Dissertation

ASSESSMENT AND IMPROVEMENT OF AUTOMATED PROGRAM REPAIR MECHANISMS AND COMPONENTS

Submitted by

Fatmah Yousef Assiri

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2015

Abstract

ASSESSMENT AND IMPROVEMENT OF AUTOMATED PROGRAM REPAIR MECHANISMS AND COMPONENTS

Automated program repair (APR) refers to techniques that locate and fix software faults automatically. An APR technique locates potentially faulty locations, then it searches the space of possible changes to select a program modification operator (PMO). The selected PMO is applied to a potentially faulty location thereby creating a new version of the faulty program, called a *variant*. The variant is validated by executing it against a set of test cases, called *repair tests*, which is used to identify a repair. When all of the repair tests are successful, the variant is considered a *potential repair*. Potential repairs that have passed a set of regression tests in addition to those included in the repair tests are deemed to be *validated repairs*.

Different mechanisms and components can be applied to repair faults. APR mechanisms and components have a major impact on APR effectiveness, repair quality, and performance. APR effectiveness is the ability to find potential repairs. Repair quality is defined in terms of repair correctness and maintainability, where repair correctness indicates how well a potential repaired program retains required functionality, and repair maintainability indicates how easy it is to understand and maintain the generated potential repair. APR performance is the time and steps required to find a potential repair.

Existing APR techniques can successfully fix faults, but the changes inserted to fix faults can have negative consequences on the quality of potential repairs. When a potential repair is executed against tests that were not included in the repair tests, the “repair” can fail. Such failures indicate that the generated repair is not a validated repair due to the introduction of other faults or the generated potential repair does not actually fix the real fault. In addition, some existing techniques add extraneous changes to the code that obfuscate the program logic and thus reduce its maintainability. APR effectiveness and performance can be dramatically degraded when an APR technique applies many PMOs, uses a large number of repair tests, locates many statements as potentially faulty locations, or applies a random search algorithm.

This dissertation develops improved APR techniques and tool set to help optimize APR effectiveness, the quality of generated potential repairs, and APR performance based on a comprehensive evaluation of APR mechanisms and components. The evaluation involves the following: (1) the PMOs used to produce repairs, (2) the properties of repair tests used in the APR, (3) the fault localization techniques employed to identify potentially faulty statements, and (4) the search algorithms involved in the repair process. We also propose a set of guided search algorithms that guide the APR technique to select PMO that fix faults, which thereby improve APR effectiveness, repair quality, and performance.

We performed a set of evaluations to investigate potential improvements in APR effectiveness, repair quality, and performance. APR effectiveness of different program modification operators is measured by the percent of fixed faults and the success rate. Success rate is the percentage of trials that result in potential repairs. One trial is equivalent to one execution of the search algorithm. APR effectiveness of different fault localization techniques is measured by the ability of a technique to identify actual faulty statements, and APR effectiveness of various repair test suites and search algorithms is also measured by the success rate. Repair correctness is measured by the percent of failed potential repairs for 100 trials for a faulty program, and the average percent of failed regression tests for N potential repairs for a faulty program; N is the number of potential repairs generated for 100 trials. Repair maintainability is measured by the average size of a potential repair, and the distribution of modifications throughout a potential repaired program. APR performance is measured by the average number of generated variants and the average total time required to find potential repairs.

We built an evaluation framework creating a configurable mutation-based APR (MUT-APR) tool. MUT-APR allows us to vary the APR mechanisms and components. Our key findings are the following: (1) simple PMOs successfully fix faulty expression operators and improve the quality of potential repairs compared to other APR techniques that use existing code to repair faults, (2) branch coverage repair test suites improve APR effectiveness and repair quality significantly compared to repair test suites that satisfy statement coverage or random testing; however, they lowered APR performance, (3) small branch coverage repair test suites improved APR effectiveness, repair quality, and performance significantly compared to large branch coverage repair tests, (4) the Ochiai fault localization technique always identifies seeded faulty statements with an acceptable performance, and (5) guided random search algorithm improves APR effectiveness, repair quality, and performance compared to all other search algorithms; however, the exhaustive search algorithms is guaranteed a potential repair that failed fewer regression tests with a significant performance degradation as the program size increases. These improvements are incorporated into the MUT-APR tool for use in program repairs.

ACKNOWLEDGEMENTS

First and above all, I thank God for giving me the opportunity and the strength to successfully complete my doctoral program.

I would like to express my special appreciation to my academic adviser, Dr. James M. Bieman, for guiding and supporting me throughout my Ph.D. program. He has always believed in me and encouraged me to keep going. He has been patient and happy to help me to be a better researcher. In addition, I would like to thank my Ph.D. committee: Dr. Sudipto Ghosh, Dr. Robert France, and Dr. Gelard Callahan, for serving as my committee members and for providing valuable comments and suggestions that helped me to improve my research. Special thanks to Dr. Sudipto Ghosh for his willingness to give me feedback to improve my experiments, writing, and presentations. My thanks as well to May Abdelsattar, my academic adviser at SACM, for finishing paperwork quickly for me. She has been my adviser since 2006; over the years, she has also been a caring friend.

I would like to thank the Computer Science department staff, particularly Kim and Sharon, for helping provide a pleasant, convenient work environment for my studies. In particular, I am grateful to the Systems Administrator group–Wayne, Dieudo, Rahul, and Mikhail–for their help in fixing my computer issues and providing extra resources as needed.

To my family, especially my parents, Mr. Yousef Assiri and Mrs. Asha Neyb: Thank you for supporting and trusting me. You sacrificed your own interests for my dreams. My success was not achievable without your presence in my life. Mom, you are my greatest inspiration. Thanks to the kindest brother, Jaber, for leaving everything behind, holding my hands, and being a great support; without your sacrifices, this journey would not have been possible. Thanks to my sisters, Bushra, Nura, Huda, and Lama, for the unconditional love and support; you have been the source of my strength. Special gratitude to my twin, Hajer, who has lit up my life and has always made me feel at home. Thanks for making my life such a joy. I would also like to thank my uncle, Mr. Ali Neyb, for believing in me before I even realized that earning a Ph.D. was a dream of mine.

To my friends, who have always stood by my side and shared my happiness, I extend my appreciation. In particular, thanks to Aljohara for being such a true friend since our first year of college, and for understanding me always. Thanks as well to Abeer, for your exceptional friendship, and for being next to me when no one else was there. In addition, I am grateful to Dr. Aritra Bandyopadhyay for our great discussions, and your willingness to discuss research ideas and share experiments data. Thanks also to Tagreed, a true friend who came into my life when I most needed a friend. You have always helped me to see the bright side of my life. Last, but not least, thanks to Rasha, Duaa, Amaal, Areej, Fatmah, Esra, Waad, Reem, Entisar, Kawthar, Wasmeia, and Afdal for the great friendship and the warmth of your love.

DEDICATION

To the greatest parents, my mother, Mrs. Asha Neyb; and my father, Mr. Yousef Assiri; to my only brother, Mr. Jaber Assiri; and to my beautiful sisters Mrs. Bushra Assiri, Mrs. Nura Assiri, Miss. Hajer Assiri, Miss. Huda Assiri, and Miss. Lama Assiri for the endless love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Debugging is a process that includes locating software faults and fixing them. Producing and maintaining bug-free software generally requires time and labor-intensive debugging. The costs of testing, debugging, and verification have been estimated to be 50% to 70% of total development cycle costs [1]. Automated approaches promise to reduce debugging costs.

1.1. Automated Program Repair (APR) Overview

Automated program repair (APR) refers to techniques that automatically locate and fix faults, and promises to dramatically reduce debugging cost. APR techniques take a faulty program and a set of repair tests, and produce a repaired program. APR techniques consist of three main steps: fault localization (Step 1), variant creation (Step 2), and variant validation (Step 3). Figure 1.1 describes the overall organization and activities of APR techniques.

First, an APR technique locates faults (Step 1 in Figure 1.1) by applying a fault localization technique. Fault localization techniques, such as Ochiai and Tarantula, locate potentially faulty statements in the source code by computing a *suspiciousness score* for each statement which indicates its likelihood of containing a fault. Then, statements are ordered based on their suspiciousness, creating a list, which is called a *list of potentially faulty statements* (LPFS). An LPFS contains statements with a suspiciousness score greater than zero to be used by the repair tool. An APR technique fixes faults (Step 2 in Figure 1.1) by modifying a faulty program using a set of program modification operators (PMOs) that change the code in the faulty statement to generate a new version of the faulty program, which is called a *variant*. An APR technique applies a search algorithm to select a PMO;

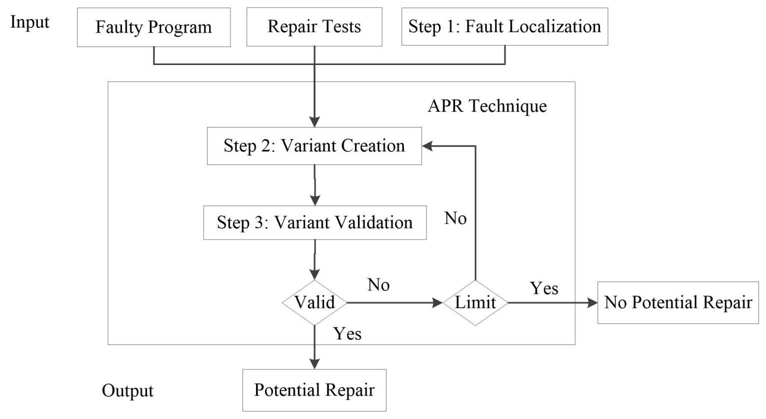


FIGURE 1.1. Overall Automated Program Repair (APR) Technique

some search algorithms run for multiple iterations and in some cases APR techniques generate a variant from a variant produced in prior iterations. The variant is validated (Step 3 in Figure 1.1) by executing it against a set of repair tests, regression tests, or formal specifications. The variant is called a *potential repair* or *potential repaired program* if it passes all of the repair tests. The repair process stops when it finds a potentially repaired program, or when the number of iterations have reached a limit. A potential repair is called a *validated repair* when it passes a set of tests (often regression tests) that were not included in repair tests.

Recent work has been directed towards automatic program repair (APR). Debroy and Wong [2, 3] propose the use of mutations through a brute-force search and a fault localization technique to automate fault fixing. Nguyen et al. [4] describe *SemFix*, which is a tool that uses Tarantula to locate faults, then employs symbolic execution and program synthesis to fix faults. Program syntheses are applied in a predefined order. Wei et al. [5] fix faults in Eiffel programs equipped with contracts. Object states are derived for passing and failing executions, then the states are compared to locate the cause of failure. A behavioral object model defines a series of method calls to change the object state from a failing state into a passing one. Kim et al. [6] repair faults by using built-in patterns. Ten patterns are created based on common patches written by humans, and used to create fix templates. Faults are located using a basic Weighting Scheme [7], then faults are fixed using the templates through an evolutionary algorithm. APR techniques are also used to fix faults for executable software [8, 9]. Evolutionary computing and genetic programming have been adapted to repair faults in C software [10, 7, 11, 12], Java [13, 14], and Python [15], and to help satisfy non-functional requirements [16, 17]. Of particular note is the GenProg tool, which uses genetic programming to modify a program until it finds a variant that passes all repair tests [10, 7, 11, 12]. GenProg was used to successfully fix the well known Microsoft Zune bug date error, which froze Microsoft devices in 2008 due to an infinite loop that occurred on the last day of a leap year [18].

Different mechanisms and components can be applied to fix faults including the following: (1) the program modification operators (e.g., the use of existing code, and the use of simple program syntactic changes), (2) properties of repair test suites to produce repairs (e.g., test suites of different types and sizes), (3) fault localization techniques (e.g., GenProg basic Weighting Scheme, Tarantula [19, 20], or Ochiai [21]), and (4) search algorithms (e.g., brute-force, or stochastic search algorithms). APR components and mechanisms have a major impact on APR effectiveness, repair quality, and performance.

APR effectiveness is the ability to find potential repairs. Repair quality is defined in terms of repair correctness and maintainability, where repair correctness indicates how well a potentially repaired program retains the required functionality, and repair maintainability indicates how easy it is to understand and maintain a potential repair. APR performance is an external measurement that is computed in terms of time and steps required to find potential repairs.

1.2. Problem

We examine how the choice of program modification operators, repair test suites, fault localization techniques, and search algorithms affect APR effectiveness, repair quality, and performance.

Program modification operators (PMO). The set of PMOs used by an APR technique has a major impact on the effectiveness of APR and the quality of potential repairs.

The set of PMOs impact APR effectiveness. An APR technique fixes faults that are only related to the supported PMOs. The GenProg tool can repair faults in programs only when code that represents a valid repair exists somewhere in the program being repaired. Even though GenProg fixes a variety of C faults [7, 12], it fails to fix simple faults such as operator faults. Ackling et al. [15] repair faults in relational operators and constants. Debroy et al. [2, 3] and Nguyen et al. [4] fix faults in binary operators such as relational, logical, and arithmetic operators. Limiting the number of PMOs will limit the fault types that can be fixed by each approach, thereby reducing APR effectiveness.

The set of PMOs also impact repair quality. Existing APR techniques can successfully fix faults, but generated fixes can have negative consequences. When a potential repair is executed against test inputs that were not included in the repair tests, the "repair" can fail. Such failures indicate that the generated repair is not a validated repair due to the introduction of other faults or the generated potential repair does not actually fix the real fault. APR techniques also add many extraneous changes to the code that can obfuscate program logic, thus reducing software maintainability. For example, the GenProg tool, which was developed by Weimer and his colleagues [10, 7], applies three PMOs—*insert*, *delete* and *replace* statements—which make use of existing code in the subject program. Although this approach often works, most potential repairs fail when a "repaired" program is executed with different test inputs. In addition, GenProg inserts many irrelevant changes to the code, thus reducing software maintainability. GenProg generated a potential repair for the faulty program in Figure 1.2, producing the program in Figure 1.3. Unfortunately, this “repair” fails on different test inputs, and it did not repair the actual program faults, a single faulty operator. In addition, GenProg fixes the fault by making three changes (Figure 1.3): (1) it adds *a-=b* after declaring variable *a* (line 3), (2) it adds an empty *else* block after the faulty *if* statement (line 9 and 10), and (3) it copies the whole *if* block after statement *a-=b* (lines 14-17). It is not easy for a programmer to identify such changes used to fix faults, and it will be difficult to maintain and understand this generated code. A study by Fry et al. [22] compares the maintainability of human-written to machine-generated software patches. They find that machine-generated patches reduce software maintainability.

```
void gcd (int a , int b) {
if (a < 0)                   //fault, should be ==
 { printf("%g\n", b);
   return 0;
 }
while (b != 0)
  if (a > b)
    a = a - b;
  else
    b = b - a;
printf("%g\n", a);
return 0; }
```

FIGURE 1.2. Faulty program: *gcd.c*

Repair tests. A set of repair tests is one component of an APR technique; it must contain both passing and failing tests. Passing tests execute required program functionality, and failing tests execute the faults. Le Goues et al. [12] assert that “test suite selection is important to both scalability and correctness.”

```
void gcd (int a, int b) {
{ a = (double )tmp;
 a -= b; }                   // inserted a-=b
b = (double )tmp_0;
if (a < (double )0) {
  printf("%g\n", b);
  return (0);
  }
else {                        //inserted empty block
    }
 while (b != (double )0)
  { if (a > b)
   { a -= b;
     if (a > b)              //inserted if block
       a -= b;
     else
       b -= a;
   }
   else
     b -= a;
 }
 printf("%g\n", a);
 return (0);}
```

FIGURE 1.3. GenProg potential repair for the fault in Figure 1.2 in *gcd.c*

Passing tests protect required functionality by preventing program locations that are executed by passing tests from getting modified, and failing tests guide the search toward program locations where faults hide in order to be repaired. Thus, the selection of repair tests can impact APR effectiveness and the quality of potential repairs. On the other hand, the number of repair tests can impact APR performance. Le Goues et al. [23] note that executing repair tests "dominates GenProg's run-time." Repair tests are executed to validate each generated variant, and thus the number of executions during the APR process depends in part on the number of tests used to repair faults. More tests therefore degrade APR performance. For example, if n is the number of repair tests used and m is the number of generated variants, then the number of executions involved until a potential repair is found is equal to $n * m$. Thus, as the size of the set of repair tests increases, APR performance is decreased.

Existing approaches use small numbers of repair tests (e.g., a set of five tests), which require few executions and obtain good performance, but they sacrifice the quality of potential repairs (or produce no validated repairs). To overcome this quality issue, using all regression tests can produce higher quality repairs, but will reduce APR performance. To illustrate the potential cost of regression testing, Rothermel et al. [24] reported that running all regression tests for a 20,000 LOC program software took seven weeks. Nguyen et al. [4] studied the effectiveness of an APR approach with different repair test sizes. They found that a large test suite decreases the success rate. Fast et al. [25] studied test suite sampling algorithms to improve fitness function performance, and found a sampling algorithm improved performance of APR by 81%.

Fault localization technique. Fault localization (FL) techniques are employed by APR techniques to guide search algorithms towards statements that are more likely to hide faults than other statements. Thus, applying fault localization technique helps to fix faults faster without breaking other required functionality. If a fault localization technique does not identify the location of the actual fault, the application of an APR technique will not be effective—it will fail to repair the fault. The number of statements in the list and their order affect APR performance. A fault localization technique that marks fewer statements and/or places the actual faulty statement near the front of the LPFS will decrease the number of invalid variants that are generated by an APR technique before generating a potential repair, which will improve APR performance.

Different fault localization techniques have been used with APR techniques to locate potential faults. Weimer et al. [10, 7] apply a simple *Weighting Scheme* that assigns weights to statements based on their execution by passing and failing tests. Higher weights are assigned to statements that are executed only by failing tests, and lower weights are assigned to statements that are executed by both passing and failing tests. They exclude statements that are only executed by passing tests to prevent changing correct statements. Nguyen et al. [4] use the *Tarantula* fault localization technique [19, 20, 26]. Debroy and Wong [2, 3], use Tarantula and *Ochiai* fault localization techniques to rank program statements based on their likelihood of containing faults, and they found that using Ochiai fixed more faults with fewer PMOs compared to Tarantula. Qi et al. [27, 28] evaluated APR effectiveness and performance of different fault localization techniques on GenProg. They found that the *Jaccard* was better at identifying actual faulty locations than other fault localization techniques. We argue that randomness of the genetic algorithm in GenProg might affect the accuracy of the reported results. Even if an fault localization technique accurately locates actual faulty statement, a search algorithm can select program modification operators that do not fix the fault.

Search Algorithms. Search algorithms are used to select a PMO from the space of possible modifications. There are two general search algorithm categories: exhaustive and stochastic searches. An APR technique is most effective with an exhaustive brute-force algorithm since it is guaranteed to repair faults related to one of the PMOs, but a brute-force search algorithm degrades performance, especially when coupled with many PMOs and large programs. Debroy and Wong [2, 3] report that an APR technique using many PMOs and a brute-force search algorithm lowered the APR performance in finding potential repair, due to the large number of possible combinations of potentially faulty statements and PMOs that can be tried before finding a potential repair. On the other hand, an APR technique using stochastic search algorithms can more efficiently search the space of possible modifications for a PMO, but it might never introduce a PMO that fixes the faults, thus reducing APR effectiveness. Additionally, search algorithms that run for more than one iteration might reduce the quality of potential repairs due to the insertion of more than one change to repair single faults.

Weimer et al. [10, 7, 11, 12], Ackling et al. [15], Arcuri [13, 29], and Kim et al. [6] used a genetic algorithm to repair faults. Debroy and Wong [2, 3] fixed faults through a brute-force search algorithm. Qi et al. [30, 31] used a random search algorithm with the GenProg tool. Of the different search algorithms proposed in the search-based software engineering (SBSE) literature [32–34], "none of them is the best on all possible problems" [29]. Both stochastic and exhaustive search algorithms have been used for APR techniques. However, the impact of search algorithms varies and needs to be evaluated for a particular framework [35].

1.3. Approach

Our APR tool is called MUT-APR, which stands for **MUT**ation-based **A**utomated **P**rogram **R**epair. It is an adaptable APR framework that includes a variety of APR mechanisms and components in order to evaluate and optimize effectiveness, repair quality, and performance. Our prototype MUT-APR tool was built by adapting the GenProg tool to repair binary operator faults, and is readily adaptable to allow us to change a variety of APR mechanisms and components.

We apply a set of simple PMOs that insert a syntactic change into a program, which were introduced to automated program repair by Debroy and Wong [2, 3]. Our work focuses on fixing simple operator faults in source code, which is consistent with the competent programmer hypothesis that programmers create programs that are close to being correct [36]. Our PMOs change each operator into sets of alternatives. We focus on fixing faulty binary operators including relational operators, arithmetic operators, bitwise operators, and shift operator in different program constructs (*return* statements, *assignments*, *if* bodies, and *loop* bodies). A study by Purushothaman et al. [37] shows that the probability that a one-line change will introduce a new fault is less than 0.04. Thus, the use of simple PMOs are not likely to create a new fault.

Some properties of the repair test suites can improve APR effectiveness, repair quality, and performance by guiding the repair process toward existing faults, and/or limiting the number of new faults introduced without using an entire set of regression tests. Key test properties are related to the type or the size of the repair test suite. We used different test selection method to generate higher quality repairs: (1) branch coverage test criteria, (2) statement coverage test criteria, and (3) random testing. We also used two different test suite sizes: (1) small repair test suites contain 5-30 test inputs, and (2) large repair test suites contain 80-400 test inputs.

Different fault localization techniques can be employed by an APR technique. We evaluated MUT-APR using four well-known fault localization techniques within MUT-APR: Tarantula [19, 20], Ochiai [21], Jaccard [21], and Optimal [38]. We also used the Weighting Scheme employed by GenProg as a baseline.

We implemented stochastic search algorithms with MUT-APR. MUT-APR can apply three different algorithms: (1) a genetic algorithm, (2) a genetic algorithm without a crossover operator, and (3) a basic random search, which we simply call a random search. A genetic algorithm applies a set of PMOs to modify a faulty program by adding changes to a faulty statement creating *variants*. A calculated fitness value for each variant determines the goodness of the variant. Then, a selection algorithm selects variants with the best fitness values for use in the next generation. A crossover operator combines two variants to generate two new child variants. However, since MUT-APR applies simple PMOs to modify a faulty program, there should be no advantages from applying a crossover operator. To test this thesis, we evaluate APR factors applying a genetic algorithm without a crossover operator. Then, to study the influence of randomness in fault fixing, we apply a basic random algorithm that does not apply either a selection algorithm or a crossover operator, and guarantees that each variant is generated by adding a single change. We also developed a guided version of each of the applied search algorithms. The guided algorithms guide the search to the correct PMO by checking the faulty operator, and only call a PMO from a group of PMOs that contain alternatives of the faulty operator. For example, if a potentially faulty statement contains a $>$ operator, we will select one PMO randomly from a group that contains all $>$ alternatives: <, <=, >=, == and !=. MUT-APR also uses a brute-force algorithm, and an ordered brute-force algorithm. The ordered brute-force algorithm orders PMOs to apply the operators that have more potential to fix faults before other operators, thus improving APR performance. To order PMOs, we used the fault hierarchy identified by Kaminski et al. [39].

We performed a set of evaluations to investigate APR effectiveness, repair quality, and performance. APR effectiveness is measured differently when different components are evaluated. APR effectiveness with different PMOs is measured by the percent of fixed faults and the success rate. Success rate is the percentage of trials that result in potential repairs. One trial is equivalent to one execution of the search algorithm. APR effectiveness, when different fault localization techniques are used, is measured by the ability of a technique to identify the actual faulty statement, and the effectiveness of APR when different repair test suites and search algorithms are used is also measured by the success rate.

We introduced different measures to indicate repair correctness. Repair correctness is measured by the percentage of failed potential repairs (PFR) for 100 trials for each faulty

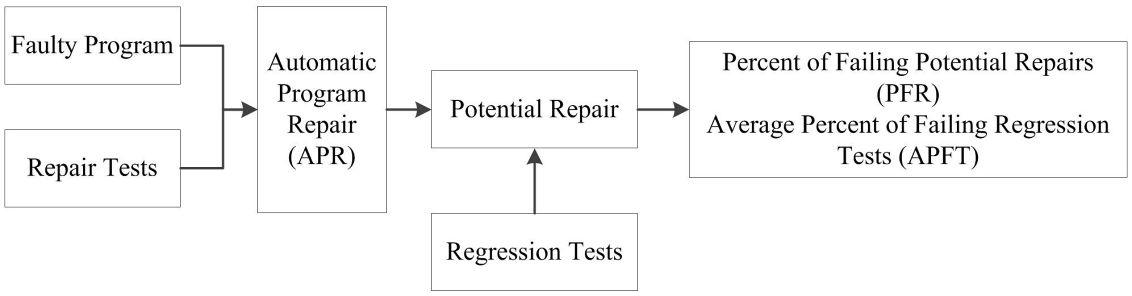

Faulty Program
Repair Tests
Automatic Program Repair (APR)
Potential Repair
Regression Tests
Percent of Failing Potential Repairs (PFR)
Average Percent of Failing Regression Tests (APFT)

FIGURE 1.4. Steps to study repair correctness.

program, and the average percentage of failed regression tests for N potential repairs (APFT); N is the number of potential repairs generated for 100 trials for each faulty program. Figure 1.4 describes steps that we use to study repair correctness. APR requires a set of repair tests generating one or more potential repairs, then we execute potential repairs on a set of regression tests, and compute PFR and APFT. Repair maintainability is measured in terms of acceptability [6] and readability [40]. Le Goues et al. [23] identify the need to develop additional measures for maintainability. To measure repair maintainability, we propose a combination of different measures since the use of multiple measures is a better approach than using a single measure for software maintainability [40]. We define two metrics to estimate repair maintainability. First, we define a static code measure for repair maintainability based on the size of a potential repair. The number of lines of code changed (LOCC) counts the number of LOC modified, deleted, and/or added to fix a fault. A second attribute relevant to repair maintainability is the distribution of modifications throughout a potential repaired program. A wider distribution of repair modifications can have a negative impact on software maintainability.

