words, x must pass at least one test that is not passed[5] by y, and y must pass at least one test that is not passed by x.
3. The length of the resulting combination xy does not exceed the maximum allowed patch length. For this study, we set this limit to three edits.

[5]Note, this includes tests that are not covered by a given patch, as well as those that fail.

After finding satisfactory combinations for the reported partial solution, ReportPartialSolution records the solution to the *exploit* space.

Evaluation

To determine whether our proposed algorithm outperforms that used by GenProg, we evaluated each against the bug scenarios used in our empirical study.

From observation, we deemed that the artificial candidate evaluation limits imposed by GenProg were far too low for use in multiple-edit scenarios—the probability of encountering, let alone combining, all of the partial fixes within this time frame is highly unlikely. Thus, to account for the low probability of encountering the components of a partial repair, and to gain a better understanding of the real performance of each configuration, we evaluated each using a time limit, rather than a limit on the number of unique candidate evaluations. These time limits were set according to the number of seeded faults in the bug scenario, as outlined below.

- *One Fault:* 30 minutes.
- *Two Fault:* 90 minutes.
- *Three Fault:* 240 minutes.

Given a limited budget and the increased demands of evaluating patches for such lengths of time, we performed five repeat runs for each bug-algorithm. To conduct this evaluation, we once again made use of the cloud compute nodes, described in Table 6.2, for a total of over 300 hours of compute time.

From the results of this evaluation, we compared the effectiveness and reliability of each algorithm using the same metrics used in the empirical study. To compare efficiency, we measured the cost of finding an acceptable patch as the total time spent searching, across all runs, divided by the number of runs wherein a repair was found. A comparison of effectiveness and the bugs fixed by each algorithm is given in Table 6.8. A summary of the reliability for each technique is provided by Table 6.9. Details of the efficiency of each technique are listed in Table 6.10.

Our results show that the greedy algorithm successfully fixes more bugs than the baseline (14 vs. 8; Table 6.8). The genetic algorithm is able to find a solution to replace-three, whereas the greedy algorithm fails. On closer inspection of the seeded fault, we see that the canonical patch requires exploration of the neutral edit space; two of the edits within this patch do not pass any of the negative test cases, and thus do not appear as partial solutions to the greedy search.

Comparing the reliability of the two techniques, shown in Table 6.9, we find that the greedy algorithm performs at the same level or greater than the genetic algorithm for 20 of the 21 bugs. For 10 of the 14 bugs solved by the greedy algorithm, it achieves a reliability of 100%. By comparison, the genetic algorithm only achieves 100% reliability for one bug, tcas-one.

Bug Scenario	Genetic	Greedy
REPLACE-ONE	✓	✓
REPLACE-TWO		✓
REPLACE-THREE	✓	
PRINTTOKENS-ONE	✓	✓
PRINTTOKENS-TWO		
PRINTTOKENS-THREE		
PRINTTOKENS2-ONE		✓
PRINTTOKENS2-TWO		
PRINTTOKENS2-THREE	✓	✓
SCHEDULE-ONE		✓
SCHEDULE-TWO		✓
SCHEDULE-THREE		
SCHEDULE2-ONE	✓	✓
SCHEDULE2-TWO		✓
SCHEDULE2-THREE	✓	✓
TOTINFO-ONE		✓
TOTINFO-TWO		✓
TOTINFO-THREE		
TCAS-ONE	✓	✓
TCAS-TWO		
TCAS-THREE	✓	✓
Successful:	8	14

Table 6.8: A comparison of the bugs patched by each technique.

In the seven cases where the efficiency of the two search algorithms can be compared, the greedy algorithm is between a factor of 1.4 and 57 faster than the genetic algorithm. For TCAS-THREE, the greedy algorithm is roughly a factor of 2.7 slower.

On closer inspection of the results of the greedy algorithm for TCAS-THREE, we observe an abnormally large number of partial solutions, numbering close to 1000. Given the opportunistic nature of the greedy algorithm, much of its time is spent attempting to combine these partial solutions, rather than exploring the single-edit search space more broadly.

In summary, our greedy algorithm outperforms the genetic algorithm used by GENPROG in terms of effectiveness, reliability and efficiency. Moreover, our approach is simpler and does not require (expensive) parameter tuning.

Bug Scenario	Genetic	Greedy
REPLACE-ONE	80%	80%
REPLACE-TWO	—	100%
REPLACE-THREE	20%	—
PRINTTOKENS-ONE	80%	100%
PRINTTOKENS-TWO	—	—
PRINTTOKENS-THREE	—	—
PRINTTOKENS2-ONE	—	40%
PRINTTOKENS2-TWO	—	—
PRINTTOKENS2-THREE	40%	100%
SCHEDULE-ONE	—	100%
SCHEDULE-TWO	—	100%
SCHEDULE-THREE	—	—
SCHEDULE2-ONE	40%	100%
SCHEDULE2-TWO	—	80.00%
SCHEDULE2-THREE	60%	100%
TOTINFO-ONE	—	100%
TOTINFO-TWO	—	100%
TOTINFO-THREE	—	—
TCAS-ONE	100%	100%
TCAS-TWO	—	—
TCAS-THREE	40%	20%

Table 6.9: A comparison of the reliability achieved by the genetic and greedy search algorithms, measured by the fraction of runs wherein an acceptable repair was found. An "—" is used to denote bug-configurations where no repair was found across any of the runs.

6.5. Future Work

Although our proposed greedy search algorithm outperforms the baseline algorithm used by GenProg, there are a number of modifications that we plan to make to improve it further still.

A number of existing, complementary techniques, described below, could be applied to the algorithm with relative ease to increase its efficiency and reliability. For the purpose of this study, we omitted these improvements in order to focus our attention on the relative performance of our greedy algorithm over GenProg's baseline genetic algorithm.

- A more effective fault localisation approach could be used, allowing the exploration space to be pruned and focused.
- A probabilistic model could be used to assign different probabilities to indi-

Bug Scenario	Genetic	Greedy	Reduction
REPLACE-ONE	712.14	528.20	1.35
REPLACE-TWO	—	438.33	—
REPLACE-THREE	57,875.16	—	—
PRINTTOKENS-ONE	495.01	8.69	56.95
PRINTTOKENS-TWO	—	—	—
PRINTTOKENS-THREE	—	—	—
PRINTTOKENS2-ONE	—	3,919.53	—
PRINTTOKENS2-TWO	—	—	—
PRINTTOKENS2-THREE	21,700.96	974.10	22.28
SCHEDULE-ONE	—	886.24	—
SCHEDULE-TWO	—	2,399.63	—
SCHEDULE-THREE	—	—	—
SCHEDULE2-ONE	2,959.97	328.98	9.00
SCHEDULE2-TWO	—	4,637.24	—
SCHEDULE2-THREE	9,697.29	2,117.91	4.58
TOTINFO-ONE	—	167.02	—
TOTINFO-TWO	—	773.71	—
TOTINFO-THREE	—	—	—
TCAS-ONE	23.01	7.37	3.12
TCAS-TWO	—	—	—
TCAS-THREE	21,651.88	59,104.75	0.37

Table 6.10: A comparison of efficiency between the genetic search and greedy search algorithms, measured by the total wall-clock time across all runs, in seconds, divided by the number of successful runs. **Reduction** describes the reduction factor achieved by the greedy algorithm, compared to the genetic algorithm.

vidual edits within the repair space, in a similar fashion to PROPHET [Long and Rinard, 2016]. Edits that are deemed more likely to occur could be given priority over less likely edits. One could also use the model to create a deterministic ordering of exploration space, as a pre-processing step. By doing so, the algorithm would become deterministic, allowing it to be evaluated without the need for repeated runs.

- Anti-patterns, proposed by Tan et al. [2016], could be incorporated into the exploration space, allowing edits associated with low-quality patches to be pruned.