APR performance is measured by the average number of generated variants (ANGV) and the average total time (ATotalTime) for N potential repairs. NGV, which was defined by Qi et al. [27], is the number of invalid variants generated before finding potential repairs. Invalid variants are generated by modifying a non-faulty location. Total time, measured in seconds, is the sum of the time needed to generate a new variant, compile and execute each generated variant on the repair tests, and compute its fitness values for all variants until producing a potential repair.

We first studied the impact of the PMOs on APR effectiveness and repair quality to fix faulty operators. The evaluation compared repairs generated by different APR techniques (MUT-APR and GenProg) using the same fault localization technique, and the same search algorithm. We also compared the effect of different repair test suites with MUT-APR to investigate the repair test suite properties that improve APR effectiveness and repair quality with an acceptable performance. Then we studied the effectiveness and the performance of MUT-APR with different fault localization techniques to investigate their impacts on both quality factors. The accuracy of fault localization techniques depends in part on the test inputs used to identify faulty statements; therefore, we used five different sets of repair tests with each faulty program. We also used an exhaustive search algorithm to eliminate the randomness that might occur if a stochastic search algorithm is used. Finally, we evaluated the use of different search algorithms. We compared the effectiveness, repair quality, and performance of different search algorithms within MUT-APR by controlling all other independent variables (repair tests and fault localization technique).

1.4. Contributions

The main contributions of this work are to identify and evaluate attributes of alternative APR component and mechanisms in order to improve APR effectiveness, repair quality, and performance:

- Simple Program Modification Operators: The use of simple PMOs to fix faulty operators improve APR effectiveness and produce higher quality repairs than the use of existing code as done by GenProg. MUT-APR fixes 87.03% of faulty operators compared to GenProg which fixes 31.48%, and MUT-APR have a higher average success rate compared to GenProg. In addition, 72.64% of repairs that are generated by the simple PMOs used by MUT-APR are validated repairs that pass all regression tests, while only 3.06% of repairs that are generated by GenProg are validated repairs.

- Properties of Repair Test Suites: Compared to the use of random tests and statement coverage tests, the use of repair test suites that satisfy branch coverage when repairing faults improves APR effectiveness and the correctness of potential repairs. Branch coverage repair tests have a higher average success rate (42.3%) compared to the other two selection methods, and it generate more validated repairs (87.23% validated repairs) compared to statement coverage repair test suites and random testing that generate 73.42% and 75.96% validated repairs, respectively. However, branch coverage repair test suites lowered APR performance compared to the other two test methods. Using small branch coverage repair test suites improved APR effectiveness significantly (average success rate is 42.2% vs. 19%). APR repair quality and performance are also improved significantly using small repair tests: small repair test suites generated more validated repairs (68.7% of repairs are validated repairs) and generated potential repairs that failed fewer regression tests (APFT = 3.5%) compared to large repair test suites. In addition, small repair tests suites required fewer NGV and less time to find potential repairs compared to those used large repair test suites.
- Fault Localization Techniques: The four fault localization techniques were effective in identifying all actual faults except Optimal. Although Optimal did not identify

some faulty statements, it obtained the best APR performance. However, APR performance was noteworthy when Ochiai FL was used since it always assigned actual faulty statements at equal or higher priority for repair than other FL techniques with an average time of 72.5 seconds and an average of 35.5 variants.

- Search Algorithms: Guided random search algorithm (GID-RS) improved APR effectiveness significantly (average success rate is 83%) compared to all other tested stochastic search algorithms. GID-RS also improved APR performance by decreasing the ANGV (an average of 63.9 variants) to find a potential repair compared to other tested search algorithms. The Genetic algorithm (GA) and guided random search algorithm (GID-RS) produced more validated repairs (65% and 61.1% of potential repairs are validated repairs, respectively); however, GID-RS, GAWoCross, and exhaustive search algorithms generated potential repairs that failed fewer regression tests (APFT = 6.5% for GID-RS, APFT = 5.4% for GAWoCross, APFT = 1.5% for Ordered-BF and APFT= 2.4% for BF). Average total time required to find potential repairs improved significantly by GA algorithm. However, GID-RS required an average of two more seconds than GA, and the least efficient algorithm took an average of 61.7 seconds.

In addition, we have developed the MUT-APR evaluation framework to apply the mechanisms and components identified in this research to evaluate and optimize effectiveness, repair quality, and performance.

CHAPTER 2

Related Work

Automated program repair (APR) has attracted considerable attention in the software engineering field. In this chapter, we describe existing APR approaches that use source-code in order to fix faults. We categorize existing approaches into: APR approaches targeting real-world faults (Section 2.1), APR approaches targeting simple faults (Section 2.2), APR approaches that fix faults at runtime (Section 2.3), APR approaches that fix data structure faults (Section 2.4), APR approaches that repair concurrent faults (Section 2.5), and in Section 2.6 we describe alternative approaches (e.g., fix faults via bug reports, or contracts).

Since we focus on the impact of APR mechanisms and components on APR quality factors, we summarize existing approaches based on the key points of variability in APR techniques: (1) identification of faulty locations, (2) automated repair methodology, (3) program modification operators (PMOs), (4) validation of the modified program, and (5) evaluations of APR quality factors: effectiveness, repair quality, and performance.

2.1. Approaches Targeting Real-World Faults

GenProg: In their groundbreaking work, Weimer et al. [10, 7, 41, 11, 12] developed the GenProg tool, which uses genetic programming to fix faults. GenProg takes as input a faulty program and a set of passing and failing tests (five passing tests, and one or two failing tests). Potentially faulty statements are identified by a applying a Weighting Scheme (1 is assigned to statements that are executed by only failing tests, and 0.1 is assigned to statements that are executed by both passing and failing tests). Only three PMOs (insert, delete, and replace statements) are used to modify subject programs, and create program variants as the initial population. Variants are validated by executing them against repair tests, and a fitness value is computed for each variant. Variants that do not compile or variants with fitness equal to zero are discarded. The remaining variants are used for the next generation. To create a population for the next generation, a crossover operator combines information from two parent variants to create two new child variants. The process stops when a variant that passes all test cases is found or the set of parameters has reached its limits. GenProg can fix a variety of faults in C programs including infinite loops and segmentation faults.

Le Goues et al. [42, 43] improved GenProg to scale to larger programs by representing repairs as patches rather than modifications to the abstract syntax tree (AST). Patches represent a variant as a list of modifications. Le Goues et al. also changed the program modification operators, introduced *fix localization*, which is a list of statements used as the source of the fix with respect to the faulty statement, and applied different weighting schemes and crossover operators. These improvements increased the success rate, decreased repair time, and fixed faults that were not fixed by the original work. GenProg has an average success rate of 77% within an average time of 236.5 seconds on open source benchmarks [12].

Schutle et al. [44] extended the original GenProg work [10, 7] to fix faults using assembly code instead of source code. The use of assembly code allows GenProg to support many programming languages. By working with assembly code, GenProg can fix faults that were not fixed by statement-level repairs including faults in declaration types and values assigned to constants. It also requires fewer modifications before a potential repair is found.

GenProg was initially evaluated on ten open source C programs with real faults, and small repair test suites that consist of one failing test and two to six passing tests [7]. On average, 58.7% of the trials found potential repairs. Repair quality was evaluated using the same repair test suites that were used to fix faults. If a potential repair fixes faults, compiles successfully, and does not fail any of the passing tests, it is considered a "good" repair. Le

Goues et al. [43] extended the evaluation of GenProg to include larger programs, to study the impact of the use of different program representation, a crossover operator, and different probabilities of applying PMOs. They found that using a patch program representation improved GenProg effectiveness by 10% compared to the use of an AST. Removing the crossover operator decreased the time required to fix faults but lowered the success rate. Assigning different probabilities of applying PMOs increased the success rate by 8.9% for difficult faults, and improved performance by 40%. Although this approach often works, most generated repairs fail when a "repaired" program is executed with different test inputs. In addition, GenProg inserts many irrelevant changes to the code, thus reducing software maintainability.

RSRpair: Qi et al. [30, 31] applied a simple random search algorithm with GenProg to study the impact of the random search algorithm on APR compared to that of genetic programming. The proposed APR, which is called *RSRepair*, applies GenProg PMOs to modify faulty code, creating variants. Variants are validated by executing them against the repair tests. Unlike GenProg, RSRepair does not compute a fitness function; when a variant fails any test case, the variant is discarded and the process continues to generate more variants. RSRepair runs for multiple iterations until a potential repair is found or the number of iterations has reached a limit. To further improve the efficiency of RSRepair, a test prioritization technique is applied to decrease the number of test case executions required until a fault is fixed [45]. Test cases are ordered based on their effectiveness in detecting invalid variants. Failing tests are executed before passing tests, and a test that causes more variants to fail has a higher priority than other tests that cause fewer variants to fail. The evaluation included seven programs (24 faulty versions) with real faults. It compared APR effectiveness and performance by comparing the results of RSRepair to those found by using

GenProg. RSRepair fixed more faults than GenProg for 16 out of 24 faults, with fewer generated variants. For 23 out of 24 faults, RSRepair required fewer test case executions until a potential repair is found compared to GenProg.

AE:. Weimer et al. [46] proposed a deterministic algorithm, which is called *AE*, to reduce the cost of repairing faults automatically. AE uses the same set of PMOs and the fault localization technique that is used by GenProg. However, the AE algorithm discards variants that are semantically equivalent (e.g., variants that are equivalent after eliminating dead code or variants that are syntactically equal), thus reducing the number of variants that need to be validated. To improve the repair algorithm further, AE first executes failing tests then passing tests, and then executes tests that have greater chance of failure. Additionally, a variant is discarded as soon as it has failed one test case. AE was evaluated by comparing it to GenProg using eight programs with 105 real faults. AE fixed more faults than GenProg (55 vs. 53 out of 105 faults). In addition, AE required an average of 186 test executions while GenProg required an average of 3252 test executions to find a potential repair.

PAR:. Kim et al. [6] described the Pattern-based Automatic program Repair tool (*PAR*), which repairs faults by generating patches using *fix patterns*. PAR uses ten patch patterns based on patches commonly written by humans. Patterns are used to create *fix templates*, which are program scripts that describe how to edit the program. An evolutionary computing algorithm is applied to repair the fault. To generate a new variant, likely faults are located using the fault localization technique used by GenProg [42]. Then, a faulty program is modified using fix templates. Each template analyzes a program's abstract syntax tree (AST), and modifies the program if a faulty statement can be modified by one of the fix templates. Templates modify the program by adding a node, replacing a parameter, or removing a predicate. Generated repairs are validated using regression tests. APR Effectiveness is measured as the number of fixed faults. PAR fixed 27 faults out of 119 in six Java open source projects with real faults. To study repair maintainability, repair acceptability was studied. Human subjects accepted patches that were generated by PAR more than the ones generated by GenProg, and PAR patches were equivalent to repairs done by humans. Kim et al. [6] report that 49% of repairs were accepted by users and developers.

MCShaper: Monperrus et al [47] proposed an approach that extracts PMOs, which they call *repair actions*, using developer changes to fix faults. Developers' changes are found by analyzing software repositories that contain fault fixing. In order to determine the repair actions, three methods are used: *commit texts*, *syntactic features*, and *semantic features*. Commit texts method identifies a group of patterns to fix faults by analyzing commit transactions that include keywords (e.g., fix, bug, patch) following the approach by Pan et al. [48]. A syntactic features method identifies a group of patterns to fix faults by analyzing the commit transactions that change one line of code, and a semantic features method identifies a group of patterns to fix higher-order faults by analyzing commit transactions involving many changes by checking the number and type of changes. Each pattern group is called a *transaction bag*, and each transaction bag consists of a set of repair actions.

A probability distribution model was defined to measure the likelihood that repair actions will repair faults. Two measurements were used to compute the probability of each repair action: (1) the number of occurrences of a repair action in a transaction bag, and (2) the frequency of a repair action over the real fixes. Then, MCSharper, Monte Carlo sharper repair algorithm, was implemented to repair faults based on the probability distribution in order to improve the probability of finding good repairs. MCSharper starts with the number of repair actions. Then, it predicts a tuple of repair actions (e.g., if the number of repair actions given to the algorithm is 2, then a tuple of two actions is created (*StmtInsert*, *StmtDelete*)), and uses a probability distribution for each repair action to guide the search towards the best repair action. The approach was evaluated on 14 open-source java projects, which found that the probability of repair actions differ between transaction bags. However, the probability distributions guided the search towards the repair actions that more likely repair faults, and it was able to fix real bugs in fewer than 1000 attempts.

2.2. Approaches Targeting Simple Faults

Syntactic construct: Kern and Esparza [14] presented a technique to fix faults in Java programs. Developers must give syntactic constructs of faulty expressions called *hotspots*, a set of alternative expressions to fix the fault, and a set of tests. A tool scans the code for expressions that match the hotspots. Then a *changeset* is created to collect information about the hotspots and their alternatives. A template is created from the original program for each hostspot. A new variant is created by replacing hotspots in each template with one of the alternatives in the relevant changeset. Potential repairs are the variants that pass a set of 84 test cases, which represent different arrays of size three, then variants are checked using a model checker to validate potential repairs. This approach was only evaluated on implementations of the Quicksort algorithm that were taken from different domains. Each implementation has an off-by-one error. Effectiveness is measured as the number of fixed faults (four out of ten algorithms were automatically repaired), and the process performance is measured as the time required to find validated repairs. It took an average of 166.1 seconds to fix the faults.

***py*EDB:**. Ackling et al. [15] developed the *py*EDB tool to automate repairs of Python software using genetic programming. *py*EDB returns the program repair as a patch, using

Tarantula to find faulty locations. It selects possible changes for a location from tables that are created before the evolutionary process. Tables map each value to all possible modifications (e.g., > maps to a set that contains >=, <, <= ,==, !=). *py*EDB selects modification operators sequentially. Small sets of repair tests (six to eight tests) were used to validate the variant. The *py*EDB tool only fixes faults in relational operators and constants. Tool effectiveness is measured as the average number of generations to complete a potential repair; *py*EDB took an average 8.6 generations out of a maximum of 50 generations to find a potential repair. Potential repairs generated by *py*EDB rarely introduce new faults. Effects on maintainability were not studied.

JAFF:. Arcuri et al. [13, 29] proposed an approach and a tool for automatic bug fixing. The tool, Java Automatic Fault Fixer (JAFF), uses evolutionary algorithms. JAFF requires either a set of tests or formal specifications to fix a fault. The process assumes that the original program is close to correct. Thus, it uses an initial population of n identical copies of the original program. To decrease the search space, the algorithm is applied to fix individual software methods. JAFF applies PMOs that change the abstract syntax tree (AST) by replacing a sub-tree with a new one, or inserting a new node between a node and its parent to fix simple faults such as operator and constant faults. The modified program is validated by using 1000 test cases that were not used to repair the faults. A fitness function is computed by Equation 1, where $Tests_{pass}$ is the number of passing tests for each created variant, and $\sum_{(v,r)\in T(p)} \mathit{diff}$ is the summation of the difference between the expected result and the variant's output for each test case pair (v, r) in the test suite $T(p)$. For example, if the expected result of a function that sums two variables is $v = 3$, and the created variant gives as output $r = 2$. Then, *diff* is equal to $|v - r|$. If the values of (v, r) are Boolean, then $d\text{iff} = 0$ if they are matched, and $d\text{iff} = 1$ otherwise. If the input values are strings, they measure the edit distance between two strings. This work is limited to programs that act on numerical values.

$$fitness = |Tests_{pass}| + \sum_{(v,r)\in T(p)} \mathit{diff} \tag{1}$$

The effectiveness of the approach was measured by counting the number of fixed faults on seven Java programs taken from the literature; faults are seeded using muJava tool [49]. JAFF fixed five out of eight faults in seven Java programs that are used for software testing and genetic programming studies. Some generated repairs introduce new faults by creating infinite loops. In addition, the approach reduced software maintainability since many irrelevant modifications were applied when fixing faults. JAFF performance is compared when three algorithms are applied: genetic programming, hill climbing, and random search. The results showed that JAFF had the best performance with genetic programming.

Debroy and Wong: Debroy and Wong [2, 3] applied a brute-force search method to repair faults. The Tarantula [26], fault localization technique was used initially to compute a suspiciousness score for each statement and ranked them. Then, the Ochiai fault localization technique was used to compare the efficiency and effectiveness of the two fault localization techniques. First-order PMOs are applied one by one, starting with statements ranked most likely to contain faults, to create a unique mutant. This work supports many PMOs including arithmetic, increment/decrement, and logical operator replacement. Each mutant is checked through string matching. If a mutant does not match the original buggy program, it is considered a "potential repair," then potential repairs is retested against all tests. Effectiveness was evaluated on all Siemens Suite programs and two larger programs: *gzip* (C program) and *ant* (Java program) in the Software-artifact infrastructure repository [50].

They also used the Unix suite [51]. The evaluation included 129 of the faulty versions from the Siemens Suite and 172 from the Unix Suite; only 17.05% and 22.09% of the faults were fixed from the Siemens Suite and Unix Suite, respectively. Of the large programs, two faults out of seven in *gzip* and three faults out of six in *ant* were fixed. To manage the performance, Debroy and Wong proposed to limit the number of PMOs, limit the number of PMOs applied for each statement, or the number of potentially faulty statements. They evaluated the performance by limiting the number of PMOs applied and the number of potentially faulty statement to modify, and they found that limiting the percentage of potentially faulty statements is a good approach for improving performance; more than 50% of faults is fixed by using 10% of potentially faulty statements. Using Ochiai fault localization technique improved effectiveness (one additional fault) and performance (fewer PMOs were required) compared to Tarantula. This work did not evaluate repair quality.

SemFix: Nguyen et al. [4] developed the SemFix tool for fixing faults through semantic analysis. Potentially faulty statements are ranked using Tarantula [26], and highly ranked statements are selected first. Then, for each test input t_i that executes potentially faulty statement si, symbolic execution is used to generate a constraint that makes the program pass the test case. For example, $x > 10$ may be a constraint on statement s_1 that allows the program to pass test t_1. Constraints are derived for all potentially faulty statements. Then, program syntheses are applied sequentially to modify basic components in a potentially faulty statement (e.g., change a constant, change an arithmetic operator, etc) until the statement satisfies its constraints. Program repairs were validated on a set of 50 test cases. SemFix successfully fixed faults in constants, mathematical operators, and relational operators in two different statement types: conditional statements and assignments. The effectiveness of SemFix was evaluated on four Siemens Suite and *grep* subject programs, and 48 out of 90 faults were fixed in an average time of 100 seconds. However, SemFix does not fix operator faults in return statements and loops.

Spreadsheets: Hofer and Wotawa [52] used genetic programming to repair spreadsheet faults. In order to fix these faults, the cell producing a faulty output must be given to the algorithm along with the expected output. Then, the algorithm computes the cone, which consists of the cells that are referenced by the faulty output cell. A set of simple PMOs, such as changing Boolean values and permuting digits, is used to fix faults. Similar to other APRs that apply random search algorithms, a PMO is picked randomly to change the faulty cell. Mutating a cell generates a mutated spreadsheet that is validated by computing a fitness value, which indicates whether the change results in a repaired spreadsheet. The approach was evaluated on 555 EUSES spreadsheets [53], and fixed faults in 131 spreadsheets and took an average of 16.3 seconds.

2.3. Approaches for Fixing Faults at Runtime

clearView: Perkins et al. [8] developed *clearView*, which automates the fixing of faults in deployed software. It uses binary code to automatically generate patches for previous commercial off-the-shelf software faults, and then reuse the generated patches to fix faults in the deployed software without terminate it. First, clearView uses the Daikon [54] dynamic detection tool to identify the invariants during executions. Then, executions are checked using monitors to be marked as passing or failing (two monitors are used to detect out-of-memory bounds and illegal control flow). If a failed execution is found, the location of the fault is determined and the execution is terminated. ClearView uses patches to collect information about a *correlated invariant*, an invariant that is true during a passing run but false during a failing run. Then, the tool generates a set of candidate repairs for each correlated invariant to change the invariant from false to true by changing the locations of memory or changing the control flow.

To fix faults in deployed software, the tool checks the correlated invariant related to the failure. Then, it applies the corresponding patches and observes the program during its execution to find the patch that fixes faults without terminating the software. ClearView effectiveness is evaluated on security vulnerabilities. It generated patches to fix seven out of ten exploits to prevent attacks such as control flow and false positive attacks. It took an average of 4.9 minutes to generate patches that allow applications to execute during attacks. It generated good quality patches that did not introduce new vulnerabilities.

ARMORE:. Carzainga et al. [9] automated the repair of faults in Java software at run-time. Their method utilizes redundancy, represented by a code segment that offers an alternative implementation of the same functionality. *ARMOR* is a tool developed to repair runtime failure in library components. A preprocessor identifies *roll-back areas* (RBA), which are locations that contain the library code that might cause failures during executions. Then, ARMOR creates alternative RBA code using library operations. Alternative RBAs are complied and stored to be used when failure occurs. When runtime failures occur, ARMOR stores the state at the last checkpoint. Then it executes the alternatives one by one to modify the RBA until the failure is avoided. If no alternative code is found to avoid the failure, an exception is thrown. The approach was evaluated on two libraries and four applications. ARMOR repaired three real faults in the JodaTime library, and 48% of the injected PMOs with a maximum of 194% runtime overhead.

Consistency specifications: Demsky et al. [55] developed a tool to repair faults in data structures. First, specifications are constructed automatically by running a program against a set of passing test cases, and a dynamic analysis tool, Daikon [54], is used to generate specifications. Then, the developed tool transforms the generated specifications into C code that is inserted into the faulty program to monitor a data structure for any specification inconsistencies. If a data structure that violated the specifications is found, the data structure is repaired by inserting code that updates the data structure to satisfy the specifications while executing. The study evaluated the approach on three software systems (CTAS, BIND, and Freeciv), and found that automatically generated specifications cover more properties than manually generated specifications. This approach successfully repaired data structure violations that occurred simultaneously and thus reduced program crash times. The repair algorithm took an average of 24.6 milliseconds to repair data structure violations in one program under test. However, this work is limited to fix faults related to data structure violations.

Juzi: Elkarablieh and Khurshid [56] developed a tool, *Juzi*, which inserts code into a data structure and predicate methods to fix data structure violations in Java programs. Juzi creates a Boolean method that takes a data structure and checks it for inconsistency constraints, which are called using an *assert* statements. If an assert fails, Juzi checks the last accessed field in the Boolean method and modify it. Three PMOs are used to modify the corrupt structure: (1) set it to null, (2) set it to a visited node, and (3) set it to an un-visited node. The Boolean method is called after each PMO to check if a PMO fixes the violation. Symbolic execution is used to repair faults in data fields. Symbolic execution is applied to find the path condition for the corrupted field. Then, an integer constraint solver is used to find the correct value. Juzi was evaluated on seven Java programs including Java libraries with 20 seeded faults. The tool fixed all 20 faults in 20 seconds.

2.5. Approaches for Repairing Concurrent Faults

AFix: Jin et al. [57] developed *AFix*, which fixes a single-variable atomicity violation. A single-variable atomicity violation occurs when a shared variable is successively accessed by one thread that is interleaved with another thread (e.g., *thread 1* reads x in line 2 and writes to it in line 6, while *thread 2* writes to variable x between the actions of *thread 1*). AFix automatically generates patches from bug reports. A bug detection tool, called CTrigger [58], is used to predict the single-variable atomicity violation that could occur using the same set of test cases, and generates a bug report by returning the three instructions (preceding(p), current(c), remote(r)) that are involved in each violation. All possible combinations of (p,c,r) tuples are reported. AFix repairs faults using one or more bug reports. To prevent a violation when one bug report is used, the tool acquires locks for all the nodes between p and c through all possible paths, when p and c are instructions in the same functions. When p and c are in different functions, locks are added into a function used by both instructions p and c.

When there are multiple bug reports, AFix checks the generated patch for each report. If two patches protect the same critical region by acquiring different locks, AFix merges the patches. Otherwise, AFix generates different patches for each bug (unmerged patches). Patches are validated using random tests and tests are reported by CTrigger. AFix fixed six out of eight real faults in open source software using merged patches, and fixed five with unmerged patches; however, unmerged patches introduced deadlock. AFix took one second to detect and repair the faults. Merging patches improved software readability in five out of six fixes.

Axis: Lui and Zhang [59] developed *Axis* to fix atomicity violations without introducing deadlocks. First, potentially faulty statements are determined using bug reports. Axis automatically creates a Petri-net model [60] and marks faulty statements. Then, constraints are constructed over the generated Petri-net model. Supervision Based on Place Invariants (SBPI) [61], a constraint solver, is used to modify the Petri net to satisfy the constraints by adding locks that prevent the atomicity violations. An evaluation study compared Axis to AFix on 13 programs including an implementation of the Apache database system and web server platform W3C. It took an average time of one second to fix faults, and it took 30 seconds in the worst case to fix deadlocks. When Axis fixed violations, it introduced deadlocks. However, executing Axis with a deadlock avoidance algorithm generated patches that did not introduce deadlocks.

2.6. Alternative Approaches

In this section we briefly describe APRs that use behavioral models, contracts, and bug reports to repair faults. We also describe a semi-automated approach that generates repairs hints that can guide developers when fixing faults.

PACHIKA:. Dallmeier et al. [62] developed *PACHIKA* to generate fixes by comparing object behavior models, which are finite state machine models of program behavior using object state and method calls for passing and failing runs to determine abnormal behavior. PACHIKA traces executions to get information about the passing and the failing runs. The tool checks the object behavior model of a passing run to get information about the methods' preconditions. Then a failing run is checked for precondition violations. The tool generates a repair by changing the behavior model of the failing run to satisfy the preconditions in two ways: (1) inserting a method call, or (2) deleting the call to the violated method. The repair is validated by inserting the changes into the model of the failing run and checking its outcome. If the changes make the failing run pass, the change is a "potential fix." Then, the potential fixes execute against the regression tests. If the repair passes all tests, it is a "validated repair." Effectiveness is evaluated by counting the number of validated repairs; PACHIKA found validated repairs for three out of 18 faults in the ASPECTJ compiler. To study the quality of the generated fixes, these fixes were compared semantically and syntactically to fixes generated by developers, and PACHIKA fixes were syntactically different from those done by humans.