Further efficiency and reliability improvements may be achieved through modifications to the algorithm itself:

- Instead of combining and exploiting complementary partial solutions at the earliest moment, the algorithm could defer this process until a combination, whose constituent edits pass all of the negative tests, amongst them. A for-

mal expression of this criterion is given in Equation 6.2, where C denotes a potential combination of edits $e \in C$, and $\text{Passes}(e, t)$ is used to assert that the singleton patch for e passes a given test t.

$$\forall t \in T_N : \exists e \in C : \text{Passes}(e, t) \tag{6.2}$$

This behaviour would dramatically improve the efficiency of the search in cases where large numbers of partial but misleading solutions are reported, such as the TCAS bugs.

- In the current version of the algorithm, patches are first evaluated against the negative test suite to check that they pass at least one negative, or all of the negatives passed by their constituent parts, before being subjected to the positive tests. To reduce the number of test evaluations, let us assume that number of patches which pass at least one negative test case is low, and that patches which pass negative tests are *unlikely* to fail their covering positive tests. We can exploit these assumptions by deferring evaluation of the positive test cases until a patch is found which passes all of the negative tests. In practice, the positive tests vastly outnumber the negative tests, and so the gains from such an optimisation could be significant.

  For the majority of programs that we have encountered, synthetic and organic, these assumptions are true; patches which pass any of the negative tests are relatively rare. In a small number of cases, however, such patches are frequently encountered. To handle this, a smart repair technique could measure the observed frequency of such patches against an expected frequency, and if the expected frequency is exceeded, a different set of optimisations could be used instead.

To allow the algorithm to solve a greater number of bugs—albeit it at some cost to efficiency—the algorithm could be tuned to exploit neutral edits, in addition to partial solutions. One way of exploiting neutral edits could be used with the optimisations described above. After exhausting the exploration space and finding an acceptable solution, a *neutral* space could be created. The set of neutral edits could then be found and saved to this space, using test prioritisation to reduce the expected cost of doing so. Once the neutral space is created, the algorithm could exhaustively combine pairs of neutral edits, in the search for partial solutions. Partial solutions discovered during this phase would be recorded to the exploitation space, as usual, and combined with complementary partial solutions.

In addition to the above optimisations, we plan to explore whether the inclusion of certain heuristics can be used to prune the search space. Specifically, we are interested in answering the following question: "Given two unique patches, A and B, with known test suite outcomes, does the knowledge of the test suite outcome for their combination AB provide us with (usable) information about the solution?" To answer this question, we intend to explore whether certain test suite outcomes for AB can be used to identify parts of the solution and to prune certain edits.

6.6. Conclusion

In this chapter, we conducted an theoretical and empirical analysis of the search algorithm used by GenProg, one of few repair techniques (theoretically) capable of performing multiple-edit repair. We identified a number of fundamental weaknesses within GenProg's algorithm, stemming from its generational, population-based nature; fixing any one of these issues causes another to be exacerbated. We also explored the role of fitness information within the search and found that the search was more efficient, effective and reliable when selection was restricted to solutions which pass all of their covering positive tests. Modifying this selection between non-destructive patches to favour patches that pass previously failing tests was found to produce further performance gains.

Motivated by these results, we proposed and evaluated an alternative search algorithm, based on a greedy search. Across a benchmark of 21 bug scenarios of varying size, we found that our algorithm beat GenProg's in terms of every performance metric we measured. Although our aggressive restriction of the search space produced a better performance overall, the greedy search was unable to patch one of the scenarios repaired by GenProg. Additionally, the high frequency of negative test passes on another bug scenario caused the efficiency of our algorithm to drop below that of GenProg. We outlined a number of improvements that we intend to make to the algorithm in the future, which may alleviate these issues and increase its performance further.

In summary, through theory and observation, we showed that GenProg's search algorithm has deficiencies. Existing approaches, such as RSRepair, AE and SPR are significantly more efficient due to aggressive optimisations, but all are limited to single-edit patches. We proposed a novel search algorithm for performing multiple-edit repair, which demonstrates promising results. In the future, we plan to optimise this algorithm further and to evaluate it against a dataset of real-world bugs in large-scale programs.

CHAPTER 7

Conclusion

In this section, we present a summary of our findings, in terms of the research questions presented in Chapter 1, and briefly outline directions for future research. Finally, we provide a set of concluding remarks.

7.1. Summary

RQ1: Can the results of candidate patch evaluations, gathered over the course of the search, be used to improve the accuracy of the fault localisation, online?

Inspired by recent results in applying mutation analysis to the problem of fault localisation, in Chapter 4 we explored whether the test outcomes of candidate patches could be used to identify promising fix locations, online. To answer this question, we conducted a 12-hour random walk of 28 different bugs, 15 of which were artificial, and the remaining 12 organic. Our results demonstrate a statistically-significant difference (with a medium-sized effect) in the between the *passing-to-failing* distributions of statements that were changed by the developer, and those that were not.

No significant difference was found between the *failing-to-passing* distributions of human-modified and non-modified statements, suggesting that this *failing-to-passing* information has little to no use in detecting potential fix locations. We proposed and evaluated a number of online fault localisation measures based on our findings, but found that none of them was able to consistently outperform the use of offline localisation; MUSE and Metallaxis were also unable to outperform offline localisation.

There may be a number of reasons for the lack of a significant improvement in the accuracy of the fault localisation:

- Our evaluation naively combined the results of the mutation analysis by simply computing their product. Determining a meaningful way of combining multiple sources of fault localisation appears to be a non-trivial problem. The use of machine learning techniques may be required to discover effective ways of combining information.

- We observe a search landscape in which most repairs either had no impact on the outcomes of the test suite, or they failed all of their passing tests. The lack of diversity in test suite outcomes hinder the effectiveness of the analysis and increase the difficulty of discriminating between fixable and non-fixable program locations. The shape of this landscape may be a result of GenProg's coarsely-grained statement level operators.

  Alternatively, it may be that real-world test suites for C programs tend to exhibit a largely all-or-nothing behaviour. Upon inspection of Python's tests, for instance, we observe that each test case in fact corresponds to a whole suite of atomic tests for one particular module. The results of mutation analysis may be improved through the use of test case atomisation—a technique which cannot be applied to C programs, due to the lack of a ubiquitous testing framework. It may also be that mutation analysis performs better when combined with purpose-built, automatically generated test suites.

- To be effective, mutation analysis may require a large number of mutants per program location. Due to the substantial overheads associated with compilation, we were able to generate relatively few mutants within our 12-hour window. Previous mutation analysis results involve the generation of tens of thousands of mutants—a feat that would require days, if not weeks of compute time to realise in large-scale, real-world programs.

RQ2: Is plastic surgery equally effective for all repair actions?

In Chapter 5, we mined thousands of real-world bug fixes from hundreds of the most popular open-source projects on GitHub. After extracting instances of 23 repair actions within these mined fixes, we determined how many of those instances could be grafted from the file under repair. We found that repair actions at the block-level (e.g., Insert Else-If Branch) were amongst the least likely to be grafted, whereas those operating at finer levels of granularity (e.g., Modify Assignment) exhibited the highest levels of graftability. To improve the efficiency of automated repair, future repair tools may exploit this observation by simply omitting (or de-prioritising) block-level repair actions.

We find that the majority of repairs occur below the statement-level—an encouraging result, given that plastic surgery appears to be most effective at finer levels of granularity. In 58% of cases, the modified version of a statement could be found elsewhere in the file (when abstract code snippets were used). We can use this finding to increase the effective of future repair tools by focusing statement replacement on structurally similar statements.

As part of our analysis, we also found that statement deletion was the second most common repair action, occurring in around a third of bug fixes. This suggests that preventing explicit statement deletion [Long and Rinard, 2015; Qi et al., 2015] may be misguided and likely to reduce the number of bugs that can be repaired.

RQ3: Can the effectiveness of plastic surgery be increased through the use of unlabelled code snippets?

In Chapter 5, we also investigated the effectiveness of plastic surgery when variable labels are stripped from snippets. When plastic surgery is performed with these *abstract code snippets*, we find that graftability is substantially increased. To increase the number of repairable bugs, we strongly advocate the inclusion of abstract donor code snippets into future search-based repair techniques.

In the future, we plan to implement our proposed repair actions and donor code snippets in a new repair tool. Building on previous work by Long and Rinard [2016] and Soto et al. [2016], we could feed the patches mined by BugHunter to a machine learning technique, to produce a model for performing fix localisation (i.e., to assign probabilities to the likelihood of certain bug fixes, based on those observed in historical patches).

RQ4: Do biases within GenProg's operators degrade its performance?

In Chapter 6, we identified a number of potential biases and inefficiencies in GenProg's search algorithm as a part of a larger theoretical analysis. We showed how allowing multiple edits at a single location may limit the search from reaching certain areas of the search space, and how patches tend to grow in size over the course of the search (i.e., bloat occurs).

As part of an empirical study, we examined whether the performance of the search could be improved by fixing these problems:

- To address the problem of bloat, we replaced GenProg's standard selection operator with a double-tournament selection. We found little to no difference in the performance of the search, despite the observation that most patches are found earlier in the search (when patches are smaller). This result may be due to the lack of tuned parameters, or it may be that bloat controls further reduce the memory of the search.
- As a simple fix to the identified operator biases, we assessed the performance of GenProg when its algorithm is limited to a single edit per location. As with the use of bloat controls, we found no discernible improvement in performance.

None of our changes to GenProg in response to these biases had a noticeable effect on its performance. The lack of any response to these changes, be it positive or negative, shows that changes to important components and parameters of GenProg's search algorithm have no effect. This observation, together with our findings regarding GenProg's fitness function, suggests that the search is not operating as intended.

RQ5: Does fitness help to guide GenProg's search algorithm towards solutions?

As part of our analysis of GenProg's search algorithm in Chapter 6, we identified the implicit positive and negative feedback mechanisms provided by fitness information: The test outcomes of previously passing tests serve to prevent the degradation of existing functionality, whereas the outcomes of previously failing tests serve to promote the exploitation of partial solutions.

Through an empirical analysis, we demonstrated that the search performs at least as well as the baseline when selection is restricted to non-destructive patches (i.e., patches that do not fail previously passing test cases). Therefore, gathering information for all of an individual's previously passing tests is redundant. The search can operate more efficiently, as measured by the number of test evaluations, by performing test case prioritisation over the previously passing tests, and terminating on the first instance of failure.

To determine the contribution of negative test outcomes, we introduced a variant of GenProg wherein selection was restricted to non-destructive individuals, but selection between those individuals was performed on the basis of the number of negative test passes. In most cases, we found that this algorithm allowed the search to find a solution in fewer candidate patch evaluations. For scenarios with particularly-weak test suites, where a large number of negative test passes were observed, this algorithm led to a reduced performance.

RQ6: Is there a more effective search algorithm for generating multiple-edit patches than the genetic algorithm used by GenProg?

Motivated by the results of our analysis of GenProg's search algorithm, in Chapter 6 we proposed an alternative search algorithm for multiple-edit repair, based on a greedy algorithm. Our proposed algorithm was shown to beat GenProg's genetic algorithm across all of our performance metrics (efficiency, effectiveness, reliability). These results display promise for the future of multi-edit, search-based program repair. In future work, we plan to explore a number of optimisations to this algorithm, and to assess its effectiveness of a larger set of real-world, multi-line bugs.

7.2. Future Work

In this section, we briefly discuss two promising areas for future research. A more detailed discussion on future research regarding repair models, fault localisation and search algorithms can be found at the end of each of the respective chapters.

- **Platform for Automated Program Repair:** Part of the reason for GenProg's perceived lack of effectiveness may not be that its repair model lacks expres-

siveness, as suggested by others [Long and Rinard, 2015]. Rather, its problems may be the consequence of a number of unintended code transformations performed by CIL—the AST analysis framework upon which GenProg is based—which may make the program unsuitable for repair. Below, we describe two examples of this behaviour:

- if-statements with multiple terms (e.g., `if (x && y && z)`) are exploded into a sequence of side-effect-free, single-term if-statements, shown below.

```
if (x) {
  if (y) {
    if (z) {
      ...
    }
  }
}
```

- Statements with possible side effects are transformed into a sequence of side-effect-free statements, through the introduction of temporary variables. For example, the statement:

```
int z = f(x) + g(z);
```

  is transformed into the following:

```
int t1 = f(x);
int t2 = g(z);
int z = t1 + t2;
```

As a result of these transformations, the program *inflates* in size. In the case of the SIR bug scenarios, we observed single statements that were exploded into more than ten. Since all of these statements belong to the same basic block, there is no potential improvement to the fault localisation (assuming spectrum-based fault localisation is used). This linear increase in the number of statements within the program leads to a quadratic increase in the size of the search space: An average inflation factor of 10 could result in a 100-fold increase in the size of the search space. Accompanying this growth in the search space is an increase in the size of the solution. Where previously a patch could be generated with a single edit (e.g., insertion of the statement from the example above), now a set of statements must be introduced, in an acceptable order.

In addition to increasing the difficulty of the problem along these two dimensions, inflation introduces other unintended issues through its use of temporary variables. Crafting a repair now requires that a particular temporary variable is in scope.

In ongoing work, we are building a new framework for automated program repair of C and C++, on top of Clang—a modern program analysis toolchain, which preserves the original source code of the program. As part of this framework, we plan to integrate a DSL for formally defining repair actions in terms of program transformations. Additionally, to reduce the considerable overheads associated with compilation, we plan to introduce super-mutation, i.e., multiple mutants are compiled once and selected between at run-time.

- **Test Suites for Automated Program Repair:** During our empirical analysis of GenProg's search algorithm, we employed a simple test suite reduction technique to minimise the original test suites for the Siemens benchmarks by two orders of magnitude. Despite the large reduction, from thousands to tens of tests, we observed few cases of overfitting. These anecdotal results would seem to suggest that any improvements to repair quality stemming from the use of larger test suites are quickly diminishing as the size of the test suite increases. In future work, we plan to explore exactly what makes a good test suite for automated program repair. These qualities may involve mutation score, coverage, the novelty of execution path, or data-flow invariants.

  Interestingly, we found that many low-quality repairs associated with GenProg are found in test suites which exhibit a low entropy in across their possible outcomes. Test harnesses that simply check the exit status of a program, such as the PHP scenarios within ManyBugs, exhibit the lowest-possible entropies. From the perspective of the test harness, most test cases appear to have identical outcomes. By using Pythia to automatically generate an oracle that checks the standard output, standard error, and resulting state of the sandbox, the number of identical tends to drop considerably (i.e., this enhanced test harness exhibits a higher degree of entropy across the behaviours of its tests). When this enhanced oracle is used, the number of plausible solutions for the TSE 2012 benchmarks drops from thousands to fewer than ten.

  If we can find a criterion to express the suitability of a test suite for APR, then we could use that criterion as an objective function when performing test-suite reduction. Alternatively, we could use this criterion to guide the automatic generation of a highly-effective, minimal test suite.

- **Fault Localisation:** In Chapter 4, we found that candidate patches generated using GenProg's statement-level operators were ineffective at localising the fault when used to perform mutation-based fault localisation (MBFL). Upon closer investigation, we found that the majority of the candidate patches generated using GenProg's coarsely-grained mutation operators exhibited an "all-or-nothing" behaviour, where the mutation either caused all covering tests to fail, or it had no effect on their outcomes at all. Given the previous successes of using MBFL with traditional mutation testing operators [Moon et al., 2014a; Papadakis and Le Traon, 2015], which typically operate at the level of expressions, rather than statements, it may be that mutation-based fault localisation is only effective when used with mutants generated by finely-grained

operators.