AutoFix-E:. Wei et al. [5] develop a tool, called *AutoFix-E*, to automate fault fixing in Eiffel programs equipped with contracts. This is similar to the previous work by Dallmeier et al. [62]. Wei et al. create a set of predicates to assess an object's state. The predicate set consists of Boolean queries, relation between the queries, and their mutations. The object state of passing and failing runs are compared using the predicate sets to determine abnormal behavior. The model defines a series of method calls that convert a failing object state from a failure state into a passing one. Then, AutoFix-E creates fix templates that can be used with the information derived from the model. To generate potential repairs, all templates are executed against passing and failing tests to determine the valid ones. Tests are generated using AutoTest [63], and the study used an average of six passing tests, and six failing tests for each repair. Dynamic and static metrics are used to rank the valid fixes based on the changes these fixes add to the code. AutoFix-E fixed 16 out of 42 faults in data structure classes from the EiffelBase and Gobo libraries, and 13 of these fixes did not introduce new faults. In additions, two out of six fixes were not similar to fixes done by humans.

R2Fix: Lie et al. [64] developed an APR tool, called *R2Fix*, which fixes software faults using bug reports. First, bug reports that contain fixes are viewed, and then fix patterns are identified for use as a template to generate repairs. Identified examples of patterns for fixing overflows include increasing the buffer size, and assigning a fewer number of bytes. When a fault is detected by receiving a bug report, the report is classified based on fault types. Then, R2Fix identifies the fix pattern for use in fixing the fault. For each pattern, a set of parameters will be collected such as file name, faulty version, function names, etc. These parameters, along with the fix pattern, are used to generate patches. Patches need to be verified by developers before applying.

R2Fix targets three type of faults: buffer overflows, null pointer, and memory leaks. The evaluation on Linux, Mozilla, and Apache software showed that R2Fix generated an average of 1.33 patches per bug report. For 57 out of 88 patched reports, R2Fix generated at least one patch that is similar or semantically equivalent to patches generated by developers, and 21 out of the 57 patches fixed security vulnerabilities. Developers confirmed that using R2Fix patches saved the time required to fix the related faults.

MintHint: Kaleeswaran et al. [65] proposed a semi-automated approach to generate *repair hints* that are used by developers to fix faults. First, potentially faulty statements are identified using the Ochiai fault localization technique. Then, the state of each potentially faulty statement is derived by taking input values and returning the output values that make all test cases pass (e.g., using symbolic execution in the case of failing test cases to find the input values that make the failed tests pass). In the following step, MintHint searches the faulty programs for expressions that can be used at the location of the fault, and then computes a *likelihood* value for each expression. Likelihood values determine the correlation between the output values of each expression and the expected value. Then, MintHint creates an ordered list of expressions based on their likelihood to be effective in repairing the faulty expression, and each expression is synthesized to create one or more repair hints.

Two type of hints are generated: *simple hints* that require a single syntactic change, and *compound hint* that require more than one change. MintHint fixes five fault types including insertion, replacement, retention, and/or removing expressions. The evaluation study on 10 faulty versions of sed, flex and grep found MintHint generated repair hints for seven faulty versions. The study also evaluated MintHint with ten human subjects. All of the human subjects repaired faults in two hours using the repair hints generated by MintHint, while only six human subjects fixed faults in two hours without using MintHint. Performance was measured for each step involved in MintHint; less than ten minutes were required for correlation analysis and repair hint generation, and none of the steps required more than five minutes.

2.7. Summary

In this chapter we summarized APR approaches that fix general defect types, simple faults, runtime faults, data structures faults, concurrent faults, and other alternative approaches. APR techniques consist of a number of mechanisms and components that vary among them including, but not limited to, the PMOs applied to faulty code, the quality of repair tests, the fault localization techniques, and the search algorithms. The selection of the mechanisms and components can impact APR quality factors.

Existing work studied repair quality in terms of repair correctness and repair maintainability. To ensure correctness, Fast et al. [25] used large repair tests (between 100-200 tests) to find repairs. None of the studies evaluated the effect of repair test suites that satisfy different coverage criterion on repair correctness. Therefore, we investigated the repair test properties that optimize repair quality without decreasing APR effectiveness and performance. To assess repair quality, existing work measures repair correctness by assessing if generated repairs introduce specific types of faults such as deadlocks or security attacks [8, 57, 59], and repair maintainability is measured in terms of understandability, acceptability, and code features (e.g., the ratio of variables out of scope and of variables used per assignment) [6, 40, 66]. However, we found no quantitative metrics to measure and predict repair correctness for general software defects other than security vulnerabilities. Therefore, we identified two metrics to measure repair correctness by estimating the number of failed regression tests for generated repairs, and we identified a metric to measure repair maintainability based on the repair size and modification locations.

Qi et al. [27, 28] evaluated APR effectiveness and performance of different fault localization techniques on GenProg, and Debroy and Wong [2, 3] evaluated the performance of mutation-based APR technique using Tarantula and Ochiai. However, an evaluation of APR effectiveness of different fault localization techniques needs to be done when all other components are fixed [3]. We evaluated different fault localization techniques with MUT-APR applied using a brute-force search algorithm to accurately measure their impacts.

Using many PMOs can improve the effectiveness of APRs by fixing more faults, but it can reduce APR performance. Different stochastic search algorithms can potentially be used to successfully fix faults without reducing APR performance. Qi et al. [30, 31] studied the impact of a random search algorithm on GenProg; however, none of the prior work evaluated the trade-off between APR effectiveness, repair quality, and performance of different search algorithms. Performance is another challenge for APR techniques, and performance problems can be mitigated by using different fault localization techniques [7, 3], and by using a small number of repair tests as proposed in the literature [15, 7]. Arcuri [29] applied APR using different search algorithms to improve performance, and found that APR using genetic programming had the best performance compared to the use of hill climbing and random search. However, “none of them is the best on all possible problems” [29]. Therefore, we evaluated the impact of stochastic and exhaustive search algorithms on APR quality factors within MUT-APR framework. To the best of our knowledge, we are the first to study the impact of different search algorithms on the quality of generated repairs. We also proposed guided search algorithms, and compared their impacts on APR effectiveness, repair quality, and performance to those of the original algorithms.

This dissertation evaluates the combination of PMOs, repair tests properties, fault localization technique, and search algorithm that improve automated program repair to fix operator faults in C programs.

CHAPTER 3

APR Components and Mechanisms

In this chapter we describe the APR components and mechanisms that are involved in our evaluation of APR effectiveness, repair quality, and performance using MUT-APR. First, we describe the program modification operators (PMOs) that we used to modify faulty operators in Section 3.1. Then, we describe the criteria we used to select repair test suites in Section 3.2. We discuss different fault localization techniques that are employed by MUT-APR, and search algorithms that are used in the APR process in Section 3.3 and Section 3.4, respectively.

3.1. Program Modification Operators (PMOs)

MUT-APR makes use of a mutation-based technique adapted from Debroy and Wong [2] to fix simple faults. Simple operator faults are common mistakes made by developers. For example, relational operator faults, such as the use of > instead of >=, can produce off-by-one errors, and many operator faults are security vulnerabilities [67]. Many faults can be fixed by one-line modifications [37, 68].

Faults are fixed in source code by constructing new operators. PMOs are applied to change a faulty operator into a set of alternatives until finding the correct operator. We apply PMOs to fix faults in all statements that contain of binary operators such as relational operators, arithmetic operators, bitwise operators, and shift operators (Table 3.1). The PMOs are as follows: (1) change relational operators in *if* statements, *return* statements, *assignments*, and *loops*, (2) change arithmetic operators, bitwise operators and shift operators in *return* statements, *assignments*, *if* bodies, and *loop* bodies.

PMOs	Description
ROR	Relational Operator Replacement
AOR	Arithmetic Operator Replacement
BWOR	BitWise Operator Replacement
SOR	Shift Operator Replacement

Table 3.1. Program Modification Operators (PMOs) Supported by MUT-APR.

The type of faults that can be fixed by an APR depends in part on the program modification operators supported by the approach. We can now apply fifty-eight program modification operators representing all binary operator transformations to change an operator into its alternatives. Our approach can be extended to apply PMOs to other program constructs such as constants, unary operators, etc. Tables B.1, B.2, B.3, and B.4 in the Appendix summarize the relational PMOs, arithmetic PMOs, bitwise PMOs, and shift PMOs, respectively.

3.1.1. Algorithm.

Algorithm 1 is used in the implementation of our PMOs. MUT-APR selects the first potentially faulty statement from the LPFS, and selects a PMO randomly (line 4). Then, it checks the operator in the selected statement. If the statement *stmti* includes an operator (line 5), the operator is checked (line 6). If the operator matches the selected PMO (line 7) (e.g., *stmti* contains >, and the selected PMO is one of five operators that change > into one of its alternatives), the statement type is checked (line 8), and a new statement *stmtj* is created (line 9). Then *stmti* is substituted by *stmtj* (line 10), and a new variant is created (line 14).

3.1.2. Illustrated Example.

A faulty Euclid's greatest common divisor adapted from an example used by Weimer et al. [7] illustrates our approach. The original fault was a missing statement (line 3). We inserted the missing statement and seeded a fault in the *if* statement in line 2 as shown in Figure 3.1.

Algorithm 1 PMOs Pseudocode to Modify Simple Operators

```
Inputs: Program P and List of Potentially Faulty Statements LPFS
Output: mutated program
for all statements stmti in the LPFS do
   let pmo = choose(ChangeOp1ToOp2),
   if stmti contains an operator then
      let stmtOp = checkOperator(stmti)
      if stmtOp = Op1 then
         let stmtType = checkStmtType(stmti)
         let stmtj= apply(stmti,mOp)
         substitute stmti with stmtj in program P
      end if
   end if
end for
return P with stmti substituted by stmtj
```

The *gcd* program in Figure 3.1 has three relational operators and two arithmetic operators: line 2: *if(a <0)*, line 6: *while(b !=0)*, line 7: *if(a>b)*, line 8: *a = a - b*, and line 10: *b = b - a*. The faulty operator is in the first *if* statement (line 2). In order to fix the fault, the operator in line 2 must be switched to ==.

```
1. void gcd (int a , int b) {
2. if (a < 0)       //fault, should be ==
3.  { printf("%g\n", b);
4.    return 0;
5.  }
6. while (b != 0)
7.   if (a > b)
8.     a = a - b;
9.   else
10.     b = b - a;
11. printf("%g\n", a);
12. return 0;
```

Figure 3.1. *gcd.c* faulty program.

To generate a potential repair for the *gcd* program, a PMO is selected randomly, and all statements in the LPFS are considered sequentially for modification. If the statement includes an operator that matches the selected PMO, a new variant with a new operator is created.

The potentially faulty statements are identified by the fault localization technique. We assume that all three relational operator statements and the two arithmetic operators are identified as potentially faulty statements plus two other statements, and the statement in line 2 is the first statement in the LPFS, thus it is selected first for modification. A PMO is selected randomly to modify the code. The faulty operator in the statement is <, MUT-APR checks the operator in the statement. If the tool selects the PMO that changes < to > for line 2, a new variant is created by constructing a new operator for line 2. The variants are compiled and executed against repair tests. This variant fails two test inputs: (0,55) and (55,0). Since the variant did not pass all repair tests, it is not a potential repair and the process continues. Another statement and PMO are selected. MUT-APR will successfully repair the fault in line 2 when it selects the correct PMO.

3.2. Repair Test Suite Properties

APR requires a set of repair tests that consist of passing and failing tests to repair faults. Passing tests protect program required functionality and prevent an APR tool from changing correct locations; failing tests guide an APR tool towards locations where faults hide so that they can be repaired. Using the entire regression test suite to search for a repair can improve the quality of generated repairs, but will be inefficient. Our aim is to improve the quality of generated repairs without decreasing APR effectiveness and performance. We study two attributes related to repair test selection: the repair test selection method, and the number of repair tests.

3.2.1. Repair Tests Suite Selection Method.

We evaluated the use of a small set of repair tests that satisfy specified coverage criteria as a step toward higher quality repairs

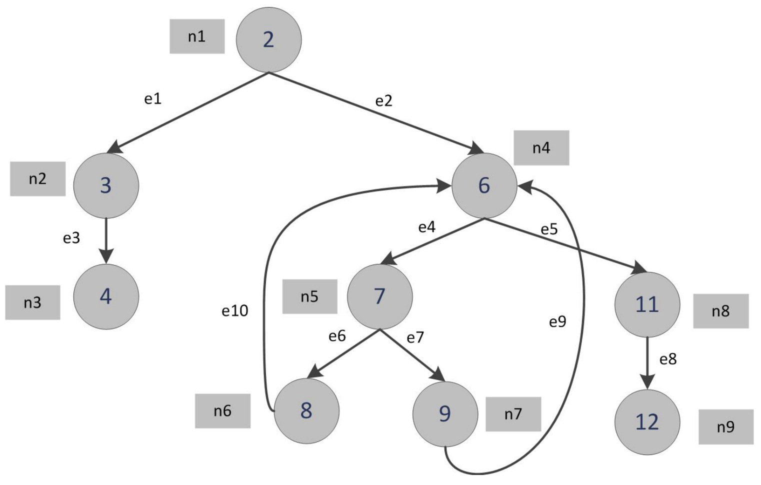


FIGURE 3.2. Flow Graph for *gcd.c* program in Figure 3.1

that does not require the use all regression tests. We examine the use of three repair test selection methods: *branch coverage*, *statement coverage* and *random testing*.

Branch Coverage Test Criterion: For a program P, a test suite satisfies the branch coverage criterion if there is at least one test for each feasible branch in the program under test. To illustrate the branch coverage criterion, we created the flow graph for the *gcd.c* program (Figure 3.2). In the flow graph, nodes represent program statements and edges represent the control flow. We used the line number from Figure 3.1 to represent program statements in the flow graph (e.g, node 2 represents line 2 in Figure 3.1). Test suites that satisfy branch coverage must cause all flow graph to execute: {e1, e2, e3, e4, e5, e6, e7, e8, e9, e10} [69].

Statement Coverage Test Criterion: For a program P, a test suite satisfies statement coverage criterion if there is at least one test for each executable statement. Using the flow graph in Figure 3.2. Statement coverage tests suites must cause all feasible nodes in the flow graph to execute: {n1, n2, n3, n4, n5, n6, n7, n8, n9} [69].

Random Testing: A random test suite contains a set of tests that are selected using a random process.

3.2.2. Repair Test Suite Size. We examined the use of large repair test suites by comparing the repairs of two different repair test suite sizes. In the group of small repair test suites, each test suite contains 5-30 test cases. Each test suites in the group of large repair test suites contains 80-400 test cases.

3.3. Fault Localization (FL) Techniques

FL techniques locate potentially faulty statements in the source code by computing a *suspiciousness score* for each statement that indicates its likelihood of containing a fault. APR tools make use of a list of potentially faulty statements (LPFS), ordered by their suspiciousness. Statements with suspiciousness scores equal to zero are discard from the LPFS since there are more likely not contain faults.

Spectrum-based fault localization (SBFL) [70–72, 20, 73, 21, 74] is a common FL approach; it compares the program behavior of a passing execution to that of a failing execution. SBFL collects information on the dynamic behavior of program statements when they are executed against each test in a test suite. SBFL methods record the number of passing and failing tests executed for each statement, and compute a suspiciousness score for each statement. Statements that are executed more often during a failing run are considered to be more likely to contain faults, thus are assigned a higher suspiciousness score than other statements in the program. Many heuristics have been proposed to compute statement suspiciousness scores [70, 19, 20, 75, 21, 21, 38]. Debroy and Wong evaluated the impact of Tarantula and Ochiai on APR performance, and they identified the need for more controlled experiment [3].

We used the FL technique employed by GenProg [10, 7, 11, 12], called the *Weighting Scheme*, as a baseline. We compared the results of using the Weighting Scheme to four other fault localization techniques: Tarantula, Ochiai, Jaccard, and Optimal. We selected these four FL techniques for the following reasons:

(1) Tarantula was used in prior work with APR techniques [2, 4].
(2) Ochiai was identified as a highly effective FL technique from a developer point of view [21, 76].
(3) Jaccard was identified as the most effective FL technique for APR with GenProg [27].
(4) Optimal is one of the recent proposed heuristics which was found to be more effective than Ochiai and Jaccard from a developer point of view [38].

3.3.1. Fault Localization Heuristics. Each of the evaluated FL techniques applies different heuristics.

GenProg Weighting Scheme: Assigns a weight to each statement based on its execution by passing and failing tests [7]. A weight of one is assigned to statements that are executed only by failing tests, and a weight of 0.1 is assigned to statements that are executed by both passing and failing tests. Statements that are are only executed by passing tests are assigned a weight of zero. The LPFS is an ordered list of statement IDs and their associated weights. The LPFT is sorted so that it orders the statements in descending order. Statements that are executed by only passing tests (with weights of zero) are excluded from the LPFS.

Tarantula: Computes a suspiciousness score for each statement s by Equation 2, where $\%FailedTests(s)$ is the percentage of failing tests that execute statement s, and $\%PassedTests(s)$ is the percentage of passing tests that execute statement s [19].

$$susp\_Turantula(s) = \frac{\%FailedTests(s)}{\%PassedTests(s) + \%FailedTests(s)} \tag{2}$$

Ochiai: Computes a suspiciousness score for each statement s by Equation 3, where $FailedTests(s)$ and $PassedTests(s)$ are the number of failing and passing tests that execute statement s, and $TotalFailedTests$ is the total number of failing test cases [21].

$$susp\_Ochiai(s) = \frac{FailedTests(s)}{\sqrt{TotalFailedTests \times (PassedTests(s) + FailedTests(s))}} \tag{3}$$

Jaccard: Computes a suspiciousness score for each statement s by Equation 4 [21].

$$susp\_Jaccard(s) = \frac{FailedTests(s)}{PassedTests(s) + TotalFailedTests} \tag{4}$$

Optimal: Computes a suspiciousness score for each statement by counting the number of non-executed passing and failing tests [38]. Assuming a faulty program contains single fault, a faulty statement will execute all failing tests, thus the number of non-executed failing tests (nf) for a faulty statement is zero. If there is at least one non-executed failing test for a statement ($nf > 0$), it indicates that the statement is not faulty, and it is assigned a suspiciousness score of -1. If the number of non-executed failing tests is zero ($nf = 0$) for a statement, it indicates that all failing tests executed on the statement, thus the statement is faulty and it is assigned a weight equal to the number of non-executed passing (np) tests for that statement.

3.3.2. Illustrated Example. To illustrate how FL techniques rank program statements using the computed suspiciousness scores, we used the C program in Table 3.2, which computes the Eculid's greatest common divisor. This example uses five test cases $TS = T_1, T_2, T_3, T_4, T_5$ in which T_1, T_2, T_3, and T_4 are passing tests, and T_5 is a failing

Stmt ID	Stmt	T_1	T_2	T_3	T_4	T_5	Sus. Score
	gcd (int a, int b) {						
1	if(a < 0) //fault					✓	1.0
2	{ printf("%g \n ", b) ;						0.0
3	return 0 ; }						0.0
4	while(b ! = 0)	✓	✓	✓	✓	✓	0.5
5	if(a > b)	✓	✓		✓	✓	0.57
6	a = a − b ;		✓		✓		0.0
7	else	✓	✓		✓	✓	0.57
8	b = b − a ;	✓	✓		✓	✓	0.57
9	printf("%g \n ", a) ;	✓	✓	✓	✓		0.0
10	return 0 ;	✓	✓	✓	✓		0.0
	}						

Table 3.2. The dynamic behavior of the faulty program *gcd* when executed against tests in $T1$, ..., $T5$. *Sus. Score* is the suspiciousness score computed using Tarantula.

Statement ID	Suspiciousness score
1	1.0
5	0.57
7	0.57
8	0.57
4	0.5

Table 3.3. List of Potentially Faulty Statements (LPFS) in format used by APR tool.

test, and Tarantula computes suspiciousness score (Equation 2) for each statement. A list of potentially faulty statements (LPFS), which consists of statement IDs and their suspiciousness scores is created and sorted (Table 3.3). The LPFS contains all statements with a suspiciousness score greater than zero, and will be used by the APR tool.

3.4. Search Algorithms

Program modification operators can be selected in a predefined order as done in a brute-force search method, or randomly as done in stochastic search. Brute-force requires an exhaustive search which is inefficient, and it can be infeasible with large programs. We use both exhaustive and stochastic search algorithms to study the impact of search algorithms with MUT-APR. We use stochastic search algorithms to more efficiently search the PMOs space for operators fix faults. We applied three stochastic search algorithms: (1) a genetic algorithm (GA), (2) a genetic algorithm without a crossover operator (GAWoCross), and (3) a random search (RS). We also developed guided versions of the algorithms that guide the APR process toward the program modification operators that fix faults: (1) guided genetic algorithm (GID-GA), (2) guided genetic algorithm without a crossover operator (GID-GAWoCross), and (3) guided random search (GID-RS). Then, we use two versions of exhaustive brute-force search algorithm to compare their results to those of the stochastic search algorithms.

3.4.1. Stochastic Search Algorithms.

We implemented and evaluated three stochastic search algorithms to select the PMO from the pool of PMOs. MUT-APR assigns equal probabilities to all program modification operators, thus the probability of selecting a particular PMO is $p = 1/\textit{Total Number of PMOs}$.

3.4.1.1. *Genetic Algorithm (GA).* Genetic programming is an evolutionary computing search method that evolves computer software by applying a genetic algorithm (GA). A genetic algorithm addresses the combinatorial optimization problems [77]. Genetic programming has been used to automate fault fixing [10, 12, 15, 7, 11].

Algorithm 2, derived from the one used by GenProg, describes how the genetic algorithm in MUT-APR can fix faults. MUT-APR takes a faulty C program, a set of repair tests, a list of potentially faulty statements (LPFS), and a set of parameters: population size (pop_size)), number of generations (gen), and maximum fitness value (max). The initial population, which consists of program variants, of size *pop_size*, is created by mutating a mutable faulty statement (line 3). A mutable faulty statement is a program statement in the LPFS that contains one or more operators that can be mutated by MUT-APR. PMOs are

Algorithm 2 Genetic algorithm Pseudocode

```
 1: Inputs: Program $P$, gen, max, pop_size, LPFS
 2: Output: variant
 3: let $pop$ = $initial\_pop$(P,pop_size)
 4: let $fitness$ = $ComputeFitness(pop)$
 5: let $len$ = List.length(LPFS)
 6: repeat
 7:    let $variants$ = $select(pop)$
 8:    if $size(variants)$ < pop_size/2 then
 9:       let $variants$ = $double(variants)$
10:    end if
11:    for all two variants $(v_{p1}, v_{p2}) \in variants$ do
12:       let $newVariants(v_{c1},v_{c2})$ = $crossover$ $(v_{p1}, v_{p2})$
13:       let $newPop$ = $\mathrm{v}_{c1}$,$\mathrm{v}_{c2}$, $v_{p1}$, $v_{p2}$
14:    end for
15:    let $i = 0$
16:    for all variant in $newPop$ do
17:       let $stmtId$ = $LPFS[i]$
18:       let $pmo$= $choose$(PMO)
19:       let $pop$ = $apply$(variant, $stmtId$, $mpo$)
20:       let $fitness$ = $ComputeFitness(pop)$
21:       if $i$ ! = $len - 1$ then
22:          $i$ ++
23:       end if
24:    end for
25: until $fitness$ = max || $size$(Pop) = pop_size
26: return variant
```

selected randomly from the pool of PMOs, and statements are selected sequentially based on the order given by the LPFS. If a statement is not mutable, or the selected PMO is not one of the alternatives of the faulty operator, the original faulty program is retained. Otherwise, a new version of the faulty program is created. The fitness function is computed for each variant (line 4) to determine if the generated variant is a potential repair or not.

The fitness function is given in Equation 5. $Tests_{pass}$ and $Tests_{fail}$ are the number of passing and failing tests respectively, and W_{pass} and W_{fail} are positive constants that represent the weights of passing and failing tests respectively. Failing tests are assigned a weight of 10 and passing tests are assigned a weight of 1. A variant (v1) that passes failing tests will have a higher fitness value than another variant (v2) that passes all passing tests but not failing tests. Thus, v1 will have higher chance to be used for the next generation than v2. If a variant that maximizes the fitness function (equal to *max*, which is one input to the algorithm) is found, a potential repair is found and the process stops (line 20).

$$fitness = |Tests_{pass}| * W_{pass} + |Tests_{fail}| * W_{fail} \tag{5}$$

If no potential repair is found, variants with a fitness equal to zero or that do not compile are discarded, and the algorithm selects variants with higher fitness values to continue the process (line 7). GA requires *pop_size/2* variants to continue the evolution process. Thus, if the number of remaining variants is less than half the population size *pop_size/2* (line 8), variants are duplicated (line 9) so that the number of variants is equal to *pop_size/2* for use by the crossover operator (line 11-14).

The crossover operator (lines 12) combines information from two variants to create two children. We applied a one-point crossover, which selects a random cut-off point, and swaps the parents' statements after the selected point to create children variants. All parents (v_{p1}, v_{p2}) and children (v_{p1}, v_{p2}) are included in the new population (line 13). Then, a PMO is applied to each variant (line 18) to create the population for the next generation (line 19). Fitness is computed for each member of the population (line 20). If a variant that maximizes the fitness functions is found (line 25), the process stops and the variant is returned (line 26).

The algorithm runs for multiple iterations. Each iteration consists of one GA loop. The genetic algorithm stops when a potential repair is found, or the number of iterations exceeds the upper bound. We set an upper bound on the number of iterations that is equal to *gen* (*gen* is one input to the repair algorithm).

Our GA algorithm is identical to the GA algorithm used by GenProg except that: (1) MUT-APR applies a different set of PMOs than those used by GenProg, and (2) MUT-APR passes the ID of the potentially faulty statement (line 19 in Algorithm 2) to the PMO to guarantee changing an operator in the selected statement.

3.4.1.2. *Genetic Algorithm Without Crossover Operator (GAWoCross).* A genetic algorithm involves a selection algorithm, program modification operators, and the crossover operator. To study the influence of the crossover operator in MUT-APR, we remove the crossover operator from genetic algorithm as done in GenProg [43]. The new algorithm is called GAWoCross. GAWoCross is similar to the genetic algorithm in section 3.4.1.1 except that (1) we do not implement a crossover operator (lines 11-14 from algorithm 2), and (2) the number of variants required to continue the process must be equal to the number of *pop_size* not *pop_size/2* as done in the GA with a crossover operator.

Variants of the initial population are created, and a fitness function is computed as before. Then, a selection algorithm is applied to select the variants with better fitness values to be used for the next generation. If the number of remaining variants (line 8 in algorithm 2) is less than the population size (pop_size), the algorithm creates more copies of the generated variants so that the number of variants is equal to the *pop_size.* Then, PMOs are applied for each variant (lines 16-23 in algorithm 2). The algorithm is repeated for many generations until the number of generations reaches its limit or a potential repair is found.

3.4.1.3. *Random Search (RS).* Our random search does not apply a selection algorithm or crossover operator as done by both GA and GAWoCross. It runs for one iteration, and applies PMOs randomly until a potential repair is found or the population size reaches a set limit. Thus, each variant contains only one change to the original faulty program. Our RS will generate variants by selecting statements sequentially from the LPFS, then apply a PMO.

Our RS algorithm differs than the RS algorithm implemented by Qi et al. [31] as follows: (1) our RS search runs for one generation to ensure each variant contains one change, while their RS search runs for multiple iterations, and (2) our algorithm computes a fitness function to validate variants, while their algorithm runs variants on repair tests and discard variants as soon as one test fails.

Algorithm 3 Random Search Pseudocode

```
1: Inputs: Program $P$, max , pop_size, LPFS
2: Output: variant
3: let $i = 0$
4: let $len$ = List.length(LPFS)
5: repeat
6:     let $stmtId = LPFS[i]$
7:     let $pmo = choose$(PMO)
8:     let $Pop = apply(P, stmtId, pmo)$
9:     let $fitness = ComputeFitness(pop)$
10:    $i$ ++
11:    if $i = len - 1$ then
12:        let $i = 0$
13:    end if
14: until fitness = max || $size$(Pop) = pop_size
15: return variant
```

Algorithm 3 describes how our random algorithm RS in MUT-APR fixes faults. RS selects a statement from the LPFS (line 6). Then, it selects a mutation operator randomly (line 7) to create a new variant (line 8). The variant is created by mutating faulty statements that are mutable. Otherwise, the original program is retained.

Fitness value is computed for each variant be checking if the variant passes all repair tests or not (line 9). The fitness function computes the number of passing and failing tests for each individual (Equation 6). The fitness function for the RS is different than the one used by GA and GAWoCross. RS does not apply a selection algorithm, thus there is no difference between a variant that passes only passing tests and another variant that passes failing tests. Thus, there is no need to assign different weights for passing and failing tests. The process continues until a variant that passes all repair tests is found, or the number of generated variants is equal to *pop_size* (line 14).

$$fitness = |Tests_{pass}| + |Tests_{fail}| \tag{6}$$

3.4.2. Guided Stochastic Search Algorithms. Guided search algorithms guide the search to the correct PMO by checking the operator in the potentially faulty statement and only calling a PMO from a group of PMOs that contain only the alternatives of the faulty operator. Implementing a guided algorithm will increase the probability of selecting a PMO that will fix a fault ($p = 1/NumberofAlternativePMOs$), and thus can improving APR process. Le Goues et al. [43] studied the impact of the use of different different probabilities for applying PMOs operators. They found that assigning different probabilities for applying PMOs increased the success rate by 8.9% for difficult faults, and improved performance by 40%.

We developed three guided search algorithms: (1) guided genetic algorithm (GID-GA), (2) guided genetic algorithm without a crossover operator (GID-GAWoCross), and (3) guided random search (GID-RS). The differences between the guided search algorithms and the algorithms that are described in the previous section (Section 3.4.1) is the selection of PMOs. Before selecting a PMO, the guided algorithm checks the operator in the potentially faulty statement (*stmtOp* = *checkOp*(stmti)). Then, the algorithm calls a group of PMOs that contains *stmtOp* alternatives, and selects one of them randomly to be applied to faulty location.

For example, if a potentially faulty statement contains the > operator, the algorithm will select one PMO randomly from the group that contains all > alternatives: {<, <=, >=, == ,!=}. To further decrease the number of generated invalid variants until producing potential with the guided random search algorithm (GID-RS), the algorithm returns no variant when the selected statement is not mutable.

3.4.3. Exhaustive Search Algorithms.

An exhaustive search algorithm, such as *brute-force*, applies all possible PMOs to the program in a predefined order until a potential repair is found. We developed a brute-force algorithm that orders PMOs randomly (Section 3.4.3.1), and we developed another brute-force algorithm that used a heuristics to order PMOs, which will be described in Section 3.4.3.2.

3.4.3.1. *Brute-Force Algorithm.*

The brute-force algorithm applies all possible PMOs for each mutable statement (a mutable statement is a program construct that can be changed by one of the supported program modification operators) from the LPFS. PMOs are applied in a predefined order. If no potential repair is found by changing the most suspicious potentially faulty statement to all of its possible alternatives (w.r.t. the set of PMOs that are supported by the APR tool), the next statement is modified and so forth. The brute-force search algorithm runs until a potential repair is found or all possible combinations are executed.

Algorithm 4 describes how the brute-force algorithm fixes faults within MUT-APR. It produces a variant (potential repaired program) from a faulty program. A variant is a copy of the faulty program with one modification. The algorithm modifies statements sequentially. It takes the first statement in the LPFS, and checks if it contains an operator (line 4). If the operator is mutable, it applies all possible alternatives (line 5-10). Each change creates a variant (line 7). The fitness value is computed for each generated variant (line 8) by executing it against the repair tests (one of the inputs to the repair process in Figure 1.1).

Algorithm 4 Brute-Force Pseudocode

```
1: Inputs: Program P , List of Potentially Faulty Statements LPFS, and maximum fitness
   value
2: Output: Variant
3: for i=0 to length(LPFS)-1 do
4:     let stmtOp = checkOp(stmti)
5:     for all program modification operators pmo for stmtOp do
6:         repeat
7:             let variant= apply(stmti, stmtOp, pmo)
8:             let variant_fitness= computeFitness(variant)
9:         until variant_fitness= maximum fitness || mOp is the last PMO for stmti
10:    end for
11:    if i != last index in the LPFS then
12:        i++
13:    end if
14: end for
15: return variant
```

If a variant that passes all repair tests is found (in other words, the variant has a fitness value equal to the maximum fitness value), a potential repair is found. If not, the process continues with the next statement in the LPFS until a potential repair is found (line 9) or the algorithm reaches the last statement in the LPFS without a potential repair.

3.4.3.2. *Ordered Brute-Force Algorithm.* We ordered relational PMOs to apply operators with a greater chance of fixing faults before other operators; thus, decreasing the number of PMOs applied in the exhaustive search to find a potential repair. To order PMOs, we used the fault hierarchy identified by Kaminski et al. [39]. The fault hierarchy for all ROR operators from Kaminski et al. study is shown in Figure 3.3.

The fault hierarchy orders the relational operators using a dominance relationship to remove the unnecessary test cases from a test suite. Any test set TS that kills a mutation operator at level 1, based on the dominance relationship, will kill a mutation operator at level 2 and 3 assuming there is at least one test t_i satisfies the detection condition (DC). DC is a test input that will kill the mutant. For example, if a test set TS kills a mutation operator that replaced > operator with != operators, TS will kill the mutation operator that replaces the > operator with < operator assuming there is at least one test input t_1 that satisfies the condition a < b (Figure 3.3-B).

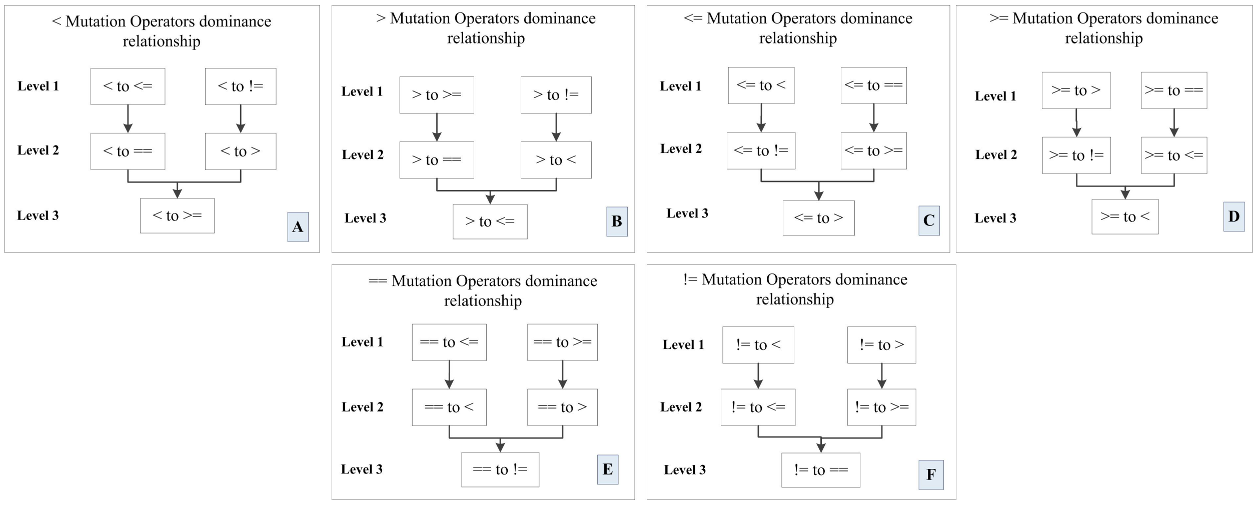


FIGURE 3.3. Class hierarchy represents a dominance relationship between each operator and its mutations [39].

We could not adapt the fault hierarchy to remove PMOs because that might remove the PMOs that can fix a fault. For example, let *(a > 0)* be the faulty operator, and *(a == 0)* or *(a <= 0)* be the correct operators with respect to the repair tests used to repair the faults. Using the fault hierarchy to apply PMOs, we would use the hierarchy in Figure 3.3-B. The PMO from the level 1 right side of the tree is applied (we can start with either side) and the faulty operator is changed into *(a ! = 0)*. For this particular action, the new code will fail some repair tests. Following the dominance relationship (when a test kills a mutation at level 1, mutations at level 2 and 3 will be killed), applying the PMO at level 2 and 3 will fail some repair tests. Thus, we would not apply the < and <= operators. Then, we apply the PMO at level 1 left side of the tree, which changes the faulty operator into *(a >= 0)*, and this will fail some repair tests. Following the dominance relationship, applying PMOs from level 2 and 3 dominated by the applied PMO will fail repair tests, and thereby we do not apply == and <= operators. As a result, we fail to find a potential repair since the two operators that fix the fault (== and <=) have been eliminated using the dominance relationship.

Therefore, we used the fault hierarchy to order PMOs only when using the brute-force search algorithm. We applied all PMOs that correspond to the level 1 mutation operators, then PMOs that correspond to level 2 and 3 mutation operators. The ordering is done only for relational program modification operators. However, the order of arithmetic operators, shift operators and bitwise operators is similar to the brute-force algorithm in the previous section (Section 3.4.3.1). Table B.5 presents the ordering of relational PMOs based on the heuristic.

3.5. Summary

In this Chapter we explained APR components and mechanisms that are used with MUT-APR: PMOs, repair test suites properties, fault localization techniques, and search algorithms. We evaluated the impact of the presented components and mechanisms to help improve APR effectiveness, repair quality, and performance, and the results are presented in Chapter 5. The next chapter describes the implementation of the MUT-APR framework.

CHAPTER 4

Implementation

We built an evaluation framework creating a configurable mutation-based APR (MUT-APR) tool. MUT-APR is a prototype tool that allows us to vary the APR mechanisms and components. The framework was built by adapting the GenProg Version.1 framework, which is implemented in OCaml. We used GenProg due to its availability, and because it is a state-of-the-art automated program repair tool that can fix faults in large C programs. GenProg is a flexible tool that can be easily extended to support different program modification operators, fault localization techniques, and search algorithms. Table 4.1 summarizes the list of methods and classes that are modified in GenProg code to create the MUT-APR tool.

In this chapter we discuss the implementation of MUT-APR components, and we describe how the tool can be used to repair faults. We also discuss limitations of the existing framework.

4.1. MUT-APR Framework

As described in Figure 1.1 in Chapter 1, APR consists of fault localization, variant creation, and variant validation. In this section we describe the implementation of MUT-APR components. Figure 4.1 shows the implemented components for each step in APR technique. Variant creation vary between search algorithms; Figure 4.1 includes the selection algorithm and the crossover operator which are part of genetic algorithm but not other algorithms. Therefore, to implement genetic algorithm without crossover operator, crossover operator is removed, and to implement random search algorithm, selection algorithm and crossover operator are removed.

Method/Class	The Changes
Fault Localization	- MUT-APR applies the changes by Qi et al. [27] to support the use of different FL techniques. - MUT-APR implements a separate function to generate the LPFS using different FL heuristics.
PMOs	- MUT-APR turned off the GenProg's PMOs classes that change code at the statement level, and implements new PMO classes that construct new operators. - MUT-APR sends the potentially faulty statement ID as a parameter to the PMO classes to guarantee changing the operator of the selected statement.
Search Algorithm	Different version of MUT-APR was implemented to apply different search algorithms. We changed the GenProg genetic algorithm code to produce new algorithms - To implement the genetic algorithm without a crossover operator, we turned off the crossover operator method. - To implement the random search, we turned off the selection algorithm and the crossover operator methods. - To implement the guided search algorithms and exhaustive search algorithms, we inserted code to check the operator of the selected potentially faulty statement and send the operator as a parameter to the mutation method.

TABLE 4.1. The GenProg methods and classes that are modified in MUT-APR.

4.1.1. FAULT LOCALIZATION. To locate faults, MUT-APR creates an instrumented version of faulty program and then applies a fault localization technique to create the LPFS (Step 1 in Figure 4.1).

Code Coverage: MUT-APR first collects code coverage information based on the test paths executed by the repair tests. MUT-APR adapted the coverage code from the GenProg tool. The coverage code uses CIL [78], an intermediate language for C programs, which generates simplified versions of source code. Then, an instrumented version of the simplified faulty version is created by assigning a unique ID for each statement. To collect code coverage, an executable version of the instrumented faulty code is created. Then the code is executed against repair tests one by one. For each test case, MUT-APR generates a text

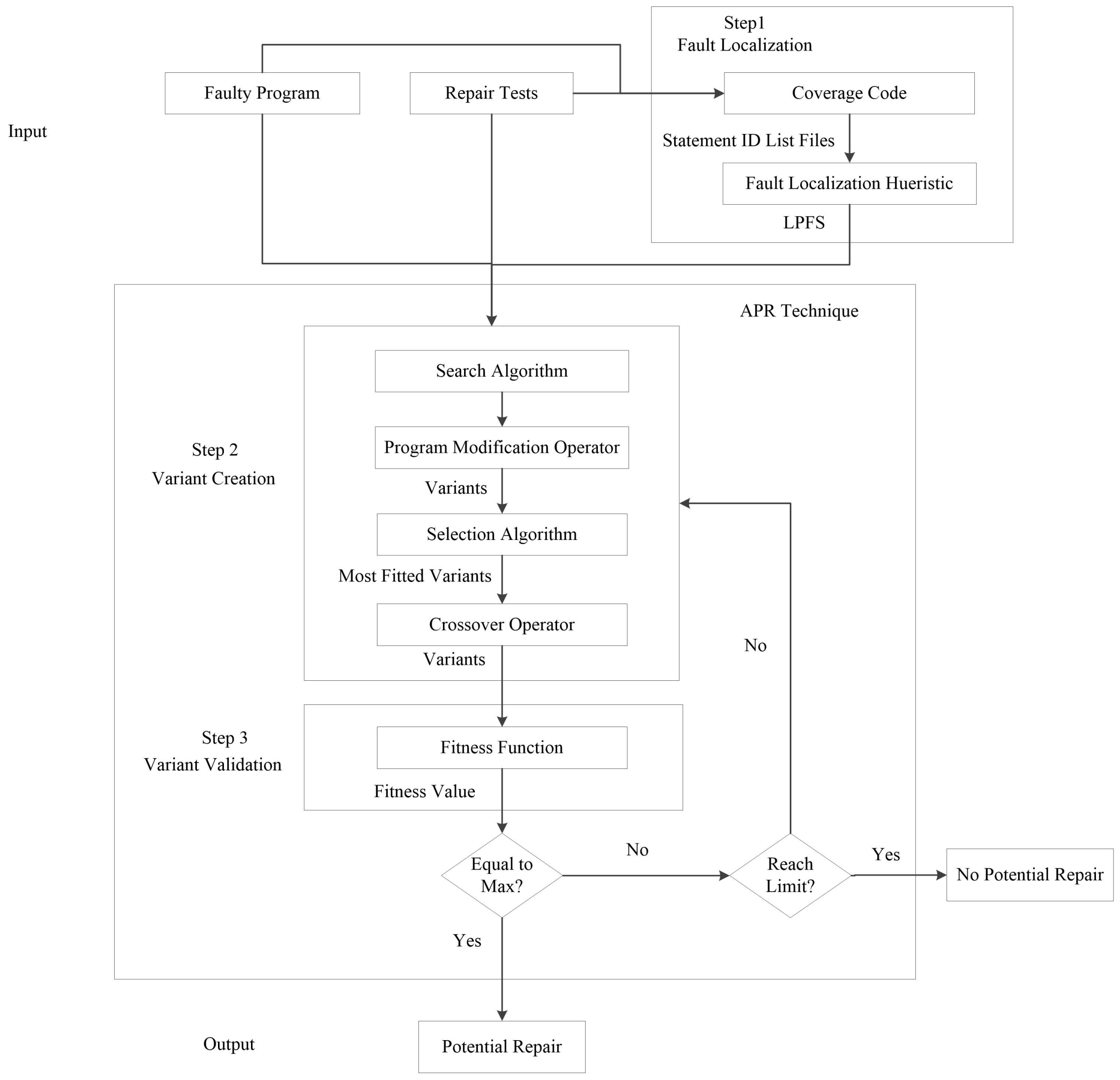


FIGURE 4.1. MUT-APR Implemented Components.

file that contains a unique list of the IDs of the statements that were executed. Coverage information is collected for each test case, creating many statement ID list files (the number of statement ID list files is equal to the number of repair tests) to be used by the fault localization technique to identify potentially faulty statements.

Fault Localization Hueristic: To identify potentially faulty statements, MUT-APR employs many fault localization techniques. All of the employed FL techniques analyze the statement ID list generated in the previous step. Fault localization code reads the statement ID lists and counts the number of times each statement is executed by passing and failing tests. Then, FL computes a suspiciousness score for each statement ID using the number of statement executions, and creates the list of potentially faulty statements (LPFS), which is an ordered unique list of statement IDs and their scores, as shown in Chapter 3-Section 3.3.

MUT-APR initially used the GenProg Weighting Scheme to compute suspiciousness scores, and we used the implementation from the GenProg framework. In order to use different fault localization techniques, we applied the changes developed by Qi et al. [27]. Then, we implemented Jaccard, Ochiai, Optimal, and Tarantula using about 300 LOC of OCaml code.

4.1.2. Variant Creation. MUT-APR creates new variants by applying a set of PMOs to change faulty operators. To select a PMO from the pool of PMOs, a search algorithm is applied. When MUT-APR implements GA and GAWoCross algorithms, it applies a selection algorithm to select the best variant to be used by the next generation and/or a crossover operator to combine changes from two variants into one variant (Step 2 in Figure 4.1).

Search Algorithm: Search algorithm selects a PMO from the set of PMOs. We implemented many versions of MUT-APR; each applies a search algorithm that determines how the PMO will be selected.

We adapted the genetic algorithm from the GenProg framework. To implement a genetic algorithm without a crossover operator, we disabled the crossover operator code used by GenProg. To convert the GenProg genetic algorithm into a random search, we disabled the GenProg selection algorithm and crossover operator. This produces a random search algorithm that only uses the GenProg code that is responsible for creating the initial population for the genetic algorithm, and the program modification operator code to generate a variant.

To implement the guided stochastic and the exhaustive search algorithms, we added code to determine the operator in the selected potentially faulty statement. Then, the algorithm passes the operator as a parameter to the program modification operator code (mutation code). Thus, the program modification code selects one of the faulty operator alternatives randomly to be applied as described in the next section.

Program Modification Operators: To implement our program modification operators, we turned off the original GenProg operators and inserted fifty-eight new program modification operators that represent all binary operator transformations. For each operator (e.g., >), we implemented an OCaml visitor class for each alternative. Each class takes as input the faulty program CIL file, and the potentially faulty statement ID, and returns a new statement by changing the operator into one of its alternatives (e.g., >=, <, <=, ==, ! =). Therefore, each class checks the type of the statement. If the statement type is supported by our tool, a new statement is constructed with a new operator.

The program modification operators are called from *mutation* method, that takes as input a copy of the faulty program, and returns a modified copy of the faulty program. We also implemented another version of the mutation method, which takes one extra operator parameter. The *Operator* parameter passes the potentially faulty operator to guide the mutation method towards a PMO from the group of faulty operator alternatives. For example, if faulty operator is >, *Operator* value is *GT*. Then, the mutation method checks the value of *Operator*, and call one of > alternatives.

Selection Algorithm: For genetic algorithms, a sampling algorithm is used to select the best variants to be used by the next generation, MUT-APR re-used the sampling process from the GenProg framework. First, variants with fitness equal to zero and those that do not compile are discarded, then a tournament sampling algorithm selects the best variants from the remaining ones [79]. Tournament sampling takes a list of pairs (variant, fitness value) and the number of variants to be returned, produces a list of variants. To select the best

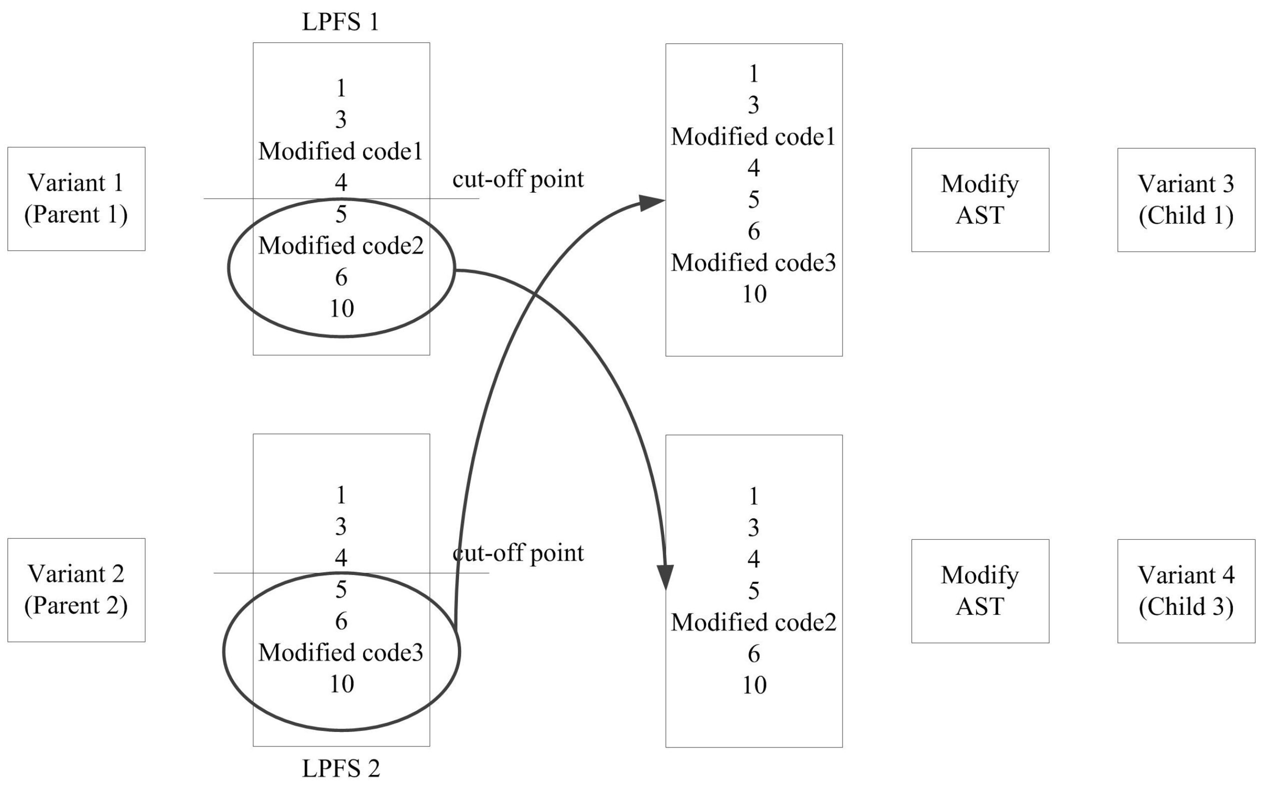


FIGURE 4.2. One-point crossover operator.

variants, the algorithm selects two variants randomly. The variant with the highest fitness value is included in the next population to be used by the crossover operator. The selection process is repeated for all remaining variants.

Crossover Operator: We used the one-point crossover operator from GenProg. The crossover operator takes two parent variants, and returns four child variants: the two newly created variants and the parent variants. To create two new child variants, the crossover operator code takes the LPFS for each variant and determines a cutoff point randomly. Then, the statements after the cut-off point are swapped creating two new LPFS lists, which are sent to a visitor class to create the new variants. Figure 4.2 shows how a one-point crossover operator creates new variants.

4.1.3. Variant Validation.

Each generated variant is validated by executing the variant against repair tests and computing its fitness value using a fitness function.

Fitness function: A fitness value is computed for each variant to determine if the generated variant is a potential repair. It is computed by running the created variant against all repair tests, and the fitness value is computed and variant is cached.

The fitness method takes a variant and returns a fitness value. To compute a fitness value for each variant, each variant is compiled. If a variant fails to compile, the variant is assigned a fitness = 0. If the variant compiles successfully, passing and failing test are executed, and a fitness value is computed by counting the number of passing and failing tests, which are multiplied by a fixed weight. Variants and their fitness values are cached. If a variant is similar to a previously created one, the fitness will not be recalculated, which reduces the number of fitness evaluations.

Each computed fitness value is compared to a maximum value, which is given as an input to the repair algorithm. If a variant has a fitness value equal to the maximum value, it is identified as a potential repair. The fitness class returns the potential repair and the process stops. If no variant maximizes the fitness values, the process is repeated for each created variant until a potential repair is found or the parameters reach the limit.