Motivated by the lack of effectiveness of GenProg's statement-level operators, we plan to investigate the effectiveness of a wide range of mutation operators at varying levels of coarseness. We also plan to explore whether the test suite behaviours of particular kinds of mutants can be used to predict the shape of underlying faults. For instance, if mutation of a particular if-condition appears to have no effect on the outcomes of the test suite, it may suggest a faulty if-condition. By probing the shape of the fault, the search space can be vastly reduced to the small sub-set of repairs that fit a believed shape. As well as exploring the behaviour of MBFL using different mutation operators, we intend to assess whether ensemble learning techniques may be used to better aggregate mutant information and offline, spectrum-based fault localisation information.

- **Repair Model:** In Chapter 5, we investigated and demonstrated the effectiveness of plastic surgery in the context of actual repair operators. Since we found that levels of graftability (i.e., the likelihood of discovering a suitable code snippet within the corpus) varied highly between different repair operators, a natural next step would be to explore whether other factors affect graftability. For instance, we may wish to determine whether missing print statements and strings are likely to be discovered within the code. By better understanding the strengths and limitations of plastic surgery, we can design repair tools that are better able to repair the subset of faults that are most amenable to plastic surgery; unlikely repairs could be pruned from the search entirely.

  Using the tool we developed to conduct the analysis in Chapter 5, BugHunter, we could build upon work by Soto et al. [2016] by devising techniques for *graft localisation* (i.e., determining the likelihood that certain snippets form part of the repair). Soto et al. [2016] look at the kind of statement that is most likely to replace a statement of a given kind. Their work could be extended by looking at likely insertions, as well as by examining a richer set of features, beyond just the kind of the statement. A richer set of features, closer to those used in [Long and Rinard, 2016], could look at the variables used by the snippet (and the extent to which they overlap with, or complement, the original statement). Machine learning techniques could be applied to that set of features to learn an effective graft localisation function.

  The results of our analysis could also be bolstered by the development and subsequent use of more sophisticated bug-fix identification procedures. One might also attempt to automatically categorise bugs, and to use that information to determine whether certain kinds of bugs are more amenable to repair, and to improve graft localisation by finding snippets that are most likely to be used to repair faults of that kind.

  For more details on future work related to the work conducted in Chapter 5, see Section 5.7.1.

- **Search:** In Chapter 6, we showed that a form of greedy search outperforms is better suited to automated repair than the genetic algorithm used by GenProg. Although our greedy search algorithm was substantially more efficient than the genetic algorithm approach, its performance could be further improved by a number of small optimisations. Instead of combining and exploiting complementary partial solutions at the earliest possible moment, an optimised version of the algorithm could wait until a combination of partial solutions, whose constituent edits pass of all of the negative tests, is found. By waiting until a (potential) complete solution is found, the search should show a vastly improved performance for deceptive problems, where many partial solutions are incorrectly reported (such as the tcas bugs). Further gains in efficiency could be made by deferring the evaluation of evaluation of the positive tests until a combination of edits that pass all of the negative tests is found.

  Rather than pursuing improvements in efficiency, one may adapt our algorithm to be less greedy to allow a greater number of bugs to be fixed. Instead of solely combining and exploiting partial solutions, the search could also combine neutral edits.

  In addition to improvements to efficiency and effectiveness, we plan to explore whether knowledge of the test suite outcomes for combinations of patches can be used to probabilistically determine whether the individual edits within that patch are parts of the solution or if they are misleading. More specifically, we seek to answer: "Given two unique patches, A and B, with known test suite outcomes, does the knowledge of the test suite outcome for their combination AB provide us with (usable) information about the solution?"

  Technical details on potential optimisations to our search algorithm can be found in Section 6.5.

7.3. Concluding Remarks

Realising the long-held goal of automatically repairing programs poses enormous challenges, far exceeding those that early results would seem to suggest. Despite the daunting vastness of the problem, in less than a decade, researchers have demonstrated techniques capable of fixing a small, but considerable sub-set of single-edit bugs in real-world, large-scale code. Over the next decade, researchers face an even greater set of challenges, as we attempt to scale techniques to a larger number of bugs, spanning multiple lines, all whilst remaining cost efficient.

The sheer magnitude of the challenges ahead may—understandably—cause some to doubt that program repair will ever become an integral part of real-world software development processes. To those that doubt, recall that a decade prior to the introduction of GenProg, the concept of automated program repair would have been

deemed purely science fiction. Since then, the question has shifted from "is automated program repair possible?" to "which bugs can we automatically repair?"

In this thesis, we explored each of the challenges facing search-based program repair over the coming decade, and in the process discovered hints of what a next generation repair technique might look like: We devised a more effective algorithm for multiple-edit repairs, found ways of exploiting plastic surgery to fix a greater number of bugs, and presented a more robust platform for conducting program repair research. Not all of our results were positive: Contrary to our expectations, and findings of previous research, we found that the use of mutation-based fault localisation had no impact on fault localisation accuracy. In such a early field of research, seemingly promising ideas are likely to fail—a small price to pay, for the privilege of pioneering. These results help us to understand where best to focus our attention—or rather, where not to waste it—and raises interesting questions regarding the cases in which mutation analysis *can* be used to improve fault localisation.

An exciting decade of research lies ahead—we'll see you on the other side.

Appendices

APPENDIX A

Reproducibility

A.1. Fault Localisation

Raw data, analysis scripts and a replication package for the results presented in Chapter 4 can be downloaded at:

`https://bitbucket.org/ChrisTimperley/ssbse-2017-data`

Source code for the modified version of GenProg used in these experiments (and those in Chapter 6) can be found at:

`https://bitbucket.org/ChrisTimperley/GP3`

A.2. Repair Model

A replication package for the results in Chapter 5 can be found at:

`https://bitbucket.org/ChrisTimperley/phd-repair-model`

Note, raw results for this study are not available online, due to size limitations (the raw data is several hundred GBs in size).

BugHunter, the tool used to mine bug fixing commits and repair actions from Git repositories can be downloaded at:

`https://github.com/ChrisTimperley/BugHunter`

A.3. Search

Raw data, analysis scripts and a replication package for the results in Chapter 6 can be found at:

`https://bitbucket.org/ChrisTimperley/phd-search`

APPENDIX B

Repair Action Mining

B.1. AST and Edit Script Generation

After identifying the bug fixing commits within a Git repository, BugHunter proceeds to generate the abstract syntax trees for each of the modified source files, in both the faulty and fixed versions of each identified bug fix, using GumTree [Falleri et al., 2014]. Within each generated AST, each node is assigned a unique integer identifier $n \geq 0$, given by its post-order position within the tree. Additionally, for each pair of faulty and fixed versions of a source code file (F, F'), a set of AST node mappings $M(F, F')$ and an AST edit script $\Delta(F, F')$ are generated, using a modified version of GumTree [Falleri et al., 2014].

The mapping $M(F, F')$ is composed of a sequence of ordered pairs, (n_f, n'_f), where n_f specifies the position of a node in F, and n'_f specifies the the location of the matching node in F'. In the case where the node located at n_f in F has no matching node in F', then n'_f is assigned to $\perp$. Similarly, when a node exists in F', but no matching node is found in F, n_f is assigned to $\perp$.

Each edit script $\Delta(F, F')$ describes a sequence of low-level operations that may be applied to the faulty source file F to transform it into its patched form F'. Unfortunately, GumTree provides no formal definition of its edit script language, and so, for the purposes of this analysis, we define its operations as follows:

- Delete(X) removes the node located at position X within P.
- Insert(X', Y', k) describes the insertion of the node located at X' in P' as the k-th child of the node located at Y' in P'. Note, that Y' is not necessarily contained in P.
- Update(X, ℓ') updates the label ℓ of the node located at X in P, with label ℓ'.
- Move(X, X') moves the node located at position X within P to location X' in P'. X' is obtained by finding the corresponding mapping for $(X, X') \in M(F, F')$.

By performing the analysis on the ASTs of each repaired source file, and their accompanying edit scripts, rather than the plain text and `diff` outputs between each version of the file, aesthetic and irrelevant changes to the code, including the modification of comments can safely be ignored. More importantly, this treatment allows each of the described repair actions to be detected using a strict set of formal rules

(based upon $M(F, F')$ and $\Delta(F, F')$), without the need to resort to unsound, ad-hoc approaches based on analysing the `diff`.

In a preliminary version of this analysis, BugHunter's (heuristic) ability to perform pre-processing on arbitrary C projects was used to handle pre-processor macros (e.g., `#ifdef`, `#define`), potentially allowing the analysis to deal with a wider range of bugs. However, pre-processing added a significant overhead to the process of AST generation (to the extent that certain projects, such as the Linux kernel, were no longer feasible to analyse), and made little difference to the detection of repair actions. To reduce the cost of the analysis, and to allow a larger, wider body of programs to be incorporated, this pre-processing stage has been dropped from the final analysis.

Edit Script Post-Processing

To reduce the complexity of the analysis, we perform a number of post-processing steps on the edit scripts and accompanying node mappings produced by GumTree:

- **Node Mapping, M:** using the mapping of node numbers in P to P', M_{num}, we generate a mapping of nodes from P to P' by their pointers. In cases where a node does not exist in either P or P', the unpaired node n is mapped to $\perp$ (i.e., $(n, \perp)$ or $(\perp, n)$).
- **Edit Script Nodes:** each node number in the edit script is replaced by a pointer to its corresponding node.
- **Before Statements, S:** computes the set of statements in P.
- **After Statements, S':** computes the set of statements in P'.

B.2. Detection Rules

In this section, we provide the rules we devised to detect instances of the repair actions listed in 5.4.3. Below, we discuss a number of helper functions that we use to reduce the complexity of these detection rules.

We introduce the function Parent$(node)$ to find the parent of a given node within its AST. If the node has no parent (i.e., it is the root of the AST), $\perp$ is returned.

To simplify finding the nearest ancestors to a given node n that belongs to a certain kind k, we introduce the function NAK:

$$\text{NAK}(n, k) = \begin{cases} \perp & \text{if } n = \perp \\ node & \text{if HasKind}(n, k) \\ \text{NAK}(\text{Parent}(n), k) & \text{otherwise} \end{cases} \tag{B.1}$$

Note, this function treats its argument as the zeroth ancestor, allowing n to be returned as its result.

We introduce a helper function, NAO, to find the nearest ancestor of a given node n' in P' with a matching node in P, and returns that matching ancestral node:

$$\text{NAO}(n') = \begin{cases} \text{NAO}(\text{Parent}(n')) & \text{if } (\bot, n') \in M_{node} \\ n & \text{if } (n, n') \in M_{node} \end{cases} \tag{B.2}$$

We introduce a function, Modified, that accepts an edit and returns the nearest node of a certain kind k within P to which a given edit e was applied:

$$\text{Modified}(e, k) = \begin{cases} \{\text{NAK}(n, k)\} & \text{if } e = \text{Delete}(n) \\ \{\text{NAK}(n, k)\} & \text{if } e = \text{Update}(n, \ell') \\ \{\text{NAK}(\text{NAO}(n'), k)\} & \text{if } e = \text{Insert}(n', p', k) \\ \left\{ \begin{array}{l} \text{NAK}(n, k), \\ \text{NAK}(\text{NAO}(n'), k) \end{array} \right\} & \text{if } e = \text{Move}(n, n') \end{cases} \tag{B.3}$$

Statement-Related Actions

$$\left\{ \text{DelStmt}(stmt) \;\middle|\; \begin{array}{l} stmt \in S \\ parent = \text{Parent}(stmt) \\ (stmt, \bot) \in M_{node} \\ (parent, parent') \notin M_{node} \\ parent' \neq \bot \end{array} \right\}$$

To detect statement deletions, we iterate through each of the statements in the original AST (i.e, $stmt \in S$) and check that there is no matched node for that statement within the modified AST (indicating it has been deleted). Redundant statement deletions (i.e., those implied by the deletion of their parent statement) are removed by ensuring that the parent $parent$ of $stmt$ maps to a node in the modified AST.

Next, we mine InsertStatement actions, which are used to represent both PrependStatement and AppendStatement actions. As with the detection of statement deletions, redundancy checks are used to ensure that nested actions are modelled using a single insertion.

$$\left\{ \text{InsStmt}(stmt, parent) \;\middle|\; \begin{array}{l} stmt \in S' \\ parent' = \text{Parent}(stmt) \\ (\bot, stmt) \in M_{node} \\ (parent, parent') \in M_{node} \\ parent \neq \bot \end{array} \right\}$$

To simplify the detection of sub-statement-level changes, an intermediary ModifyStatement is introduced, which describes a series of edits applied to a given statement. To avoid the parent of a modified statement from triggering the ModifyStatement action, edits are only associated with their nearest statement in the original form of the AST.

$$\left\{ \text{ModStmt}(s, s', edits) \;\middle|\; \begin{array}{l} (s, s') \in M_{node} \\ s \in S \wedge s' \in S' \\ edits = \{e \in E \mid \text{Modified}(e, \textsc{Stmt}) = s\} \\ edits \neq \emptyset \\ s \neq s' \end{array} \right\}$$

If-Statement-Related Actions

$$\left\{ \text{Wrap}(s, w, g) \;\middle|\; \begin{array}{l} (s, s') \in M_{node} \\ w = \text{If}(g, s, []) \\ (\bot, if) \in M \end{array} \right\}$$

$$\left\{ \text{Unwrap}(s, s') \;\middle|\; \begin{array}{l} s = \text{If}(g, s', []) \\ s \in S_P \wedge s \notin S_{P'} \\ s' \in S_{P'} \end{array} \right\}$$

$$\left\{ \text{ReplaceIfCond}(s, s', c, c') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c, then, else) \\ s' = \text{If}(c', then, else) \\ c \neq c' \end{array} \right\}$$

$$\left\{ \text{ReplaceThen}(s, s', then, then') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c, then, else) \\ s' = \text{If}(c, then', else) \\ then \neq then' \\ then' \neq [] \end{array} \right\}$$

$$\left\{ \text{ReplaceElse}(s, s', els, els') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c, then, else) \\ s' = \text{If}(c, then, else') \\ else \neq else' \\ else' \neq [] \end{array} \right\}$$

$$\left\{ \text{RemoveElse}(s, s', els) \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c, then, els) \\ s' = \text{If}(c, then, []) \\ els \neq [] \end{array} \right\}$$

$$\left\{ \text{InsertElse}(s, s', els) \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c, then, []) \\ s' = \text{If}(c, then, els) \\ els \neq [] \end{array} \right\}$$

$$\left\{ \text{InsertElseIf}(s, s', elif, c_2, then_2) \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(c_1, then_1, []) \\ s' = \text{If}(c_1, then_1, els) \\ elif = \text{If}(c_2, then_2, []) \\ c_2 \neq c_1 \\ then_2 \neq [] \\ (\bot, then_2) \in M \end{array} \right\}$$

$$\left\{ \text{GuardElse}(s, s', g_2) \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{If}(g_1, then, els) \\ s' = \text{If}(g_1, then, els') \\ els' = \text{If}(g_2, els, []) \end{array} \right\}$$