4.2. Repairing Faults Using MUT-APR

Fixing faults using MUT-APR requires access to the faulty C program source code, and a set of repair tests. A user provides the passing and failing tests, and computes the maximum fitness value that determines if the generated variant is a potential repair or not. First, in a pre-processed step, faults must be identified. In order to identify potentially faulty locations, the user should run the *coverage code* on the faulty program (Step 1 in Figure 4.3) creating the instrumented version of the faulty program (FaultyProgram-coverage.c). The instrumented faulty version is complied (Step 2 in Figure 4.3) creating executable instrumented code

(FaultyProgram-coverage). Then, each test case is executed (Step 3 in Figure 4.3) creating statement ID list files that used to create LPFS.

```
# 1.Create Instrumented version:
   ../coverage FaultyProgram.c > FaultyProgram-coverage.c

# 2.Compile the instrumented version
  gcc -o  FaultyProgram-coverage  FaultyProgram-coverage.c

# 3.Run the instrumented version against test cases
./FaultyProgram-coverage testInput
```

FIGURE 4.3. Linux shell script to collect coverage information on the faulty program.

To create an LPFS, fault localization (FL) code is executed. FL code takes as command-line arguments the number of passing tests, the number of failing tests, the FL technique, and the statement ID list files that are generated in the previous step, and it returns the LPFS. Figure 4.4 is the command line to create an LPFS, where *#Pass* is the number of passing tests, *#Fail* is the number of failing tests, *fl* is the name of the fault localization technique, and *TextFile1, TextFile2, ..., TextFilen* are the statement ID list files generated by executing each test case on the instrumented code.

```
# 4.Run fault localization code
../faultLoc --pass {#Pass} --fail {#Fail} --fl {FLTechnique} TextFile1 TextFile2
                TextFile3 ... TextFilen > LPFS
```

FIGURE 4.4. Linux shell script to run fault localization technique.

To repair faults, a user runs the repair tool through a command-line as shown in Figure 4.5, where *#Gen* is the number of generations, *#Pop* is the population size in each generation, *Max* is the maximum fitness value, and *FaultyProgram* is the faulty program name.

```
# 5.Run the repair code
../modify --gen {#Gen} --pop {#Pop} --max {Max} {FaultyProgram}.c
```

Figure 4.5. Linux shell script to run repair code

The repair tool will produce a text file containing summary information for the repair algorithm execution, and the new version of the faulty program if a potential repair is found.

4.3. MUT-APR Framework limitations

MUT-APR targets simple binary operator faults; however, there are number of limitations related to the framework implementation.

Because GenProg and MUT-APR depend on the CIL framework, MUT-APR transforms logical operators into if-then and if-else blocks. Thus, MUT-APR cannot fix some type of faults such as logical operators and the equality operator. To repair logical operators, we will need to either mutate the if blocks generated by CIL, or redesign MUT-APR so that it does not depend on CIL.

Our program modification operators change arithmetic operators in *return* statements, *assignments*, *if* bodies, and *loop* bodies, but it cannot access arithmetic operators in *if* statements (e.g, if(x + 1 > 0)). This is also a limitation due to the use of CIL. Additionally, MUT-APR targets fault that required one line modification; therefore, the current implementation does not fix faults that required multiple line changes.

4.4. Summary

In this Chapter we explained the implementation of MUT-APR framework components and mechanisms. We also explained the steps to repair faults using MUT-APR. Lastly, we explained the limitations of the current implementation. The next chapter presents the results of our evaluations.

CHAPTER 5

Evaluations

In this chapter, we present our evaluation studies. Section 5.1 describes the benchmarks and faults that are used in our evaluations. Section 5.2 describes the study of the effects of different program modification operators (PMOs) on APR effectiveness and repair quality [80]. We studied the impact of two types of PMOs: MUT-APR program modification operators and GenProg program modification operators. Section 5.3 presents the study of the effect of different repair test suite properties on APR effectiveness, repair quality, and performance [80]. Repair test suite properties include: (1) the test selection methods, such as branch coverage, statement coverage, and random testing, and (2) the size of repair test suites. Section 5.4 presents the results of applying different fault localization techniques on APR effectiveness and performance. The fault localization techniques that we studied include the following: Tarantula, Ochiai, Jaccard, Optimal, and GenProg's Weighting Scheme [7]. Section 5.5 provides the results of applying stochastic and exhaustive search algorithms [81]. The stochastic search algorithms studied include the following: a genetic algorithm, a genetic algorithm without a crossover operator, a basic random search, a guided genetic algorithm, a guided genetic algorithm without a crossover operator, and a guided basic random search. The exhaustive search algorithms studied include a brute-force and an ordered brute-force that applies heuristics to order PMOs.

5.1. Benchmarks and Faults

We used six C programs from the Software artifacts Infrastructure Repository (SIR) [50] along with a comprehensive set of test inputs. We used the Siemens Suites: tcas, replace, schedule2, and tot_info. We also used two larger programs: space and sed. Even though

Program	LOC	# Regression Tests	Description
replace	564	5542	Pattern matching and substitution
tcas	173	1608	Aircraft collision avoidance system
schedule2	374	2650	Priority schedulers
tot_info	565	1052	Computes statistics
space	6195	5670	Array Definition Language Interpreter
sed	14427	370	Stream editor
Total	22201	16952	

Table 5.1. Benchmark programs. Each *Program* is an original program from the SIR [50]. *LOC* is the number of lines of codes. *#Regression Tests* is the number of regression tests.

many benchmarks are available in the SIR, we excluded faulty versions which have no tests to execute faults. We also excluded other faulty versions with faults other than operator faults (print-tokens, print-tokens2, and schedule) since the ability to fix faults depends on the set of PMOs that are supported by an APR. For example the seeded fault in print-tokens program (in the SIR) can be fixed by inserting or deleting a statement and it cannot be fixed by our PMOs. We also excluded grep and gzip because they could not be compiled within our framework.

Each subject program was seeded with a single fault. We used multiple faulty versions for each subject program. Faulty versions are taken from the SIR. We also used the Proteum/IM 2.0 [82] C mutation tool to create additional faulty versions. The faults in the subject programs are seeded in statements of different types: *if* statements, *return* statements, *loops*, and *assignments*. We also include subject programs that contain a fix such as replace-v6. Table 5.1 identifies the subject programs along with their size, the number of regression tests, and the program description.

5.2. Program Modification Operators

Program modification operators impact APR effectiveness and repair quality. We designed this evaluation to study the impact of applying simple PMOs versus the use of existing code as done by GenProg to fix faulty operators with APR techniques.

This is a necessary "smoke test" as MUT-APR is specially designed to fix faulty operators and only faulty operators while GenProg is not.

5.2.1. Research Questions.

The evaluation study is designed to answer the following research questions:

- RQ1 APR Effectiveness : Does applying simple PMOs improve APR effectiveness in fixing faulty operators compared to the use of GenProg PMOs?
- RQ2 APR Repair Correctness: Does applying simple PMOs to fix faulty operators improve repair correctness compared to the use of GenProg PMOs?
- RQ3 APR Repair Maintainability: Does applying simple PMOs to fix faulty operators improve repair maintainability compared to the use of GenProg PMOs?

5.2.2. Evaluation Design.

We first compared APR effectiveness of MUT-APR PMOs to those of GenProg, which use existing code to repair faults. We selected faulty versions with operator faults; we included four programs from the Siemens suites from Table 5.1 (tcas, replace, schedule2, and tot_info) with 54 faults after removing equivalent mutants. We used small repair test suites (containing 5-30 test cases) that satisfy the branch coverage test criterion since branch coverage repair tests generate more validated repairs [80], and each repair test suite has at least one passing and one failing tests.

To compare the repair correctness produced by the two PMOs, we only include subject programs that can be repaired by both approaches studied (only 17 of them can be fixed by both techniques). We include 648 potential repairs that are generated by MUT-APR PMOs, and 475 potential repairs that are generated by GenProg PMOs.

For a valid comparison, we applied the same search algorithm (genetic algorithm) to search the space of possible modifications for a PMO. MUT-APR takes the same set of parameters as GenProg. We used the default parameters from Weimer et al. [7]. Both MUT-APR and GenProg use random techniques to search the space of possible modifications, we ran each tool 100 times to ensure that it will generate a potential repair in at least one execution for each subject program. In each execution, called a trial, the genetic loop runs for 10 generations, and each generation consists of a population size of 40 versions. The weights used for the passing and the failing tests are 1 and 10 respectively.

We used GenProg version.2 to study the ability of GenProg to fix the faulty operators since Le Goues et al. results showed that GenProg version.2 fixed more faults, and we set the parameters in GenProg.v2 using the values specified in Le Goues et al. [42, 43].

5.2.3. Measurements. To assess APR effectiveness we measured the total percent of fixed faults, and the success rate. The success rate is the percentage of trials resulting in a potential repair. Repair correctness is measured by the percent of failed repairs (PFR) and average percent of failed tests (APFT): PFR is the percentage of potential repairs produced by APR for 100 trials that fail when tested on regression tests, and APFT is the average percent of failing regression tests for N potential repairs for each faulty version; N is the number of generated potential repairs for 100 trials. To measure repair maintainability, we compute the size of potential repairs. The number of lines of code changed (LOCC) is the number of LOC modified, deleted, or added until a potential repair is found. We also analyze potential repairs to check the distribution of modifications. Automatically generated code that is scattered may reduce software maintainability.

Program	tcas	replace	schedule2	tot_info	Percent of fixed faults
Total # of faults	21	14	8	11	
MUT-APR	18	14	6	9	87.03%
GenProg.v2	2	10	5	0	31.48%

Table 5.2. MUT-APR vs. GenProg: number/percent of operator faults fixed.

5.2.4. Results. We analyzed our results to answer the three research questions. To compare the effectiveness and repair quality of two APR techniques, we computed the percent of fixed faults, the success rate, PFR, and APFT across all faulty versions, and then we compared the averages of each APR technique. We compared the distributions of the success rate, PFR and APFR using the raw data and a transformed data; we used *sqrt* function to transfer data to make it more normal. Since our data is not normally distributed, we applied the non-parametric test, Wilcoxon Signed-Rank Test [83], to analyze the statistical difference at the 0.95 confidence level.

5.2.4.1. *RQ1: APR Effectiveness.* This experiment shows the feasibility of using simple PMOs to fix faulty operators. As expected, MUT-APR successfully repaired faulty operators in 47 out of 54 (87.03%) of the programs. GenProg fixes many faults that MUT-APR cannot fix such as missing statements and segmentation faults [7, 12]. However, GenProg repaired only 17 (31.48%) of the faulty operators. GenProg can only work if the original source code contains a fix for the fault. GenProg can fix faults in tcas-v1 and replace-v6 because these programs contain the correct code somewhere. However, GenProg fixed faults in only one version (v0) of replace-v6; it failed to fix faults in all other versions even though the original source code contains a fix. Table 5.2 shows the number of potential repairs found for each benchmark by running each faulty program on both MUT-APR and GenProg. MUT-APR was able to repair more operator faults than GenProg for each subject program.

We studied why both MUT-APR and GenProg did not fix faults in some faulty versions of replace.c, schedule2.c, and tot_info.c. We suspected that the reason might be the randomness of MUT-APR; we executed the faulty versions using an exhaustive search method (brute-force). We found these faults were not fixed by brute-force. We looked at the LPFS and found that the faults were in statements that did not appear in the LPFS, and thus, were not treated as potentially faulty. A better fault localization technique would improve the effectiveness of our approach.

For faulty versions when both MUT-APR and GenProg fixed faults, we compared the success rate. Figure 5.1 summarizes the success rate for each APR technique. For eight out of 17 faulty versions, MUT-APR had a higher success rate than GenProg. MUT-APR had a higher average success rate (38.17%) compared to Genprog (33.52%). We applied Wilcoxon Signed-Rank Test to measure the statistical difference between success rate of GenProg and MUT-APR; the difference is not significant (p-value = 0.261) at the 0.95 confidence level, but the success rate is significant at the 0.75 confidence level. This indicates a strong evidence that MUT-APR improved success rate when fixing faulty operators compared to GenProg. Figure 5.2 shows success rate distribution for each APR technique. Figure 5.2a shows the distributions of the raw success rate data, and Figure 5.2b shows the distribution of the transformed success rate using the *sqrt* function.

5.2.4.2. *RQ2: APR Repair Correctness.* To determine which technique produces more validated repairs, we executed the generated potential repairs on a set of regression tests, and computed the PFR and APFT as described in Figure 1.4. This evaluation only included the faulty versions that could be repaired by both PMOs. As shown in Figure 5.3, the PFR for potential repairs that are generated by MUT-APR is less than the PFR of potential repairs that are generated by GenProg. For ten out of 17 programs, all GenProg potential repairs

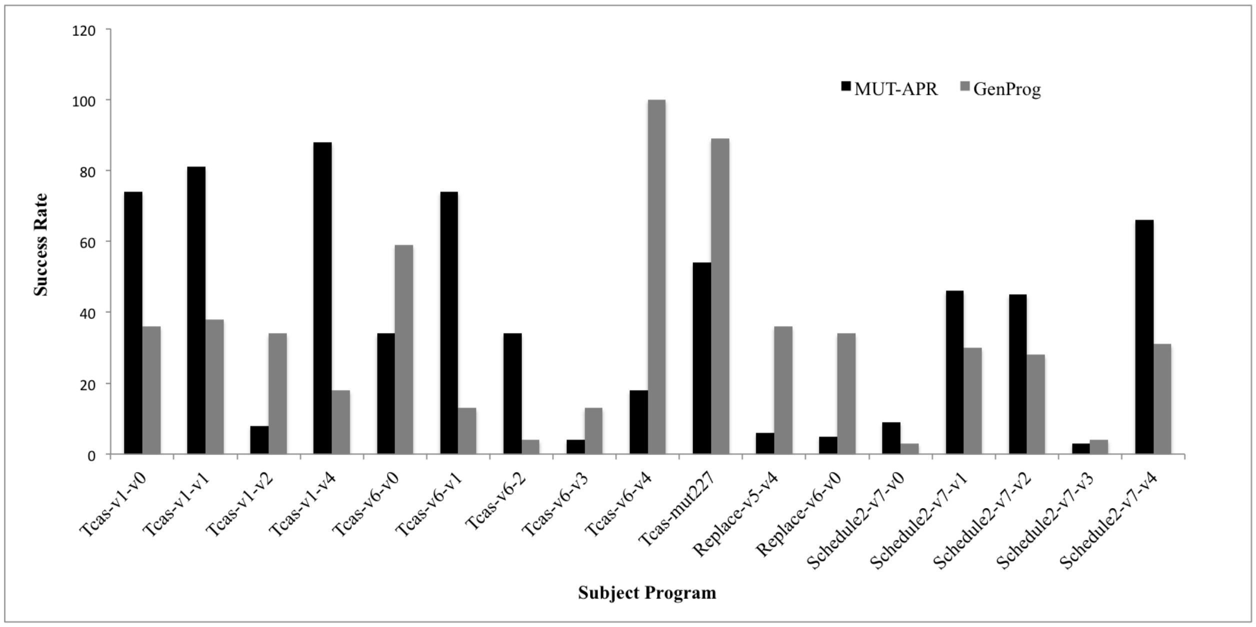
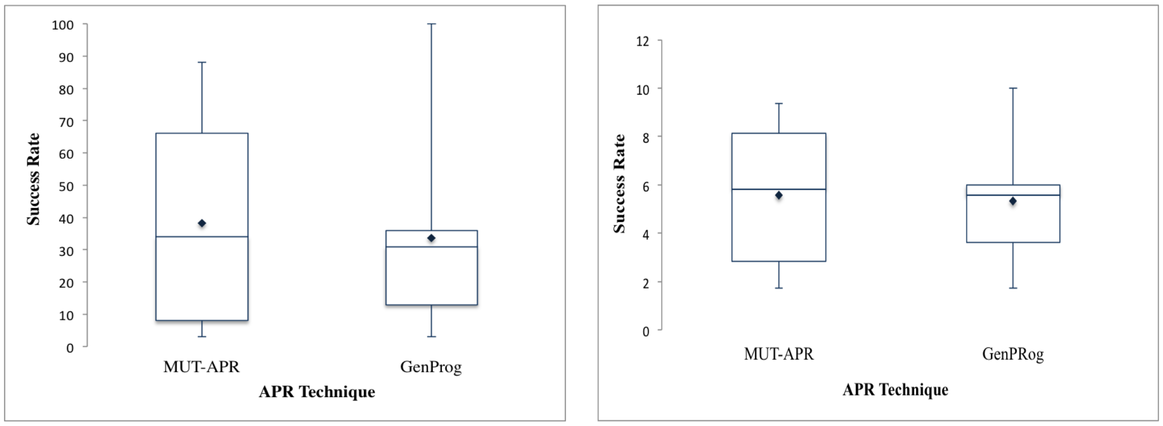


FIGURE 5.1. Success rate by mutation-based technique (MUT-APR) and the use of existing code (GenProg). Higher success rate is better.



(A) Raw Success Rate Data (B) Transformed Success Rate Data

FIGURE 5.2. MUT-APR effectiveness using MUT-APR and GenProg to repair faults. Higher success rate is better.

failed regression tests. We found that, for all the benchmarks, 27.36% of potential repairs that are generated by the MUT-APR technique failed regression tests (PFR=27.36%), while 96.94% of repairs that are generated by GenProg failed regression tests (PFR=96.94%). For six faulty versions, MUT-APR generates potential repairs that did not fail any regression tests (with zero PFR). Figure 5.4 shows the distribution of the percent of failing repair

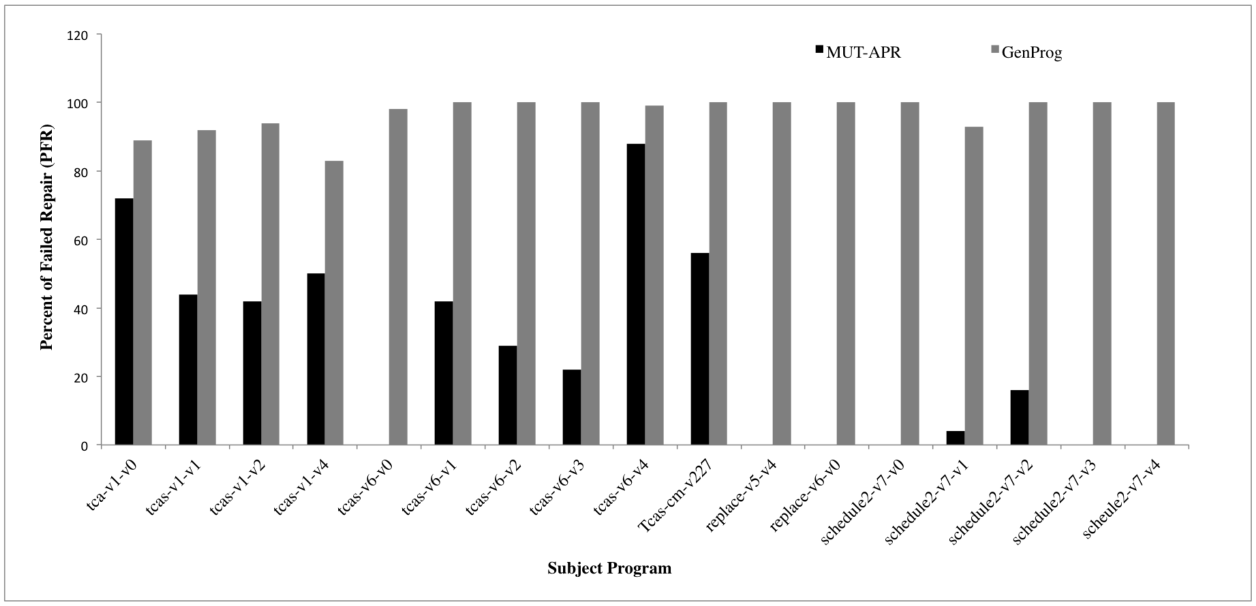
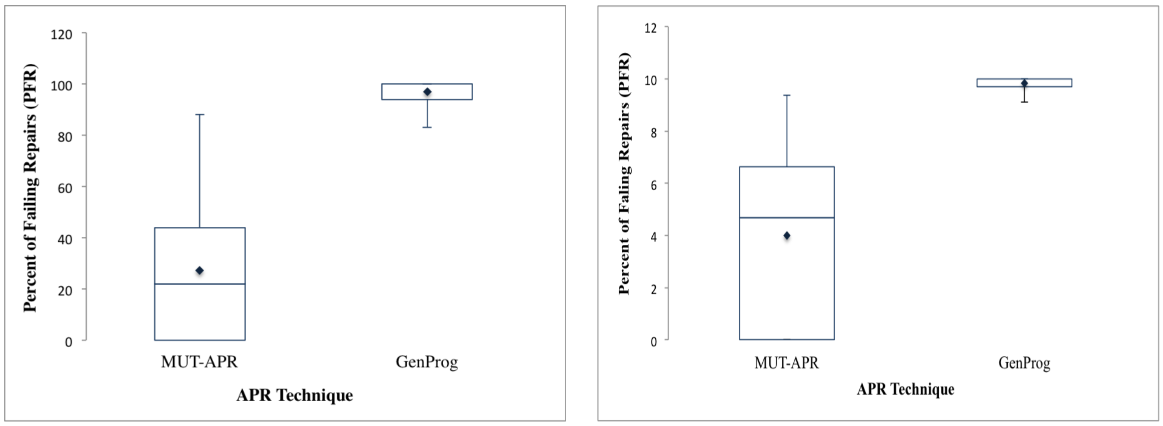


Figure 5.3. Percent of Failed Potential Repairs (PFR) for 100 trials for each faulty version using MUT-APR and GenProg. Lower PFR is better.

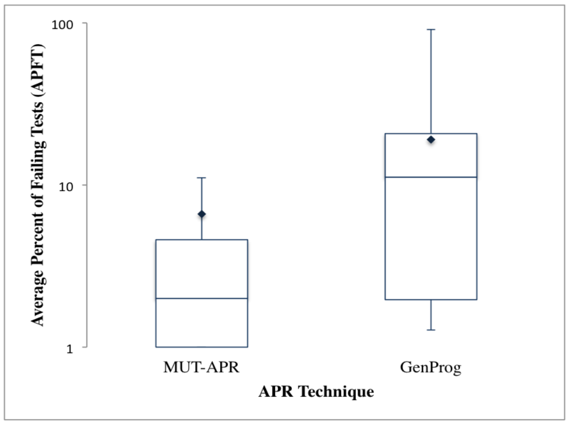


(a) Raw PFR Data

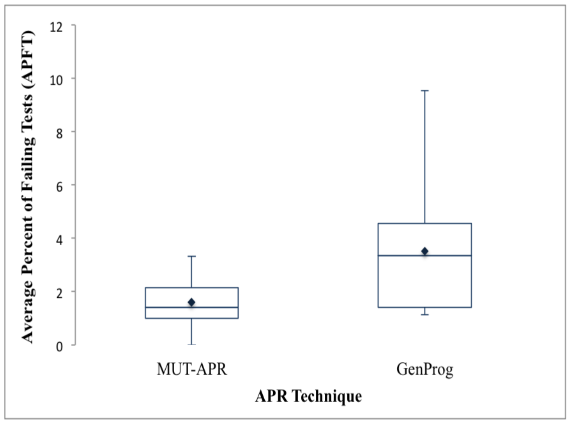
(b) Transformed PFR Data

Figure 5.4. Percent of Failing Potential Repairs (PFR) for MUT-APR and GenProg. PFR is the percent of potential repairs failed for 100 trials. Lower PFR is better.

(PFR) for each APR technique: Figure 5.4a shows the distributions of the raw data, and Figure 5.4b shows the distributions of the transformed data using *sqrt* function. We studied the difference between MUT-APR and GenProg using Wilcoxon Signed-Rank Test. The PFR difference is statistically significant ($p\text{-}value = 0.00$) for all benchmarks at the 0.95 confidence level.



(A) Raw APFT Data



(B) Transformed APFT Data

FIGURE 5.5. Average percent of failing regression tests (APFT) using MUT-APR and GenProg to repair faults. Lower APFR is better.

We also computed the average percent of failed regression tests for N potential repairs for each each faulty version (APFT); *N* is the number of potential repairs generated for 100 trials for each faulty version. We only included faulty versions with at least one failing potential repair; in other words, we removed faulty versions with PFR = 0. Potential repairs that are generated by MUT-APR failed in an average of 6.65% of regression tests (APFT = 6.65%), while potential repairs that are generated by GenProg failed in an average of 19.05% of the tests (APFT = 19.05%). Figure 5.5 shows the APFT distributions for each APR technique. Figures 5.5 and 5.5b show the distributions of the raw APFT data and a transformed form of APFT data using the sqrt function, respectively. We analyzed the difference in APFT using Wilcoxon Signed-Rank Test; the difference is statistically significant (p-value = 0.004) at the 0.95 confidence level.

5.2.4.3. *RQ3: APR Repair Maintainability.* To measure repair maintainability we compare the modifications made by the mutation-based repair technique (MUT-APR) to those using existing code (GenProg). Fewer and smaller modifications will make the software easier to understand and maintain, and will also improve repair correctness [66].

Since MUT-APR only modifies an operator, a MUT-APR potential repair minimizes changes to the code making it similar to repairs done by humans, and thus should not reduce software maintainability. In contrast, GenProg's use of existing code to repair faults produces many irrelevant changes to the code which reduce maintainability [42]. When GenProg generates a potential repair, it also makes many extraneous changes to the code that obfuscate the change that fixed the fault. In addition, GenProg actually does not change the faulty operator to correct an operator fault. For example, the *gcd* program in Figure 3.1 was repaired by both MUT-APR and GenProg. MUT-APR changed one statement as shown in Figure 5.6. On the other hand, GenProg fixed the faults by making three changes as shown in Figure 1.3 in Chapter 1.

```
if (a == (double )0)
```

FIGURE 5.6. MUT-APR potential repair for the fault in Figure 3.1

We compared line of code changed (LOCC) when both MUT-APR and GenProg generated potential repairs. We used the Linux *diff* command to count the LOCC, and we checked the distributions of changes. If changes are scattered in the code, the code will be harder to understand and read. We compared the differences between the original code, and potential repairs generated by MUT-APR and by GenProg. MUT-APR potential repairs change an average of l.72 LOC in all subject programs, while the GenProg potential repairs change an average of 28.68 LOC. In addition, GenProg changes code in many locations, which will tend to make the code less readable and maintainable.

5.2.5. CONCLUSIONS AND LESSONS LEARNED. We compared APR effectiveness and repair quality when using two APR techniques: the use of existing code (GenProg) versus use of simple PMOs (MUT-APR). MUT-APR is more effective than GenProg in fixing operators faults (87.03% vs. 31.48%), and MUT-APR had a higher average success rate (38.71%) compared to GenProg (33.52%). The results by Debroy and Wong howed that simple PMOs fixed 39.14% of faulty versions [2, 3]; however, their study included non-operator faults.

We found that 27.36% of potential repairs that are generated by the mutation-based repair technique failed regression tests, while 96.94% potential repairs that are generated by using existing code failed regression tests. Potential repairs of operator faults produced by mutations (MUT-APR) tend to be validated repairs more often than those produced by the use of existing code (GenProg). These results are expected due to the different PMOs used by MUT-APR and GenProg. GenProg PMOs make many changes to the code, unlike MUT-APR, which uses PMOs that insert one change into the code and thus reduces the risk of introducing new faults.

Unlike the use of existing code, using simple PMOs to fix faults changes only a single operator in each run. Therefore, program repair is similar to repairs done by humans, and should not reduce maintainability.

This evaluation demonstrates that an APR tool, like MUT-APR, that is designed to repair a specific kind of fault, like single operator faults, can be more effective in repairing programs with these faults compared to an APR tool like GenProg that does not target specific fault types. Focusing on fault types can improve both APR effectiveness and the quality of generated repairs.

5.3. Repair Test Suites

Repair test suite properties can impact APR effectiveness, repair quality, and performance. This evaluation studies the impact of the method used to select repair test suits (Section 5.3.1) and repair test suite size (Section 5.3.2) on MUT-APR.

5.3.1. Repair Test Suite Selection Method. In this section, we studied the effect of repair test suite selection method. We compared the results from using three test selection methods: branch coverage, statement coverage, and random testing.

5.3.1.1. *Research Questions.* Our evaluation study is designed to answer the following research questions:

- RQ1 APR Effectiveness: Does the selection of repair test suite method (branch coverage, statement coverage, and random testing) affect APR effectiveness?
- RQ2 APR Repair correctness: Does the selection of repair test suite method (branch coverage, statement coverage, and random testing) affect repair correctness?
- RQ3 APR Performance: Does the selection of repair test suite method (branch coverage, statement coverage, and random testing) affect APR performance?

5.3.1.2. *Evaluation Design.* This evaluation requires a test suite per test selection method for each faulty version, and each test suite has at least one passing and one failing tests. Branch coverage and random test suites are from the SIR repository [50]. *Gcov* [84] is used to create statement coverage test suites. We consider a test suite to have achieved a specific coverage criteria (either branch or statement coverage) if it covers at least 90% of the test requirements. We used small repair test suites; each suite contains 5-30 test cases.

To study the effect of three test methods on the repair process, we selected the benchmark programs from Table 5.1 that include at least one test suite of each test selection method that has at least one passing and one failing tests. We included a total of 26 faulty versions, and we ran MUT-APR on the selected programs. Table 5.3 gives the number of faulty versions of each subject program, and the average size of repair test suites of each type. This evaluation includes 765 potential repairs that are generated using branch coverage test

		Average Repair Test Suite Size		
Program	# Faulty Ver.	Branch Coverage	Statement Coverage	Random Tests
replace	10	17.5	13.1	19.9
tcas	3	6	5	12
schedule2	1	7	5	15
tot_info	12	8.3	8	10.6

TABLE 5.3. Benchmark programs to study the impact of repair test suite selection methods. Each *Program* is an original program from the Siemens suite [50]. *# Faulty Ver.* is the number of faulty versions. *Average Repair Test Suite Size* is the average number of repair tests per test type.

suites, 1,234 potential repairs that are generated using statement coverage test suites, and 986 potential repairs that are generated using random tests (a total of 2,985 repairs).

5.3.1.3. *Measurements.* Effectiveness is measured by the success rate; success rate is the percentage of trials that resulted in a potential repair. Repair quality is measured by the percent of failing potential repairs (PFR) for 100 trials for each faulty version, and the average percent of failing regression tests for N potential repairs for each faulty version (APFT); N is the number of potential repairs generated for 100 trials for each faulty version. Performance is measured by the average number of generated variants (ANGV) for N potential repairs for each faulty version, and the average total time (ATotalTime) for N potential repairs for each faulty version.

5.3.1.4. *Results.* We analyzed our results to answer the three research questions. To compare the impact of different repair test methods, we compared the success rate, PFR, APFT, ANGV, and ATotalTime when each test method is used to repair fault. We compared the distributions of the measurments using the raw data and a transformed data; we used the *sqrt* function to make the data more normal. We applied Wilcoxon Signed-Rank Test to analyze the statistical difference at the 0.95 confidence level.

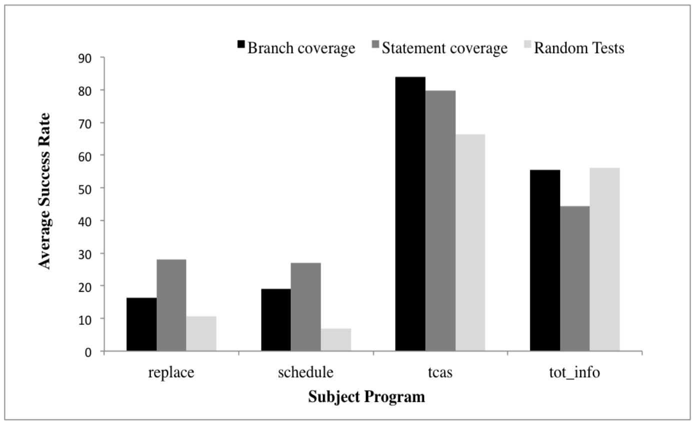
5.3.1.4.1. *RQ1: APR Effectiveness*. We compared the effectiveness of MUT-APR when three different repair test suite methods are used to generate repairs: repair test suites

Measurements		Branch Coverage		Statement Coverage		Random Tests	
		Raw	Trans.	Raw	Trans.	Raw	Trans.
Success rate	Mean	42.30	6.10	41.46	6.09	37.92	5.45
	Median	39.30	6.28	32.50	5.70	20.00	4.47
	Std.Dev.	29.60	2.29	27.70	2.14	35.20	2.91
PFR	Mean	12.77	2.29	26.58	3.46	24.04	3.87
	Median	1.000	0.71	0	0	11.50	3.38
	Std. Dev.	18.30	2.79	31.90	3.89	28.40	3.07
APFT	Mean	3.200	1.580	4.900	1.990	4.800	1.980
ANGV	Mean	187.34	13.63	186.71	13.63	176.50	12.79
	Median	182.20	13.49	186.10	13.64	187.90	13.70
	Std. Dev.	34.600	1.270	26.900	0.970	78.300	3.630
ATotalTime	Mean	14.25	3.65	11.87	3.35	8.23	2.65
	Median	12.30	3.51	11.00	3.33	6.4	2.52
	Std. Dev.	7.200	0.98	5.300	0.80	7.5	1.12

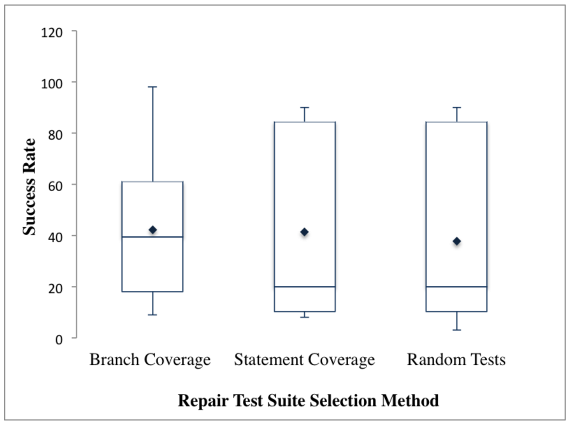
TABLE 5.4. MUT-APR effectiveness, repair quality, and performance of using different test methods to select repair tests. *Raw* column is the raw data and *Trans* column is the transformed data using *sqrt* function.

that (1) satisfy branch coverage, (2) satisfy statement coverage, and (3) repair tests that are randomly selected. Table 5.4 shows the average and median success rate for each test method. Random tests decreased the average success rate (37.92%) compared to branch coverage (42.30%) and statement coverage (41.46%). Branch coverage repair test suites were the most effective of the three methods.

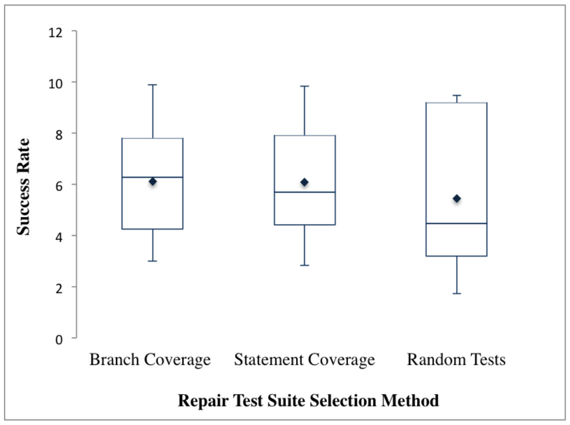
Figure 5.7a graphically shows the average success rate per subject program when different repair test suite methods used. Statement coverage repair test suites had the highest average success rate for replace and schedule2 programs, while branch coverage repair test suites had the highest average success rate for tcas, and random test repair test suites had the highest average success rate only for the tot_info program. Then, we computed the success rate when each test method was used across all faulty versions. Branch coverage repair test suites had the highest success rate for 38% of faulty versions compared to statement coverage and random repair test suites which had the highest success rate for 34% and 15% of faulty versions, respectively. Figures 5.7b and 5.7c show the success rate distributions for each test



(A) Average Success Rate Per Program



(B) Raw Success Rate Data



(C) Transformed Success Rate Data

Figure 5.7. MUT-APR effectiveness using different test methods to select repair tests. Higher success rate is better.

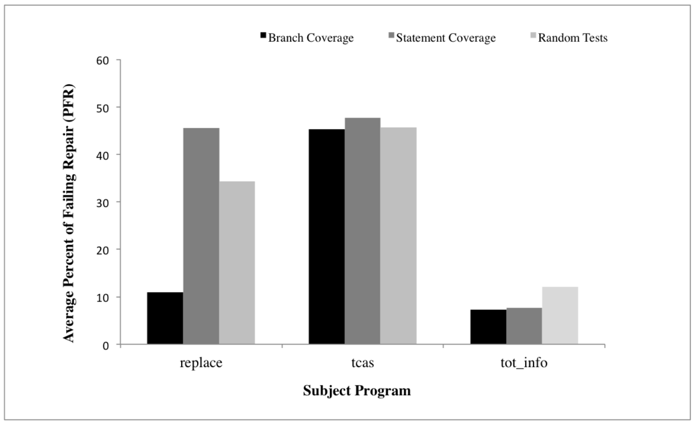
method. The difference in success rate is not significant between the three test methods at the 0.95 confidence level.

5.3.1.4.2. *RQ2: APR Repair Correctness*. To study repair correctness for generated repairs, we follow the steps described by Figure 1.4. We executed the regression tests on all potential repairs that are generated using branch coverage (*Repair Test1*), statement coverage (*Repair Test2*), and random tests (*Repair Test3*). Then we computed the percentage of failed potential repairs (PFR) for 100 trials for each faulty version, and the average percent of failed regression tests (APFT) for N potential repairs for each faulty version. To measure APFT, we excluded faulty versions that had no failing potential repairs (PFR = 0).

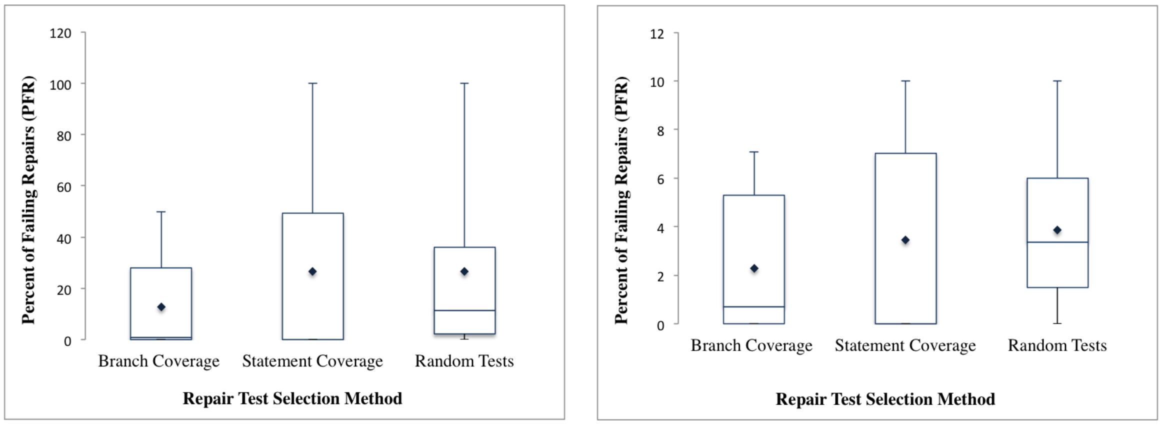
Table 5.4 summarizes the correctness of repairs that were generated using different test suite methods. We found that an average of 12.77% of potential repairs that are generated using branch coverage test suites failed regression tests, while 26.58% and 24.04% of potential repairs that are generated using statement coverage and random test suites failed regression tests, respectively. Figure 5.8a shows the PFR per program; we removed schedule2 from the chart because it has PFR = 0 by all test methods. Branch coverage produced fewer failing potential repairs for all faulty versions compared to statement coverage and random tests. Random tests produced fewer failing potential repair than statement coverage for tcas and replace programs. All repair test suite methods had the same impact on repairs to the schedule2 programs; all generated potential repairs passed all regression tests (PFR = 0).

Then, we compared the PFR for each faulty version. For 38% of repaired faults, branch coverage repair test suites generated repairs with lower PFR than statement coverage repair test suites, and for 57% of repaired faults, branch coverage repair test suites generated potential repairs with lower PFR than random repair tests. For 50% of repaired faults, branch coverage repair test suites generated potential repairs that did not fail any regression test (with zero PFR). The largest PFR value using branch coverage repair test suites to produce potential repairs is 50%, while statement coverage and random repair test suites generated potential repairs that failed all regression tests (with PFR equal to 100). Figures 5.8b and 5.8c show PFR distributions when each test method is used to find potential repairs.

Potential repairs that are generated using branch coverage failed in an average of 3.2% (APFT= 3.2%) of regression tests, while potential repairs generated using statement coverage failed in an average of 4.9% (APFT= 4.9%) of regression tests, and potential repairs generated using random test suites failed in an average of 4.8% (APFT= 4.8%) of regression tests. We analyzed the improvements of repair correctness when different repair test suite



(A) Average PFR Per Program



(B) Raw PFR Data (C) Transformed PFR Data

Figure 5.8. MUT-APR repair correctness using different test methods to select repair tests. Lower PFR is better.

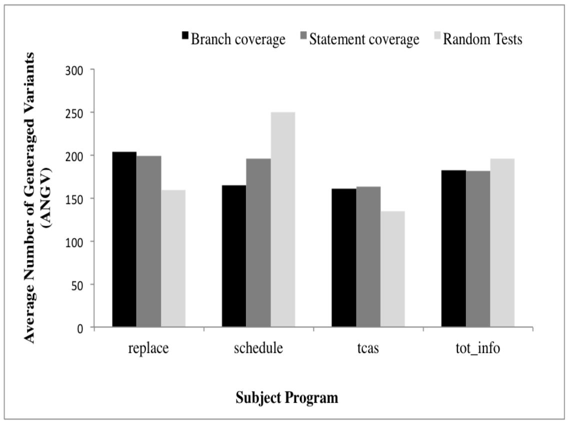
methods were used by applying Wilcoxon Signed-Rank Test. The difference is statistically significant between branch coverage and statement coverage (p-value = 0.02), and between branch coverage and random tests (p-value = 0.03) at the 0.95 confidence level. However, the difference is not statistically significant between statement coverage and random tests (p-value = 0.436).

5.3.1.4.3. *RQ3: APR Performance.* We compared the ANGV and ATotalTime for N repairs for each faulty version; N is the number of potential repairs for 100 trials for each faulty version. We excluded faulty versions that have no repairs (success rate = 0). Random repair tests generated an average of 176.5 variants until a potential repair is found, while branch coverage and statement coverage repair tests generated an average of 187.34 and 186.71 variants, respectively. We also compared the ANGV required to repair faults for each subject program (Figure 5.9a). For the tcas and replace programs, branch coverage and statement coverage generated more variants until a potential repair is found compared to random tests. Random tests generated more variants compared to the other repair test suite methods for the schedule2 and tot_info programs. Figures 5.9c and 5.9d show the ANGV distributions across all faulty versions using the raw data, and the transformed data. We analyzed the results using Wilcoxon Signed-Rank Test. The difference in ANGV for the three test methods is not significant at the 0.95 confidence level (Table 5.5).

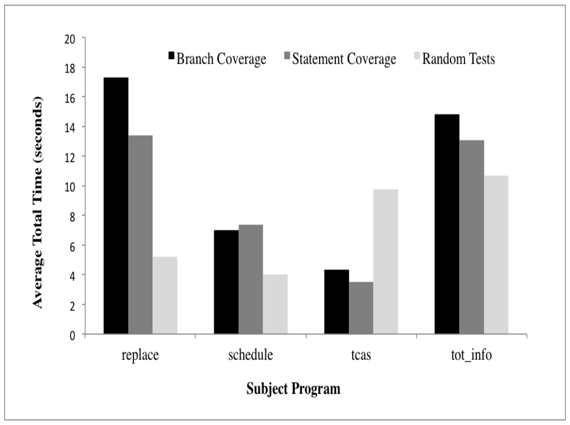
Then, we compared the ATotalTime required to find potential repairs when each test method was used. Random tests reduced the average total time to find a potential repair; it took an average of 8.23 seconds to find potential repairs using random tests compared to branch coverage and statement coverage which took an average of 14.25 and 11.87 seconds to find potential repairs. The average total time required to fix faults per subject program when random tests were used to repair faults is reduced for all programs except tcas program (Figure 5.9b). Branch coverage increased the average total time to find potential repairs compared to statement coverage and random testing for replace and tot_info, which represent %85 of the faulty versions used in this evaluation. Figures 5.9e and 5.9f show the ATotalTime distributions across all faulty versions using the raw data, and the transformed data. The difference in ATotalTime between branch coverage, statement coverage, and random tests is significant at the 0.95 confidence level (Table 5.5).

5.3.2. Repair Test Suite Size.

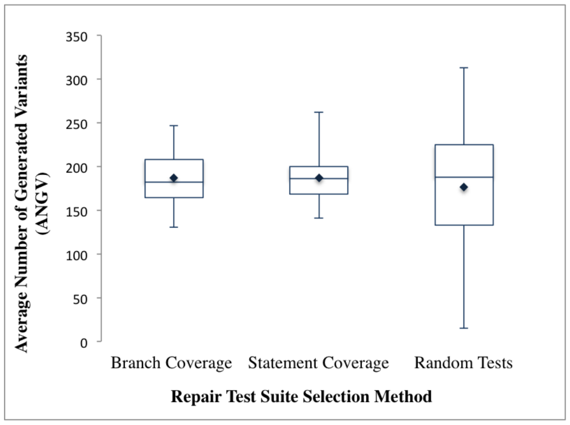
In this section, we compared the results of using small and large repair test suites. Since results from Section 5.3.1 shows that repair test suites that satisfy branch coverage produced more validated repairs, we used small and large branch



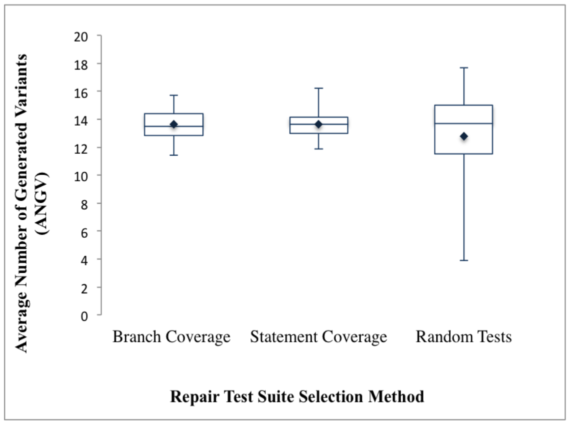
(A) ANGV Per Program



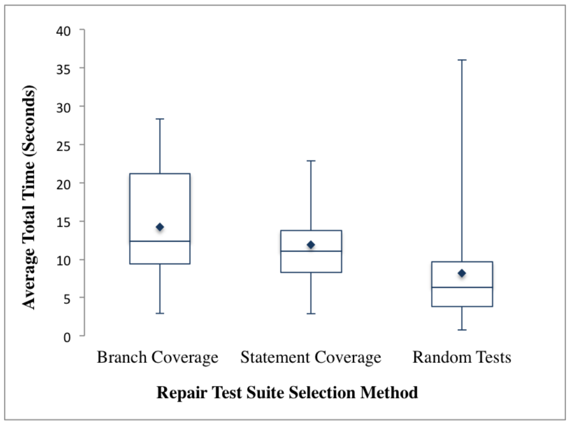
(B) ATotalTime per program



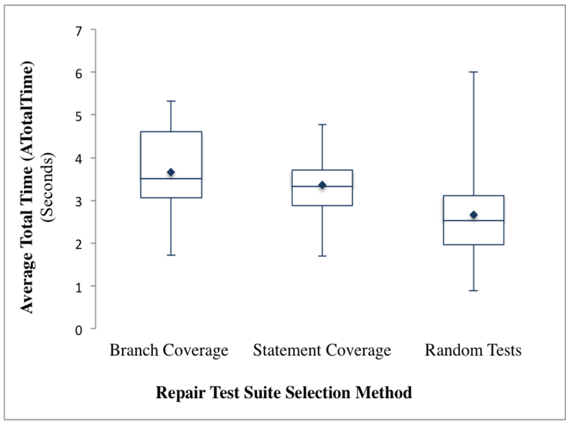
(C) Raw ANGV Data



(D) Transformed ANGV Data



(E) Raw ATotalTime Data



(F) Transformed ATotalTime Data

Figure 5.9. MUT-APR performance using different test methods to select repair tests. Lower ANGV and ATotalTime is better.

Selection method	ANGV P-values	ATotalTime P-values
Branch Coverage - Statement Coverage	0.36	0.001
Branch Coverage - Random Testing	0.41	0.020
Statement Coverage - Random Testing	0.48	0.002

Table 5.5. P-values for ANGV and ATotalTime between test methods.

coverage repair test suites to study the impact of repair test suites size on APR effectiveness, repair quality, and performance.

5.3.2.1. *Research Questions.* Our evaluation study is designed to answer the following research questions:

- RQ4 APR Effectiveness: Does the selection of repair test suite size (small and large) improve APR effectiveness?
- RQ5 APR Repair correctness: Does the selection of repair test suite size (small and large) improve repair correctness?
- RQ6 APR Performance: Does the selection of repair test suite size (small and large) improve APR performance?

5.3.2.2. *Evaluation Design.* We used the small repair test suites that were used in the previous evaluation (Section 5.3.1), and the large repair test suites are taken from the SIR repository [50]. Each have at least one passing and one failing tests. We measure the coverage for the large test suite; it reached 90% or more of the branches of the programs under test. Each test suite contains 80-400 test cases, and have at least one passing and one failing tests. We used the benchmark programs from Table 5.1 that come with small and large test suites that satisfy branch coverage.

We included 46 faulty versions in total. Table 5.6 summarizes the number of faulty versions for each program and the average size of small and large test suites that were used in this evaluation. To repair faults, we ran MUT-APR on the selected programs.

Program	# Faulty Ver.	Average Repair Test Suite Size Small test suite	Average Repair Test Suite Size Large test suite
replace	17	18.1	389.2
tcas	12	5.7	82.1
schedule2	5	7	226.8
tot_info	12	8.3	193
Total	46	39.2	893.0

Table 5.6. Benchmark programs to study the impact of repair test suites size. Each *Program* is an original program from the Siemens suite [50]. *# Faulty Ver.* is the number of faulty versions. *#Repair Test Suite Size* is the average number of repair tests per test size.

5.3.2.3. *Measurements.* Effectiveness is measured by the success rate. Success rate is the percentage of trials that resulted in a potential repair. Repair quality is measured by the percent of failing potential repairs (PFR) for 100 trials for each faulty version, and the average percent of failing regression tests for N potential repairs for a faulty version (APFT); N is the number of potential repairs generated for 100 trials for each faulty version. Performance is measured by the average number of generated variants (ANGV) for N potential repairs for each faulty version, and the average total time (ATotalTime) for N potential repairs for each faulty version.

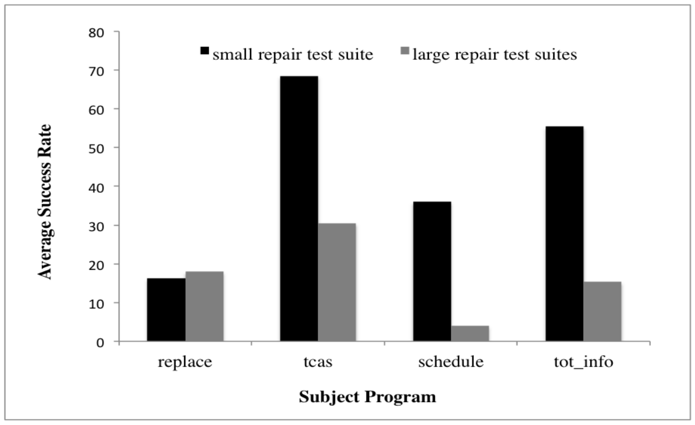
5.3.2.4. *Results.* We analyzed our results to answer the three research questions. To compare the impact of different repair test suite size, we compared the success rate, PFR, APFT, ANGV, and ATotalTime when each test suite is used to repair a fault. We compared the data distributions of the raw data and the transformed data using the *sqrt* function to make the data more normal. We applied the non-parametric Wilcoxon Signed-Rank Test to analyze the statistical difference when the data distribution is not normal, and the Paired T-Test when the data distribution is normal. We studied the statistical difference at the 0.95 confidence level.