Switch-Related Actions

$$\left\{ \text{RepSwitchExp}(s, s', exp, exp') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{Switch}(exp, blk) \\ s' = \text{Switch}(exp', blk) \\ exp \neq exp' \end{array} \right\}$$

Loop-Related Actions

To detect `RepLoopGuard` repair actions, three intermediate repair actions, corresponding to the different loop types are introduced. `RepLoopGuard` actions are then found by computing the union of the sets for these intermediate repair actions.

$$\left\{ \text{RepForGuard}(s, s', g, g') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{For}(init, g, incr, blk) \\ s' = \text{For}(init, g', incr, blk) \\ g \neq g' \end{array} \right\}$$

$$\left\{ \text{RepWhileGuard}(s, s', g, g') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{While}(g, blk) \\ s' = \text{While}(g', blk) \\ g \neq g' \end{array} \right\}$$

$$\left\{ \text{RepDoWhileGuard}(s, s', g, g') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{DoWhile}(g, blk) \\ s' = \text{DoWhile}(g', blk) \\ g \neq g' \end{array} \right\}$$

$$A_{\text{RepLoopGuard}} = A_{\text{RepForGuard}} \cup A_{\text{RepWhileGuard}} \cup A_{\text{RepDoWhileGuard}}$$

Like RepLoopGuard, RepLoopBody is composed of the union of three intermediate repair actions, which are otherwise ignored for the rest of the analysis.

$$\left\{ \text{RepForBody}(s, s', b, b') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{For}(init, g, incr, b) \\ s' = \text{For}(init, g, incr, b') \\ b \neq b' \wedge b' \neq [] \end{array} \right\}$$

$$\left\{ \text{RepWhileBody}(s, s', b, b') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{While}(g, b) \\ s' = \text{While}(g, b') \\ b \neq b' \wedge b' \neq [] \end{array} \right\}$$

$$\left\{ \text{RepDoWhileBody}(s, s', b, b') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{DoWhile}(g, b) \\ s' = \text{DoWhile}(g, b') \\ b \neq b' \wedge b' \neq [] \end{array} \right\}$$

$$A_{\text{RepLoopBody}} = A_{\text{RepForBody}} \cup A_{\text{RepWhileBody}} \cup A_{\text{RepDoWhileBody}}$$

Assignment-Related Actions

$$\left\{ \text{RepRHS}(s, s', rhs, rhs') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{Assign}(lhs, op, rhs) \\ s' = \text{Assign}(lhs, op, rhs') \\ rhs \neq rhs' \end{array} \right\}$$

$$\left\{ \text{RepLHS}(s, s', lhs, lhs') \;\middle|\; \begin{array}{l} (s, s') \in M \\ s = \text{Assign}(lhs, op, rhs) \\ s' = \text{Assign}(lhs', op, rhs) \\ lhs \neq lhs' \end{array} \right\}$$

Function-Call-Related Actions

As with other types of repair action, an intermediate action is introduced for all repair actions related to function calls. This intermediate action, ModCall, is used to quickly identify that the only modification to a given statement is rooted at a given function call (or, to be more precise, the function call closest to all edits made at that statement).

$$\left\{ \text{ModCall}(s, s', c, c', E) \;\middle|\; \begin{array}{l} \text{ModStmt}(s, s', E) \in A_{ModStmt} \\ call = \text{Call}(t, args) \\ call' = \text{Call}(t', args') \\ call \neq call' \\ (call, call') \in M \\ \forall e \in E \mid \text{Nearest}(e, \textsc{Call}) = call \end{array} \right\}$$

$$\left\{ \text{RepCallTarg}(s, s', t, t') \;\middle|\; \begin{array}{l} \text{ModCall}(s, s', c, c', E) \in A_{\text{ModCall}} \\ call = \text{Call}(t, args) \\ call' = \text{Call}(t', args) \\ t \neq t' \end{array} \right\}$$

$$\left\{ \text{ModCallArgs}(args, args') \;\middle|\; \begin{array}{l} \text{ModCall}(s, s', c, c', E) \in A_{\text{ModCall}} \\ call = \text{Call}(t, args) \\ call' = \text{Call}(t, args') \\ args \neq args' \end{array} \right\}$$

$$\left\{ \text{RepCallArg}(args, arg, arg') \;\middle|\; \begin{array}{l} \text{ModCall}(args, args') \in A_{\text{ModCallArgs}} \\ args = l \oplus arg \oplus r \\ args' = l \oplus arg' \oplus r \\ arg \neq arg' \end{array} \right\}$$

$$\left\{ \text{InsertCallArg}(args, arg') \;\middle|\; \begin{array}{l} \text{ModCall}(args, args') \in A_{\text{ModCallArgs}} \\ args = l \oplus r \\ args' = l \oplus arg' \oplus r \end{array} \right\}$$

$$\left\{ \text{RemoveCallArg}(args, arg) \;\middle|\; \begin{array}{l} \text{ModCall}(args, args') \in A_{\text{ModCallArgs}} \\ args = l \oplus arg \oplus r \\ args' = l \oplus r \end{array} \right\}$$

APPENDIX C

Additional Fault Localisation Results

In Table C.1, we report additional results from our comparison of the effectiveness of various fault localisation approaches in Section 4.3.1.

Scenario	GenProg	Adj. Cov.	P2F	P2F-Cov	Metallaxis	MUSE	Jaccard	Ochiai	Tarantula
ct-openssl-0a2dcb6	1.227	0.142	0.907	0.046	0.533	1.379	1.819	1.655	1.443
ct-openssl-6979583	37.681	75.256	15.958	79.753	8.464	8.333	25.000	14.418	8.588
ct-openssl-8e3854a	0.714	0.106	1.005	0.100	0.730	0.948	0.609	0.801	0.934
ct-openssl-eddef30	66.667	76.036	45.920	77.583	41.667	41.667	54.545	47.804	41.968
sir-gzip-v1-KP_1	6.631	4.455	4.673	6.485	2.598	2.439	4.626	3.813	3.265
sir-gzip-v1-TW_3	9.202	14.211	4.689	15.324	2.097	1.935	6.601	4.380	2.925
sir-gzip-v4-KL_1	2.685	2.727	1.724	3.083	0.989	0.935	3.028	2.367	1.232
sir-gzip-v5-KL_1	0.400	0.326	0.360	0.394	0.268	0.264	0.337	0.317	0.292
sir-gzip-v5-KL_8	2.169	6.585	0.338	4.672	0.887	0.296	4.080	1.568	0.483
sir-sed-v2-AG_17	0.185	0.019	0.211	0.023	0.530	0.186	0.149	0.223	0.226
sir-sed-v3-AG_11	0.573	6.953	0.606	3.086	0.810	0.573	1.923	1.128	0.910

Table C.1: The effectiveness of various fault localisation schemes, measured by the probability of sampling a fixable statement.

Bibliography

Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2007). On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98.

Ackling, T., Alexander, B., and Grunert, I. (2011). Evolving Patches for Software Repair. In *Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1427–1434.

Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. (1993). Incremental Regression Testing. In *Conference on Software Maintenance*, ICSM '93, pages 348–357.

Al-Ekram, R., Adma, A., and Baysal, O. (2005). diffX: An Algorithm to Detect Changes in Multi-version XML Documents. In *Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 1–11.

Arcuri, A. and Yao, X. (2008). A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Congress on Evolutionary Computation*, CEC '08, pages 162–168.

Avizienis, A. (1985). The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501.

B. Le, T.-D., Lo, D., Le Goues, C., and Grunske, L. (2016). A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *International Symposium on Software Testing and Analysis*, ISSTA '16, pages 177–188.

Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. (2014). The Plastic Surgery Hypothesis. In *International Symposium on Foundations of Software Engineering*, FSE '14, pages 306–317.