Measurements		Small repair test suites		Large repair test suites	
		Raw	Trans.	Raw	Trans.
Success rate	Mean	42.2	6.06	19.0	3.48
	Median	42.0	6.48	12.0	3.46
	Std.Dev.	29.0	2.37	19.8	2.66
PFR	Mean	31.3	3.88	89.1	8.98
	Median	6.60	2.55	100	10.0
	Std. Dev.	38.7	3.99	30.3	2.95
APFT	Mean	3.5	1.48	13.7	2.34
	Median	3.0	1.73	0.10	0.32
	Std. Dev.	3.2	1.90	24.7	2.92
ANGV	Mean	186.8	13.61	203.0	14.18
	Median	185.8	13.63	202.6	14.23
	Std. Dev.	32.90	1.290	40.30	1.410
ATotalTime	Mean	13.4	3.48	39.2	5.68
	Median	12.3	3.51	35.6	5.97
	Std. Dev.	8.30	1.15	34.7	2.66

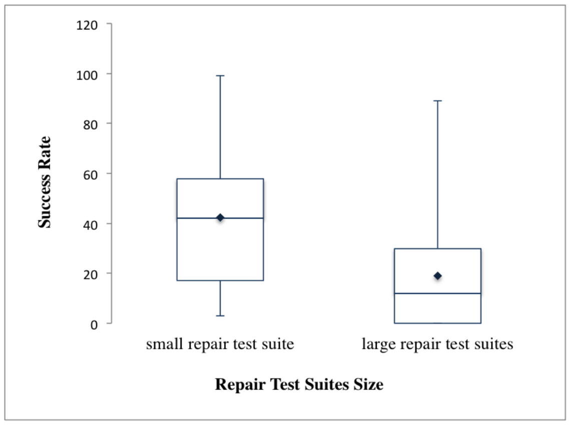
TABLE 5.7. MUT-APR effectiveness, repair correctness, and performance using small and large repair test suites on benchmarks. *Raw* column is the raw data and *Trans* column is the transformed data using *sqrt* function.

5.3.2.4.1 *RQ1: APR Effectiveness*. To compare APR effectiveness when different repair test suite size were used to repair faults, we compared the average success rate (Table 5.7). The average success rate when small repair test suites were used is 42%, while the average success rate when large repair test suites were used is 19%. Figure 5.10a shows the average success rate per program. For all the programs, except replace program, using small repair test suites increased the average success rate compared to the large repair test suites. For eight out of seventeen faulty versions of replace program small repair test suites decreased success rate. The difference between success rate for small and large repair test suites is significant (p-value = 0.00) at the 0.95 confidence level using Wilcoxon Signed-Rank Test. These results support those found by Nguyen et al. [4]. Figure 5.10b and 5.10c show the success rate distributions for different repair test suite sizes.

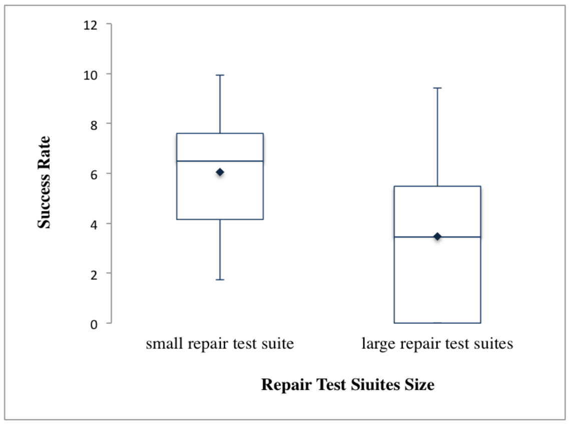
5.3.2.4.2 *RQ2: APR Repair Correctness*. We compared repair correctness by first measuring the PFR when each repair test suite size were used (Table 5.7). The average PFR



(A) Average Success Rate Per Program



(B) Raw Success Rate Data



(C) Transformed Success Rate Data

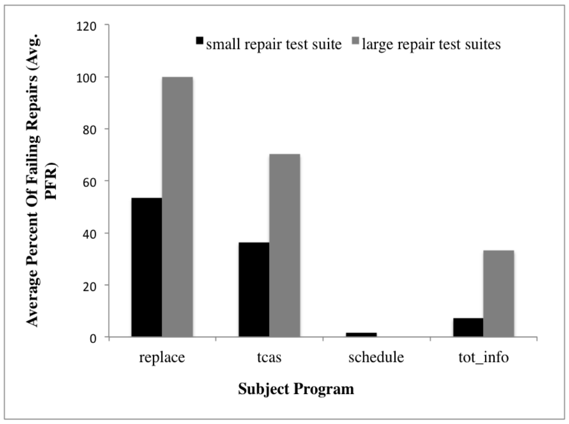
FIGURE 5.10. MUT-APR repair effectiveness using different repair test size. Higher success rate is better.

when small repair test suites were used is 31.3%, and the average PFR for large repair test suites is 89.1%. Figure 5.11a shows the average PFR per program. Small repair test suites decreased the average PFR for all subject programs except schedule2. Small repair test suites generated more failing potential repairs for schedule2 program (average PFR for small repair test suites is 1.6) compared to large repair test suites which generated no failing potential repairs. The difference in PFR for different repair test suite size is significant (p-value = 0.000) at the 0.95 confidence level. Figures 5.11c and 5.11d show the PFR distributions for different repair test suite sizes.

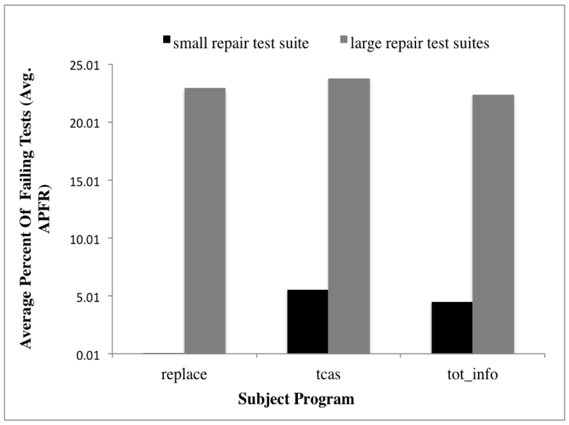
Then, we compared how far each failing potential repair is from being a validated one by comparing the APFT. APFT for small repair test suites is 3.5%, and APFT for large repair test suites is 13.7%. Figure 5.11b is the average APFT per subject program. We did not include schdule2 in the graph because using large repair test suites produced potential repairs for only one version of schedule2, and the produced potential repair is a validated repair—it does not fail any regression tests (PFR = 0). For the rest of the subject programs, small repair test suites generated potential repairs that failed fewer regression tests. We studied the statistical difference using Wilcoxon Signed-Rank Test; the difference is significant (p-value = 0.00) at the 0.95 confidence level. Figures 5.11e and 5.11f are the APFT distributions for each repair test suite size. In summary, smaller sized repair test suites generated more validated repairs, and failing potential repairs failed fewer regression tests.

5.3.2.4.3 *RQ3: APR Performance*. APR performance is measured by the ANGV and the ATotalTime required to find potential repairs for *N* potential repairs for each faulty version. (Table 5.7). The ANGV for small repair test suites is 186.8 variants, and the ANGV for large repair test suites is 203 variants. Figure 5.12a is the ANGV per program. Using small repair test suites decreased the ANGV compared to the use of large repair test suites for all faulty programs except tot_info. We analyzed the statistical difference using Paired t-Test since the ANGV distribution is normal; the difference in ANGV when small and large repair test suites were used is significant (p-value = 0.000) at the 0.95 confidence level. Figures 5.12c and 5.12d show the ANGV distribution when different repair test suite sizes were used.

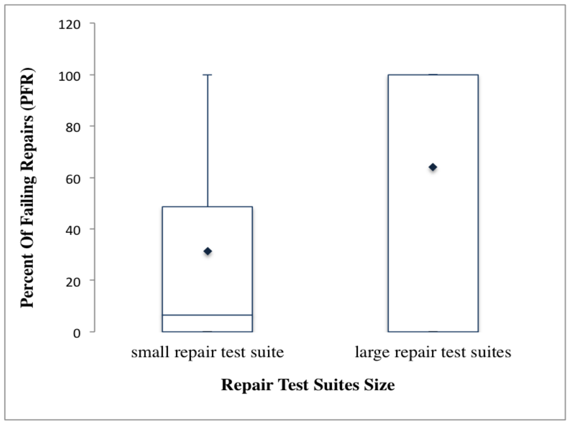
Then, we compared APR performance using the ATotalTime metric. The ATotalTime required to repair faults when small repair test suites were used is 13.4 seconds, while the ATotalTime when large repair test suites were used is 39.2 seconds. Figure 5.12b shows the ATotalTime to fix faults per program. Small repair test suites decreased the ATotalTime



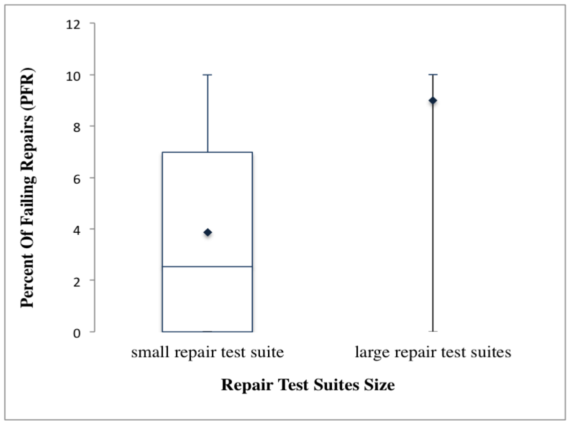
(A) Average PFR Per Program



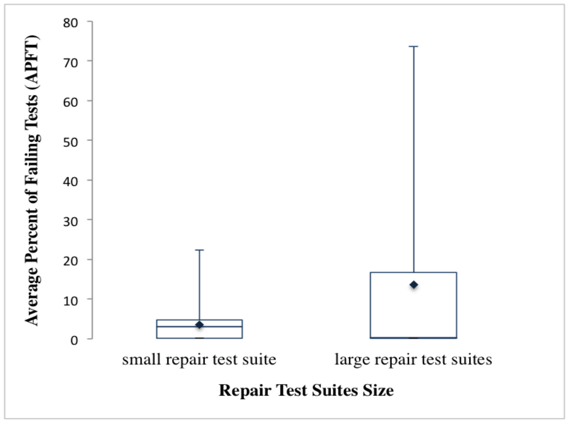
(B) Average APFT Per Program



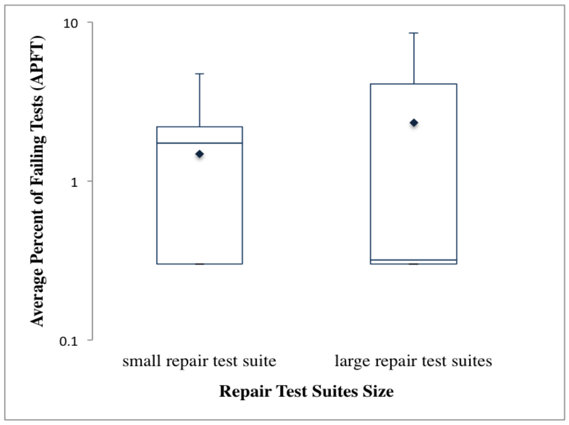
(C) Raw PFR Data



(D) Transformed PFR Data



(E) Raw APFT Data



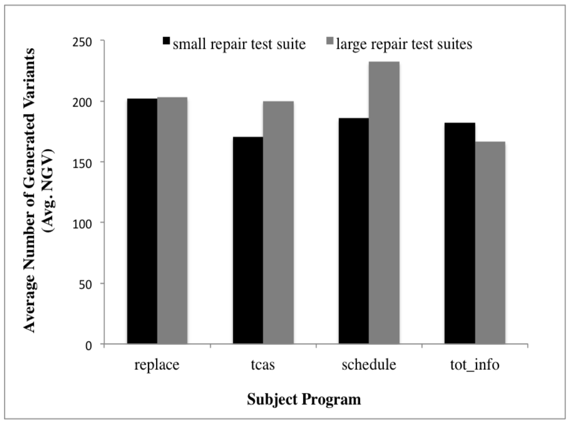
(F) Transformed APFT Data

FIGURE 5.11. MUT-APR repair correctness using different repair test suite size to repair faults. Lower PFR and APFT is better.

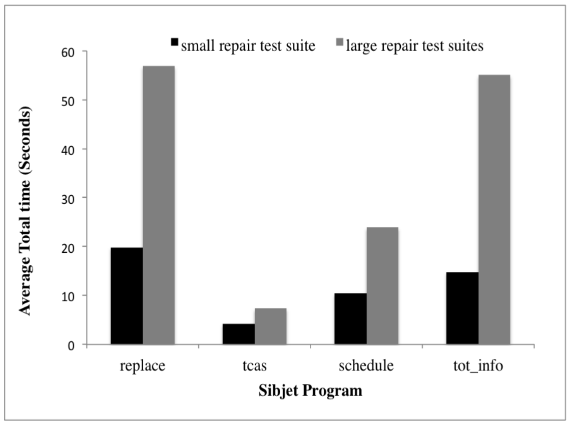
required to fix faults for all subject programs; the difference is significant (p-value = 0.00) at the 0.95 confidence level using Wilcoxon Signed-Rank Test. Figure 5.12e and Figure 5.12f show the ATotalTime distribution when different repair test suite sizes were used.

5.3.3. Conclusions and Lessons Learned. Repairing faults can introduce new faults when the selected repair tests do not provide good coverage. Using repair test suites that satisfy branch coverage can improve the quality of generated potential repairs significantly by generating more validated repairs (reduce PFR and APFT) compared to repair test suites that satisfy statement coverage and randomly generated repair tests. Branch coverage repair test suites also improved average success rate compared to statement coverage and random repair test suites, but the difference is not significant. On the other hand, branch coverage repair test suites lowered APR performance by generating more variants and requiring more time to repair a fault. This might be due to the coverage level provides by branch coverage repair test suites. Branch coverage repair test suites include test cases for each feasible branch in the program under test, thus each branch that produces the correct output is protected from unwanted modification by a at least one passing test. The APR tool modifies the locations that are executed by failing tests. Therefore, branch coverage repair test suites give a good guidance to locations that can be modified by the APR tool, thereby produce more validated repairs, but require more time compared to the other test methods.

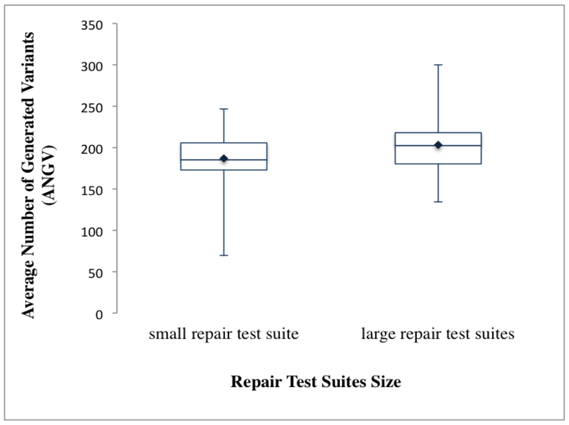
From our evaluation of different repairs test suite sizes, small repair test suites improved APR effectiveness significantly. In addition, repair quality and performance are improved significantly using small repair test suites. Thus, using a large number of repair tests decreased the success rate, since it hard to find a potential repair that passes a large number of tests. In addition, if potential repairs are found, that requires large time since all repair tests must be executed for each generated variant to compute its fitness value.



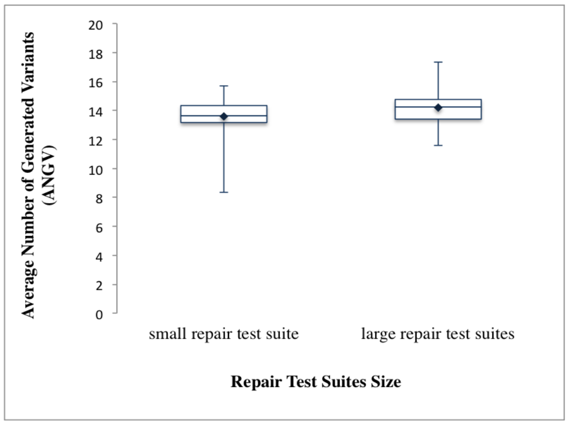
(A) Average ANGV Per Program



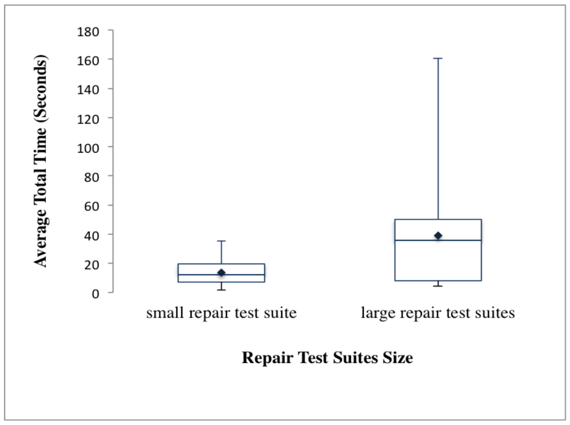
(B) Average ATotalTime per program



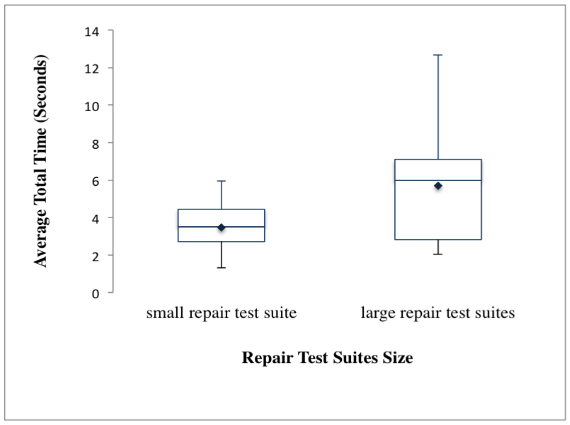
(C) Raw ANGV Data



(D) Transformed ANGV Data



(E) Raw ATotalTime Data of



(F) Transformed ATotalTime Data

Figure 5.12. MUT-APR performance using different repair test suite sizes to repair faults. Lower ANGV and AtotalTime is better.

5.4. Fault Localization Techniques

In this section, we study the impact of different fault localization techniques on APR effectiveness and performance.

5.4.1. Research Questions.

Our evaluation study is designed to answer the following research questions:

- RQ1 APR Effectiveness: What is the relative APR effectiveness when different FL techniques are employed?
- RQ2 APR Performance: What is the relative APR performance when different FL techniques are employed?

5.4.2. Evaluation Design.

We used the Siemens Suites: tcas, replace, schedule2, and tot_info. We also used two larger programs: space and sed. For each subject program, we excluded faulty versions when faults were seeded in the same operator since fault localization techniques gave the same results. We also excluded faulty versions when none of the fault localization techniques identify actual faulty statements such as replace-v25 and replace-v30.

One of the inputs to an APR tool is a set of repair tests. We selected a set of repair tests that have at least one failing test, and one passing test. Repair test suites for the Siemens Suites are taken from the SIR repository. Test suites for the large programs provided by the SIR contain too many test inputs, which will slow the APR process, since repair tests are used to validate each generated variant. We created repair test suites for each large program containing at least one failing test and 20 passing tests following Qi et al.[27]. In addition, a study by Abreu et al. [21] found that fault localization techniques give a stable behavior when no less than 20 test inputs are used.

Program	# Faulty Versions	Average # Repair Tests
tcas	4	6.4
replace	5	19
schedule2	2	9
tot_info	4	8
space	2	38.4
sed	1	28
Total	19	108.9

TABLE 5.8. Benchmark programs to study the impact of fault localization techniques. Each *Program* is an original program from the SIR [50].*#Faulty Versions* is the number of faulty versions. *Average # Repair Tests* is the average number of repair tests for each faulty version.

To validate our results, we repeated the study for each faulty version using five different repair test suites that are selected/created randomly using test data provided by the SIR except for two versions of tcas program (tcas-v5 and tcas-25), which used three and one repair test suites, respectively, since there are no other test suites that execute faults. Table 5.8 identifies the subject programs, the number of faulty versions, and the average size of repair test suites.

To fix faults, we used our MUT-APR repair tool. For this experiment, we applied a brute-force search algorithm to accurately study the impact of fault localization techniques on APR effectiveness and performance. A brute-force search algorithm guarantees a potential repair when the FL technique identifies the faulty statements and the repair is supported by the set of program modification operators.

For each faulty version, we used each FL technique to create an LPFS. Then the list is used by MUT-APR to find a potential repair. We executed each FL technique five times on each faulty version with five different repair test suites except for tcas. On one version of tcas we used three different repair test suites, and on the other version of tcas we used one repair test suite. We excluded the trials in which at least one FL technique did not identify the faulty statements, and we excluded two versions of replace program in which the faulty statements were not identified by all FL techniques. In total, our evaluation includes 79 trials (faulty versions × number of test suites for each faulty version). We compute the average value of our measurements for each faulty version across the multiple test suites.

5.4.3. Measurements. APR effectiveness is the ability to fix faults. The effectiveness of APR when different FL techniques is measured by the ability of an FL to identify actual faulty location. An FL technique that successfully determines faulty locations improves APR effectiveness. On the other hand, an FL technique that fails to identify actual faulty locations limits APR effectiveness.

We first studied the accuracy of FL technique using the LPFS rank metric; LPFS rank of a faulty statement indicates the quality of the FL technique alone. LPFS rank is the position of a statement in the LPFS. Statements with higher suspiciousness scores are placed near the head of the list, thus have a lower LPFS rank compared to other statements. For example, in Table 5.13a statement ID 2 has the lowest LPFS rank (LPFS rank =1) since it has the highest suspiciousness score. On the other hand, statement ID 4 has the highest LPFS rank (LPFS rank = 7). An FL technique that assigns higher suspiciousness score to a faulty statement, placing it near the head of the LPFS (lower LPFS rank), improves APR performance compared to another FL technique that places a faulty statement far from the head of the LPFS (higher LPFS rank).

To study APR performance of different FL techniques, we used the average number of generated variants (ANGV) for N potential repairs for each faulty version, and the average total time (ATotalTime) for N potential repairs for each faulty version; N is the number of potential repairs generated for 100 trials for each faulty version. Since we are applying a brute-force search algorithm in this evaluation, $N = 1$ because the brute-force algorithm

Statement ID	Suspiciousness score	LPFS rank
2	0.96	1
3	0.91	2
1	0.80	3
5	0.71	4
7	0.68	5
8	0.57	6
4	0.5	7

(A) LPFS1

Statement ID	Suspiciousness score	LPFS rank
5	0.96	1
8	0.91	2
1	0.80	3
2	0.72	4
3	0.6	5
4	0.5	6
10	0.5	7

(B) LPFS2

FIGURE 5.13. List of Potentially Faulty Statements (LPFS) for *gcd* created by two FL techniques: *LPFS1* is creared by FL1 and *LPFS2* is created by FL2

executes one trial for each faulty version. Thus, the ANGV and the ATotalTime are equivalent to the NGV and TotalTime, which we used to compare the results (lower NGV and TotalTime are better). An FL technique that assigns a low LPFS rank to a faulty statement requires fewer statements to be modified, thus creating fewer variants (reducing NGV).

The difference between the LPFS rank metric and NGV is that the LPFS rank metric is totally dependent on the FL technique; however, NGV can be influenced by the number of mutable statements with lower LPFS ranks than the faulty statement. For example, consider the use of two FL techniques (FL1 and FL2) to identify a faulty statement in the *gcd* program (Figure 3.2). FL1 creates LPFS1 (Table 5.13a), and FL2 creates LPFS2 (Table 5.13b). Both techniques assign the same LPFS rank for the faulty statement (the faulty statement, statement ID 1, in both lists has an LPFS rank = 3). However, NGV depends on the number of mutable statements prior to the faulty statement. LPFS1 consists of two statements prior to the faulty statement (statement 2 and 3) but neither can be mutated by MUT-APR program modification operators; thus NGV can be equal to any value between 1 and 5 (depending on the order of alternative program modification operators that transform faulty operator $<$ into the correct one $==$). On the other hand, LPFS2 consists of two mutable statements prior to the faulty statement (statement 5 and 8), thus NGV can be equal to any value between 3 and 15.

Total time is the time required to find a potential repair. We compared the total time, measured in seconds, when each FL technique is used.

5.4.4. Results.

We compared the values of each individual subject program using our measurements. Then, we compared the average values of each metric (LPFS rank, NGV, and TotalTime) to measure the impact when using each FL technique. We compared the distributions of the measurements for the raw data and the transformed data using *sqrt* function to make the data more normal. Then, to measure the statistical difference we applied the Mixed Model statistic test [85], which is preferable to ANOVA, to study the effect of many measurements using the same subject program. The Mixed Model assumes normality of the data. Since our data is not normally distributed, we applied the Mixed Model using the ranks of the transformed data. We studied the difference at the 0.95 confidence level.

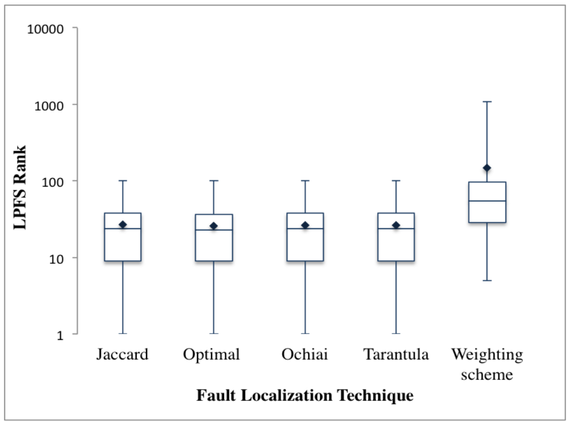
5.4.4.1. *RQ1: APR Effectiveness.*

To compare APR effectiveness when different FL techniques are used, we studied the ability of FL techniques to identify actual faulty statements. If an FL technique failed to identity actual faulty statements, APR will fail to repair the faults. The four FL techniques successfully identified actual faulty statements for all faulty versions except that Optimal failed to identity the actual faulty locations for two trials of the space program. We investigated the reason for Optimal's failure. In these two faulty versions, faults were in statements that are executed by some failing tests, which makes the number of non-executed failing tests for these statements greater than zero. Optimal assumes that a faulty statement will be executed by all failing tests for a single-fault program. Thus, if a statement is executed by some but not all the failing tests (number of non-executed failing tests > 0), Optimal does not identify this statement as faulty. In trials of these two versions of space program, there were two non-executed failing tests (number of non-executed failing tests $= 2$), thus optimal failed to identify these statements as faulty statements.