Black, P. E. (2007). Software assurance with SAMATE reference dataset, tool standards, and studies. In *Digital Avionics Systems Conference*, DASC '07, pages 6.C.1–1–6.C.1–6.

Bloch, J. (2008). *Effective Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition.

Böhme, M. and Roychoudhury, A. (2014). CoREBench: Studying Complexity of Regression Errors. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 105–115.

Cadar, C., Dunbar, D., and Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Conference on Operating Systems Design and Implementation*, OSDI '08, pages 209–224.

Carbin, M., Misailovic, S., Kling, M., and Rinard, M. C. (2011). Detecting and Escaping Infinite Loops with Jolt. In *European Conference on Object-oriented Programming*, ECOOP '11, pages 609–633.

Chandra, S., Torlak, E., Barman, S., and Bodik, R. (2011). Angelic Debugging. In *International Conference on Software Engineering*, ICSE '11, pages 121–130.

Chen, T. Y. and Cheung, Y. Y. (1997). On Program Dicing. *Journal of Software Maintenance: Research and Practice*, 9(1):33–46.

Coker, Z. and Hafiz, M. (2013). Program Transformations to Fix C Integers. In *International Conference on Software Engineering*, ICSE '13, pages 792–801.

Coldewey, D. (2008). Zune bug explained in detail. https://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/. Accessed May, 2017.

de Moura, L. and Bjørner, N. (2008). *Z3: An Efficient SMT Solver*, pages 337–340. ETAPS '08.

Demsky, B. and Rinard, M. (2003). Automatic Detection and Repair of Errors in Data Structures. In *Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 78–95.

Demsky, B. and Rinard, M. (2005). Data Structure Repair using Goal-Directed Reasoning. In *International Conference on Software Engineering*, ICSE '05, pages 176–185.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435.

Downey, A. B. (2011). *Think Stats: Probability and Statistics for Programmers*. O'Reilly Media, 2nd edition.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *International Conference on Software Engineering*, ICSE '13, pages 422–431.

Eiben, A. E. and Smith, J. E. (2015). *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45.

Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, ASE '14, pages 313–324.

Fast, E., Le Goues, C., Forrest, S., and Weimer, W. (2010). Designing Better Fitness Functions for Automated Program Repair. In *Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 965–972.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS '15, pages 171–172.

Fry, Z. P., Landau, B., and Weimer, W. (2012). A Human Study of Patch Maintainability. In *International Symposium on Software Testing and Analysis*, ISSTA '12, pages 177–187.

Gall, H. C., Fluri, B., and Pinzger, M. (2009). Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33.

Gallagher, K., Binkley, D., and Harman, M. (2006). Stop-List slicing. In *International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 11–20.

Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B., and Mei, H. (2015). Safe Memory-Leak Fixing for C Programs. In *International Conference on Software Engineering*, volume 1 of *ICSE '15*, pages 459–470.

Giffhorn, D. and Hammer, C. (2007). An Evaluation of Slicing Algorithms for Concurrent Programs. In *International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 17–26.

Glover, F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research*, 13(5):533–549.

Gupta, R. and Soffa, M. L. (1995). Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. *SIGSOFT Software Engineering Notes*, 20(4):29–40.

Harman, M. and Hierons, R. M. (2001). An Overview of Program Slicing. *Software Focus*, 2(3):85–92.

Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yiz, L. (2000). An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. *Journal of Software Testing, Verification, and Reliability*, 10(3).

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA.

Holland, J. H. (2000). Building Blocks, Cohort Genetic Algorithms, and Hyperplane-Defined Functions. *Evolutionary Computing*, 8(4):373–391.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60.

Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In *International Conference on Software Engineering*, ICSE '94, pages 191–200.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579.

Janssen, T., Abreu, R., and v. Gemund, A. J. C. (2009). Zoltar: A Toolset for Automatic Fault Localization. In *International Conference on Automated Software Engineering*, ASE '09, pages 662–664.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided Component-based Program Synthesis. In *International Conference on Software Engineering*, ICSE '10, pages 215–224.

Joachims, T. (2002). Optimizing Search Engines Using Clickthrough Data. In *International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 133–142.

Jones, J. A. and Harrold, M. J. (2005). Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *International Conference on Automated Software Engineering*, ASE '05, pages 273–282.

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering*, ICSE '02, pages 467–477.

Jones, T. and Forrest, S. (1995). Fitness Distance Correlation As a Measure of Problem Difficulty for Genetic Algorithms. In *International Conference on Genetic Algorithms*, ICGA '95, pages 184–192.

Judge Business School, Cambridge University (2013). Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. `http://www.prweb.com/releases/2013/1/prweb10298185.htm`. Accessed April, 2017.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440.

Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. (2014). MintHint: Automated Synthesis of Repair Hints. In *International Conference on Software Engineering*, ICSE '14, pages 266–276.

Ke, Y., Stolee, K. T., Le Goues, C., and Brun, Y. (2015). Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering*, ASE '15, pages 295–306.

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811.

Kling, M., Misailovic, S., Carbin, M., and Rinard, M. (2012). Bolt: On-demand Infinite Loop Escape in Unmodified Binaries. In *International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 431–450.

Korel, B. and Rilling, J. (1998). Dynamic program slicing methods. *Information and Software Technology*, 40(11–12):647–659.

Kossak, F., Mashkoor, A., Geist, V., and Illibauer, C. (2014). *Improving the Understandability of Formal Specifications: An Experience Report*, pages 184–199. REFSQ '14.

Krinke, J. (2003). Barrier slicing and chopping. In *International Workshop on Source Code Analysis and Manipulation*, SCAM '03, pages 81–87.

Krinke, J. (2004). Slicing, Chopping, and Path Conditions with Barriers. *Software Quality Journal*, 12(4):339–360.

Landsberg, D., Chockler, H., Kroening, D., and Lewis, M. (2015). Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme. In Egyed, A. and Schaefer, I., editors, *International Conference on Fundamental Approaches to Software Engineering*, FASE '15, pages 115–129.

Langdon, W. B. (2015). *Genetically Improved Software*, pages 181–220. Springer International Publishing, Cham.

Langdon, W. B. and Poli, R. (1998). *Fitness Causes Bloat*, pages 13–22. Springer London, London.

Le, X.-B. D., Lo, D., and Le Goues, C. (2016). History Driven Program Repair. In *International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012a). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE '12, pages 3–13.

Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012b). GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Le Goues, C., Weimer, W., and Forrest, S. (2012c). Representations and Operators for Improving Evolutionary Software Repair. In *Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 959–966, New York, NY, USA.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable Statistical Bug Isolation. In *Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26.

Long, F. and Rinard, M. (2015). Staged Program Repair with Condition Synthesis. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 166–178.

Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. In *Symposium on Principles of Programming Languages*, POPL '16, pages 298–312.

Luke, S. and Panait, L. (2002). Fighting Bloat with Nonparametric Parsimony Pressure. In *International Conference on Parallel Problem Solving from Nature*, PPSN VII, pages 411–421.

Lyle, J. R. (1987). Automatic program bug location by program slicing. *International Conference on Computers*.

Maldonado, J. C., Delamaro, M. E., Fabbri, S. C. P. F., Simão, A. d. S., Sugeta, T., Vincenzi, A. M. R., and Masiero, P. C. (2001). Mutation Testing for the New Century. chapter Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation, pages 113–116. Kluwer Academic Publishers, Norwell, MA, USA.

Mann, H. B. and Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60.

Martinez, M. and Monperrus, M. (2013). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205.

Martinez, M., Weimer, W., and Monperrus, M. (2014). Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *International Conference on Software Engineering*, ICSE '14, pages 492–495.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). DirectFix: Looking for Simple Program Repairs. In *nternational Conference on Software Engineering*, ICSE '15, pages 448–458.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, ICSE '16, pages 691–701.

Meng, N., Kim, M., and McKinley, K. S. (2013). LASE: Locating and Applying Systematic Edits by Learning from Examples. In *International Conference on Software Engineering*, ICSE '13, pages 502–511.

Miller, B. P., Fredriksen, L., and So, B. (1990). An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44.

Monperrus, M. (2014). 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 *International Conference on Software Engineering*, ICSE '14, pages 234–242.

Monperrus, M. and Martinez, M. (2012). CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software. Technical Report hal-00769121, INRIA.

Montana, D. J. (1995). Strongly Typed Genetic Programming. *Evol. Comput.*, 3(2):199–230.

Moon, S., Kim, Y., Kim, M., and Yoo, S. (2014a). Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *International Conference on Software Testing, Verification and Validation*, ISSTA '14, pages 153–162.

Moon, S., Kim, Y., Kim, M., and Yoo, S. (2014b). Hybrid-MUSE: Mutating Faulty Programs for Precise Fault Localization. Technical report, KAIST.

Naish, L., Lee, H. J., and Ramamohanarao, K. (2011). A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3):11:1–11:32.

Naish, L., Lee, H. J., and Ramamohanarao, K. (2012). Spectral Debugging: How Much Better Can We Do? In *Australasian Computer Science Conference*, ACSC '12, pages 99–106.

Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002). CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, CC '02, pages 213–228.

Neumann, G., Harman, M., and Poulding, S. (2015). Transformed Vargha-Delaney Effect Size. In *International Symposium on Search Based Software Engineering*, SSBSE '15, pages 318–324.

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering*, ICSE '13, pages 772–781.

Ning, J. Q., Engberts, A., and Kozaczynski, W. V. (1994). Automated Support for Legacy Code Understanding. *Communications of the ACM*, 37(5):50–57.

Nishimatsu, A., Jihira, M., Kusumoto, S., and Inoue, K. (1999). Call-mark slicing: an efficient and economical way of reducing slice. In *International Conference on Software Engineering*, ICSE '99, pages 422–431.

Ochiai, A. (1957). Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Nippon Suisan Gakkai Shi*, 22(9):526–530.

Oliveira, V. P. L., Souza, E. F. D., Le Goues, C., and Camilo-Junior, C. G. (2016). Improved Crossover Operators for Genetic Programming for Program Repair. In Sarro, F. and Deb, K., editors, *International Symposium on Search Based Software Engineering*, SSBSE '16, pages 112–127.

Pan, H. and Spafford, E. H. (1992). Heuristics for Automatic Localization of Software Faults. Technical Report SERC-TR-116-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, USA.

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

Parnin, C. and Orso, A. (2011). Are Automated Debugging Techniques Actually Helping Programmers? In *International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209.

Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. (2009). Automatically Patching Errors in Deployed Software. In *Symposium on Operating Systems Principles*, SOSP '09, pages 87–102.

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The Strength of Random Search on Automated Program Repair. In *International Conference on Software Engineering*, ICSE '14, pages 254–265.

Qi, Y., Mao, X., Lei, Y., and Wang, C. (2013). Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. In *International Symposium on Software Testing and Analysis*, ISSTA '13, pages 191–201.

Qi, Z., Long, F., Achour, S., and Rinard, M. (2015). An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis*, ISSTA '15, pages 24–36.

Renieris, M. and Reiss, S. P. (2003). Fault Localization With Nearest Neighbor Queries. In *International Conference on Automated Software Engineering*, ASE '03, pages 30–39.

Reps, T., Ball, T., Das, M., and Larus, J. (1997). The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '97, pages 432–449, New York, NY, USA. Springer-Verlag New York, Inc.

Rosin, C. D. and Belew, R. K. (1997). New Methods for Competitive Coevolution. *Evolutionary Computation*, 5(1):1–29.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948.

Schulte, E., Fry, Z. P., Fast, E., Weimer, W., and Forrest, S. (2013). Software mutational robustness. *Genetic Programming Evolvable Machines*, 15(3):281–312.

Sidiroglou-Douskos, S., Lahtinen, E., Long, F., and Rinard, M. (2015). Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Conference on Programming Language Design and Implementation*, PLDI '15, pages 43–54.

Silva, J. (2012). A Vocabulary of Program Slicing-based Techniques. *ACM Computing Surveys*, 44(3):12:1–12:41.

Sivagurunathan, Y., Harman, M., and Danicic, S. (1997). Slicing, I/O and the Implicit State. In *International Workshop on Automated Debugging*, pages 59–65.

Smith, E. K., Barr, E., Le Goues, C., and Brun, Y. (2015). Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 532–543, Bergamo, Italy.

Soto, M., Thung, F., Wong, C.-P., Le Goues, C., and Lo, D. (2016). A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *International Conference on Mining Software Repositories*, MSR '16, pages 512–515.

Stolee, K. T., Elbaum, S., and Dobos, D. (2014). Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26.

Takada, T., Ohata, F., and Inoue, K. (2002). Dependence-cache slicing: a program slicing method using lightweight dynamic information. In *International Workshop on Program Comprehension*, IWPC '02, pages 169–177.

Tan, H. B. K. and Ling, T. W. (1998). Correct program slicing of database operations. *IEEE Software*, 15(2):105.

Tan, S. H. and Roychoudhury, A. (2015). Relifix: Automated Repair of Software Regressions. In *International Conference on Software Engineering*, ICSE '15, pages 471–482.

Tan, S. H., Yi, J., Yulis, Mechtaev, S., and Roychoudhury, A. (2017). Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *ICSE '17 Poster*. To appear.

Tan, S. H., Yoshida, H., Prasad, M. R., and Roychoudhury, A. (2016). Anti-patterns in Search-Based Program Repair. In *International Symposium on Foundations of Software Engineering*, FSE '16.

Thung, F., Lo, D., and Jiang, L. (2012). Automatic Defect Categorization. In *Working Conference on Reverse Engineering (WCRE)*, WCRE '12, pages 205–214. IEEE.

Timperley, C. S. (2013). Reflective Method Matching for Object-Oriented Programs. Master's thesis, University of York, York, England.

Timperley, C. S. and Stepney, S. (2014). Reflective Grammatical Evolution. In *ALife XIV*, pages 71–78. MIT Press.

Vargha, A. and Delaney, H. D. (2000). A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

Venkatesh, G. A. (1991). The Semantic Approach to Program Slicing. *SIGPLAN Notices*, 26(6):107–119.

Walcott, K. R., Soffa, M. L., Kapfhammer, G. M., and Roos, R. S. (2006). Time-Aware Test Suite Prioritization. In *International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12.

Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. (2010). Automated Fixing of Programs with Contracts. pages 61–72.

Weimer, W., Fry, Z. P., and Forrest, S. (2013). Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering*, ASE '13, pages 356–366.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering*, ICSE '09, pages 364–374.

Weiser, M. (1979). *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan.

Weiser, M. (1981). Program Slicing. In *International Conference on Software Engineering*, ICSE '81, pages 439–449.

Willmor, D., Embury, S. M., and Shao, J. (2004). Program Slicing in the Presence of a Database State. In *International Conference on Software Maintenance*, ICSM '04, pages 448–452.

Xuan, J., Martinez, M., DeMarco, F., Clément, M., Marcote, S. L., Durieux, T., Berre, D. L., and Monperrus, M. (2017). Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 43(1):34–55.

Yan, X. and Han, J. (2002). gSpan: Graph-Based Substructure Pattern Mining. In *International Conference on Data Mining*, ICDM '02, pages 721–.

Yoo, S. (2012). Evolving Human Competitive Spectra-Based Fault Localisation Techniques. In *Search Based Software Engineering*, Lecture Notes in Computer Science, pages 244–258. Springer Berlin Heidelberg.

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.