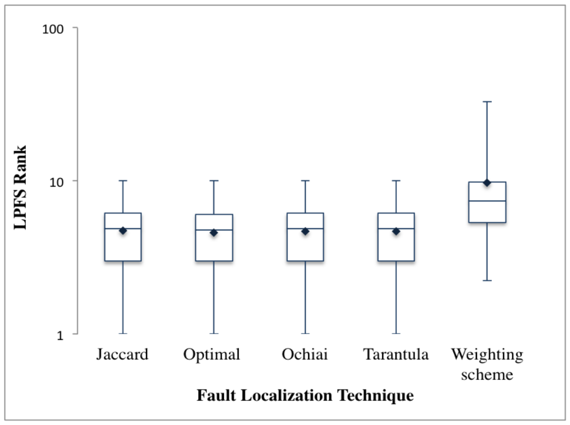
All FL techniques failed to identify the actual faulty statement in two faulty versions of replace (version 25 and version 31). We checked these two versions of replace to investigate why the FL techniques failed to identify these faulty statements. In both versions, faults were in an *if* statement nested inside a *switch* statement. When the program is executed against the failing tests, the *if* statement is checked and it returns false. Thus, the execution jumps to the *default* statement, which caused failures. However, the faulty *if* statements were not recorded as one of the executed statements. We are not certain of the reason; it might be a problem in the code that creates the instrumented version of faulty programs which records coverage information.

5.4.4.2. *RQ2: APR Performance.*

Which FL technique ranked actual faulty statements lowest? We evaluated the accuracy of FL techniques in identifying actual faulty statements by comparing the LPFS rank of the actual faulty statement produced by each FL technique. We excluded the two versions of the replace program in which the actual faulty statements were not identified. We compared the LPFS rank of actual faulty statements using seventeen faulty versions (79 trials).



(A) Raw LPFS Rank Data



(B) Transformed LPFS Rank Data

FIGURE 5.14. LPFS rank for each FL technique. Lower LPFS rank is better.

Figure 5.14a and Figure 5.14b show the mean, median, and the distributions of LPFS rank across all trials.

First, we compared the LPFS rank of actual faulty statements when using Jaccard, Optimal, Ochiai and Tarantula to that of the Weighting Scheme for each trial. We found that in, 12 out of 79 (15.18%) of the trials, the Weighting Scheme assigned the lowest LPFS rank to actual faulty statements, but in 67 out of 79 (84.8%) of the trials Jaccard, Optimal, Ochiai and Tarantula assigned the lowest LPFS rank to actual faulty statements.

Then, we compared the behavior of FL techniques when using different test suites for each faulty version. We compared the LPFS ranks given by Jaccard, Optimal, Ochiai and Tarantula, since they have similar behavior. For the small subject programs (58 trials), all four FL techniques assigned the same LPFS rank to the actual faulty statements across the five test suites used for each faulty version. For space and sed programs, all FL techniques assigned different LPFS rank to the faulty statement across the five test suites; however, the difference is marginal.

We compared FL across all trials. For replace version 5 and tot_info versions 7 and 13, all four FL techniques performed equally by assigning the same LPFS rank for actual faulty

Measurement		Jaccard		Optimal		Ochiai		Tarantula		Weighting Scheme	
		Raw	Trans.	Raw	Trans.	Raw	Trans.	Raw	Trans.	Raw	Trans.
LPFS Rank	Mean	27.0	4.74	25.7	4.59	26.5	4.69	26.4	4.67	147.1	9.72
	Median	24.0	4.89	23.0	4.79	24.0	4.89	24.0	4.89	55.00	7.42
	Std. Dev.	21.7	2.16	21.4	2.14	21.4	2.14	21.3	2.14	265.2	7.29
NGV	Mean	36.3	5.24	34.4	5.08	35.5	5.17	35.6	5.17	211.8	11.55
	Median	24.0	4.89	22.0	4.69	24.0	4.89	26.0	5.09	90.00	9.490
	Std. Dev.	34.1	2.99	33.2	2.93	33.6	2.96	33.9	3.00	365.5	8.910
TotalTime	Mean	74.20	5.69	28.6	4.29	72.50	5.61	67.60	5.55	503.50	11.51
	Median	17.00	4.13	11.9	3.46	15.60	3.95	15.50	3.94	25.100	5.010
	Std Dev.	211.8	6.50	36.1	3.21	214.4	6.45	180.7	6.11	1826.9	19.39

TABLE 5.9. MUT-APR performance when using Jaccard, Optimal, Ochiai, Tarantula and the Weighting scheme on each faulty version. *Raw* column is the raw data and *Trans* column is the transformed data using *sqrt* function.

statements in most (89.66%) of the trials. In 6 out of 58 (10.34%) of the trials Optimal and Tarantula assigned the lowest (best) LPFS rank to actual faulty statements compared to Jaccard and Ochiai, but in these trails Ochiai performed better than Jaccard by assigning a lower LPFS rank to actual faulty statements. In 3 out of 58 (5.17%) trials, Jaccard assigned the highest (worst) LPFS rank to actual faulty statements compared to Optimal, Ochiai and Tarantula.

For the space program, Optimal failed to identify actual faulty statements in two trials (one trial for each faulty version), but it assigned the same or lowest LPFS rank to actual faulty statements compared to Jaccard, Ochiai and Tarantula in the other trials. For the sed program, Optimal assigned the same or lowest (best) LPFS rank to actual faulty statements compared to other FL techniques.

We compared the mean LPFS rank for all FL techniques (Table 5.9). On average, Jaccard, Optimal, Ochiai and Tarantula assigned lower (better) LPFS ranks to faulty statements than the Weighting Scheme. The average LPFS ranks of Optimal and Tarantula is slightly better than Jaccard and Ochiai. We also compared the median; Optimal assigned a lower

LPFS rank to faulty statements, but the difference is just one position compared to Jaccard, Ochiai and Tarantula.

One thing to notice here is the large average and standard deviation values for the Weighting Scheme, as shown in Table 5.9. These values relate to the size of programs used in our evaluations. The data set includes large programs that required checking more than one thousand statements before reaching faulty statements, thus a very high rank was assigned to actual faulty statements which skew the average and the standard deviation values of LPFS rank, NGV, and TotalTime.

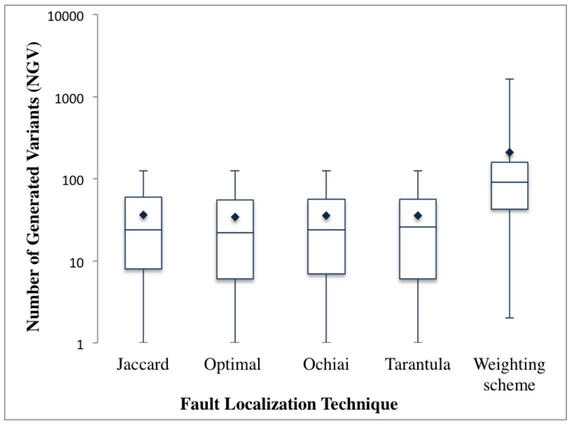
In summary, all four FL techniques assigned lower (better) LPFS ranks to actual faulty statements compared to the Weighting Scheme, and the difference is significant between the Weighting Scheme and all four alternative FL techniques (Jaccard, Optimal, Ochiai, and Tarantula) at the 0.95 confidence level. Optimal and Tarantula had the worst performance in 6 trials, and Jaccard was the worst in 3 trials. Although Ochiai was not the best FL technique based on average LPFS rank, it always identified actual faulty statements, and it never assigned the highest (worst) LPFS rank to actual faulty statements across all trials. The differences are not significant between Jaccard, Optimal, Ochiai, and Tarantula.

Number of generated variants (NGV). In order to repair faults, APR tools apply program modification operators to modify a location in the faulty program to generate variants. Modifying a non-faulty location generates an invalid variant (a variant that does not pass all repair tests). If the faulty statement is placed earlier in the LPFS, then the APR will change the faulty statement and produce fewer invalid variants.

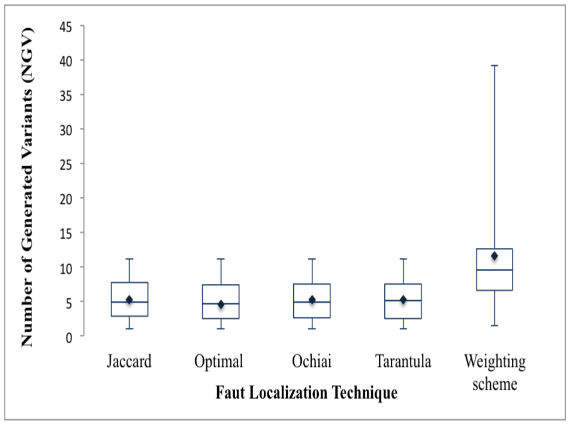
We compared the number of generated variants (NGV) until a potential repair is found to determine the most effective FL technique for use in APR. An FL technique that gives lower NGV is better since it decreases the chances of producing invalid variants before finding a potential repair. First, we compared the NGV produced when using Jaccard, Optimal, Ochiai and Tarantula to that produced using the Weighting Scheme. In 5 out of 79 (6%) trials, the Weighting Scheme decreased NGV compared to other four FL techniques, and, in 4 out of 79 (5%) trials, the Weighting Scheme generated the same NGV as other FL techniques. The Weighting Scheme increased the NGV in 68 out of 79 trials (86.1%). Then we compared the NGV for Jaccard, Optimal, Ochiai and Tarantula. For the small benchmarks (59 trials), all FL techniques generated the same NGV except in 4 out of 59 (6.78%) trials in which Jaccard generated more variants (high NGV) until a potential repair is found. For space and sed programs, in 6 out of 20 (30%) of the trials Optimal decreased NGV, and in one out of 20 (5%) trials Jaccard decreased NGV. In 11 out of 20 trials (55%) Jaccard, Optimal, Ochiai, and Tarantula generated the same number of variants until a potential repair is found.

Table 5.9 shows the results summary NGV for each FL technique across all trials. Average NGV is almost similar for all techniques except the Weighting Scheme, which generated an average of 211.8 variants. We also compared the median values; the Weighing Scheme required 90 variants while other FL techniques required between 22 and 26 variants. Figures 5.15a and 5.15b show the distributions of NGV for each FL technique. The difference is significant between the Weighting Scheme and all alternative FL techniques at the 0.95 confidence level (p-values < 0.0001).

Total time. We compared the total time required to find a potential repair when each FL technique is used to generate the LPFS. We compared the average time for each FL technique. Table 5.9 shows that using Optimal required the shortest average total time (28.6 seconds) to repair faults, and using Jaccard, Ochiai and Tarantula required 74.2, 72.5, 67.6 seconds, respectively. Using the Weighting Scheme required an average of 503.5 seconds to find a potential repair. The average and the standard deviation values of FL techniques



(A) Raw NGV Data

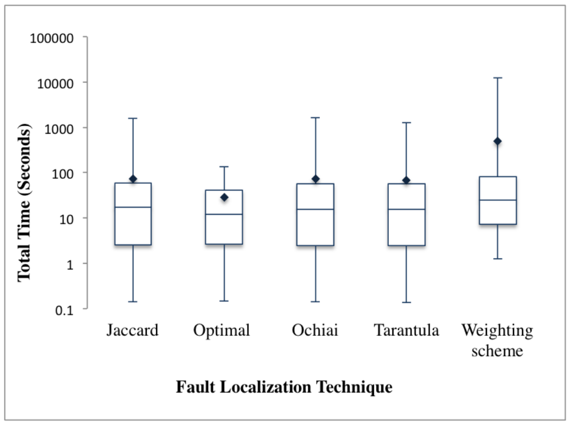


(B) Transformed NGV Data

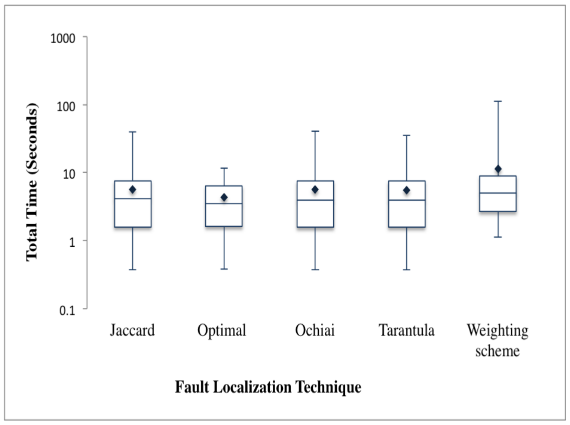
FIGURE 5.15. MUT-APR NGV required to find potential repairs for each FL technique. Lower NGV is better.

are very high due to the variation of program size in our data set; the small programs ranges between 173 and 560 LOC and large programs have more than 5000 LOC. Large programs required a long time to find potential repairs, which in response effects the average and the standard deviation. The Weighting Scheme is more sensitive to the program variation size; the difference in TotalTime between the small and large programs is very large (e.g, the TotalTime minimum value for the Weighing Scheme is 1.258 seconds and the maximum value is 122486 seconds, while for other FL technique, such as Ochiai, Totaltime value is 0.14 seconds and maximum value is 1556.45 seconds). Figure 5.16a and Figure 5.16b show the distributions of the total time required to repair faults when each FL technique is used to identify faulty statements. The total time improvement of Jaccard, Optimal, Ochiai, and Tarantula over the Weighting Scheme (p-values < 0.0001) is significant at the 0.95 confidence level.

5.4.5. CONCLUSIONS AND LESSONS LEARNED. Fault localization techniques that identify actual faulty statements will improve APR effectiveness. An FL technique that places actual faulty statements at the head of the list of potentially faulty statements (LPFS), will



(A) Raw TotalTime Data



(B) Transformed TotalTime Data

FIGURE 5.16. MUT-APR Total time required to find potential repairs for each FL technique. Lower TotalTime is better.

Correlation	Jaccard	Optimal	Ochiai	Tarantula	Weighting Scheme
LPFS Rank-NGV	0.949	0.872	0.985	0.986	0.988
LPFS Rank-Time	0.050	0.332	0.277	0.342	0.784
NGV-Time	0.110	0.492	0.309	0.375	0.776

TABLE 5.10. Correlation results between performance metrics: LPFS Rank, Number of Generated Variants (NGV), and Time of Jaccard, Optimla, Ochiai, Tarantula and the Weighting Scheme.

improve APR performance since fewer variants will be generated before a potential repair is found, thus decreasing the time required to fix faults. Our evaluation shows that MUT-APR is least effective when Optimal was used, and has the worst performance when the Weighting Scheme was used to localize faults. However, Ochiai never failed to identify actual faulty statements and never assigned the highest LPFS rank to actual faulty statements compared to the other FL techniques. APR performance is measured in terms of NGV, and total time. LPFS Rank and NGV are strongly correlated, and they are not platform dependent compared to the time metric (Table 5.10).

5.5. Search Algorithms

This section reports on our evaluation of the impact of different search algorithms on APR effectiveness, performance and repair quality.

5.5.1. Research Questions. We designed experiments to answer the following research questions:

- RQ1 APR Effectiveness: What is the relative APR effectiveness when different search algorithms are employed?
- RQ3 APR Repair Correctness: Does the use of different search algorithms affect the correctness of generated potential repairs?
- RQ2 APR Performance: What is the relative APR performance when different search algorithms are employed?

5.5.2. Evaluation Design. Our evaluation applies six different stochastic and two exhaustive search algorithms to repair faulty operators. We selected a set of repair test suites that have at least one passing and one failing test, and satisfy the branch coverage criterion since our previous study (Section 5.3) showed that the use of branch coverage repair test suites produces more validated repairs. We used the Ochiai fault localization technique to generate the LPFS since our evaluation of the impact of different fault localization techniques on APR showed that Ochiai is one of the best techniques to place actual faulty statements near the head of the LPFS (Section 5.4). To ensure the accuracy of our results, we used subject programs where Ochiai identified faulty statements, and a potential repair was successfully found using a brute-force search algorithm. Table 5.11 includes the subject programs along the number of faulty versions for each program, and the average number of statements in the LPFS.

Our evaluation used our repair tool MUT-APR with implementation of six stochastic algorithms: (1) genetic algorithm (GA), (2) genetic algorithm without a crossover operator (GAWoCross), (3) random search (RS), (4) guided genetic algorithm (GID-GA), (5) guided genetic algorithm without a crossover operator (GID-GAWoCross), and (6) guided random search (GID-RS), and two exhaustive search algorithm: brute-force (BF) and Ordered-brute-force (Ordered-BF).

Since stochastic search algorithms select program modification operators randomly, we ran each algorithm 100 times on each faulty version so that it is more likely to fix a fault in at least one execution; each execution is called a *trial*. Each trial of GA, GAWoCross, GID-GA, and GID-GAWoCross consists of many generations/iterations of the genetic algorithm loop. Each generation consists of a population size that is equal to the number of potentially faulty statements in the LPFS (Table 5.11).

We investigated the use of five and ten generations with the genetic algorithm (GA). Increasing the number of generations improved the success rate. The average success rate improvement is 21.8%. Since we are generating a large population for each generation, we execute the genetic algorithms for only five generations and compare the results. The random searches (RS and GID-RS) run for one generation since they do not apply a selection or crossover operator. Thus, each execution of RS and GID-RS consists of one large generation and population size of $5 \times |LPFS|$.

5.5.3. Measurements. To assess effectiveness, we computed the success rate. The success rate is the percentage of the trials resulting in a potential repair. An algorithm that generates more potential repairs improves the success rate. To assess repair quality we measured the percent of failing potential repairs (PFR) and the average percent of failing regression tests (APFT). PFR is the percent of potential repairs that failed at least one

Program	Faulty Ver.	Average $\|LPFS\|$
tcas	13	60
replace	10	178
schedule2	6	97.3
tot_info	9	92.9
space	2	619
sed	1	1388
Total	41	3921.2

TABLE 5.11. Benchmark programs to study the impact of different search algorithms. Each *Program* is an original program from the SIR [50]. *#Faulty Ver.* is the number of faulty versions. *Average* $|LPFS|$ is the average number of statements in the LPFS.

regression test for 100 trials for each faulty version, and APFT is average number of failing regression tests for *N* potential repairs for each faulty version; *N* is the number of potential repairs generated for 100 trials for each faulty version (lower is better). To assess APR performance, we computed the average number of generated variants (ANGV) for *N* potential repairs for each faulty version and the average total time (ATotalTime) *N* potential repairs for each faulty version (lower is better).

5.5.4. RESULTS. We compared the values of each individual subject program using our measurements. Then, we compared the average values of each metric (success rate, PFR, APFT, ANGV, and ATotalTime) to measure the impact when using each algorithm. We excluded faulty versions where no potential repair is found by applying an algorithm, and we computed the average of remaining faulty versions. We compared the distributions of the measurements using the raw data and transformed data; we used the *sqrt* function to make the data more normal. To measure the statistical difference, we applied Wilcoxon Signed-Rank Test, and Mann-Whitney U Test [86] at the 0.95 confidence level. Wilcoxon Signed-Rank Test is applied with paired data and Mann-Whitney U Test is applied with non-paired data.

5.5.4.1. *RQ1: APR Effectiveness.* APR effectiveness of different search algorithms includes only the stochastic search algorithms since an exhaustive search algorithm guarantees a potential repair (100% success rate). To evaluate APR effectiveness when different stochastic search algorithms were used, we compared the average success rate for each algorithm (Tables 5.12 and 5.13). The average success rate for GA is 10.3%, for GAWoCross is 14.8%, and for RS is 15.2%. The RS algorithm improved the average success rate compared to the two versions of the genetic algorithm (GA and GAWoCross). The difference between RS and GA success rate is 4.9%, and between RS and GAWoCross average success rate is 0.4%. We analyzed the statistical difference by Wilcoxon Signed-Rank Test; the difference between RS and GA success rate is significant (p-values = 0.001); however, the difference between RS and GAWoCross is not significant (p-value = 0.452) at the 0.95 confidence level. Thus, RS improved the success rate compared to the two versions of genetic algorithms (GA and GAWoCross). These results are different than those found using GenProg [43, 87]; they found that removing crossover operator decreased success rate. The difference between GenProg's and our results due to the PMOs used. MUT-APR's PMOs fix single operator faults, thus combining changes from two variants, as done by the crossover operator, takes the variant away from being a potential repair. Therefore, there is no advantages from applying a crossover operator within MUT-APR.

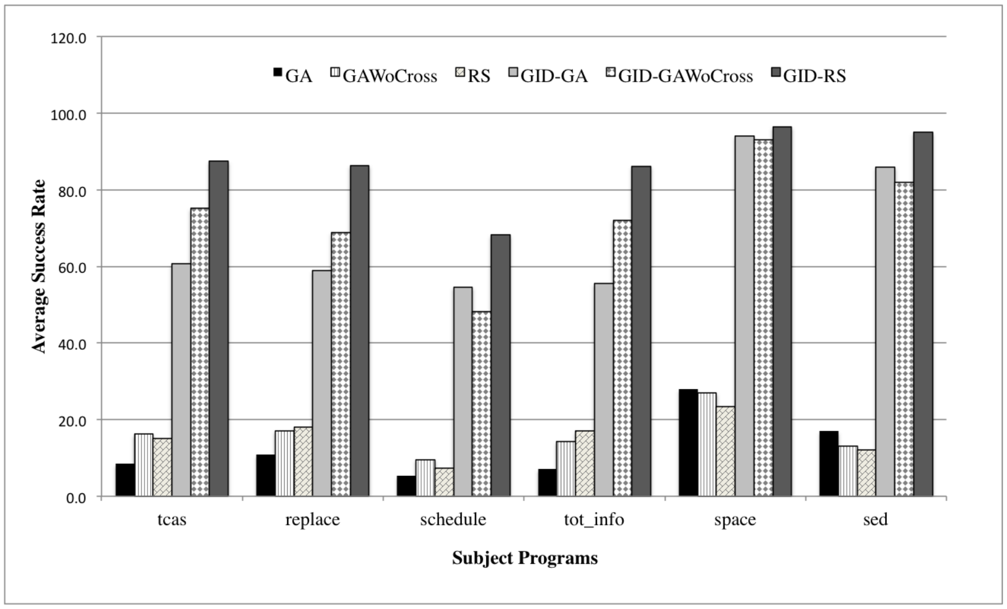
Then, we compared the average success rate for the guided stochastic search algorithms. The average success rate for GID-GA is 59.1%, for GID-GAWoCross is 68.8%, and for GID-RS is 83%. Thus, the guided version of each algorithm improved the average success rate compared to its corresponding original algorithm (e.g., GID-GA average success rate is 59.1% and GA average success rate is 10.3%), and the difference is significant (p-values = 0.000). Of all of the tested stochastic search algorithms, the guided random search (GID-RS) is

Measurements		GA	GAWoCross	RS	GID-GA	GID-GAWoCross	GID-RS	BF	Ordered-BF
Success	Mean	10.3	14.8	15.2	59.1	68.8	83.0	100	100
	Median	7.00	13.0	12.0	60.0	72.0	93.0	100	100
	Std. Dev.	10.3	9.50	9.90	22.8	21.0	18.5	0.0	0
PFR	Mean	35.0	39.2	61.1	42.3	43.8	38.9	43.9	36.5
	Median	9.00	22.2	75.8	28.9	43.8	20.4	-	-
	Std. Dev.	42.7	43.5	43.0	42.1	40.3	43.4	-	-
APFT	Mean	26.6	5.4	18.6	6.30	19.2	6.50	2.40	1.5
	Median	12.4	0.2	4.30	3.20	3.50	1.30	0.10	0.1
	Std. Dev.	25.7	9.1	24.4	14.1	24.7	18.4	3.23	2.9
ANGV	Mean	491.5	588.7	378.7	236.6	272.0	63.9	64.80	64.90
	Median	319.0	366.4	239.0	173.5	202.7	53.5	18.50	17.00
	Std. Dev.	697.8	581.2	580.0	278.8	288.8	71.0	152.3	152.4
ATotalTime	Mean	36.60	51.60	36.60	47.5	44.31	38.3	40.40	61.70
	Median	6.700	4.300	7.400	13.3	16.80	14.2	3.500	3.400
	Std. Dev.	133.4	200.3	109.5	94.6	94.80	89.7	145.5	196.6

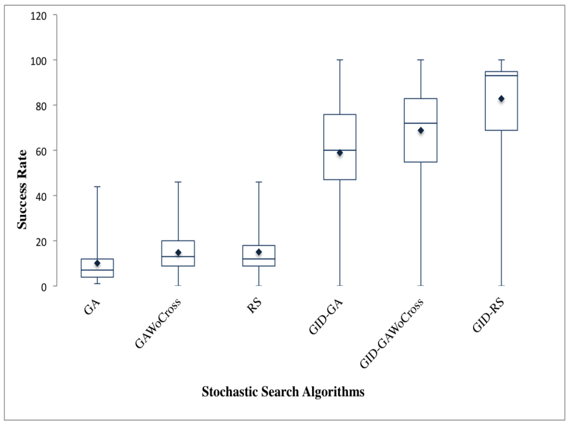
TABLE 5.12. Applying different search algorithms. *Success* is the average success rate. *APFT* are the average percent of failing potential repairs and the average percent of failing regression tests for *N* repairs for each faulty version; *N* is the number of repairs generated for 100 trials for each faulty version. *ANGV* is the average number of generated variants until a potential repair is found for *N* repairs for each faulty version. *AToalTime* is the average time required to fix faults until a potential repair is found for *N* repairs for each faulty version. *PFR* and

Measurements		GA	GAWoCross	RS	GID-GA	GID-GAWoCross	GID-RS	BF	Ordered-BF
Success	Mean	2.94	3.62	3.69	7.47	8.13	8.98	10	10
	Median	2.65	3.61	3.46	7.75	8.49	6.64	10	10
	Std. Dev.	1.34	1.33	1.25	1.87	1.70	1.61	0	0
PFR	Mean	4.05	4.44	6.59	5.22	5.55	4.35	-	-
	Median	3.02	4.71	8.69	5.38	6.62	3.19	-	-
	Std. Dev.	4.34	4.48	4.27	3.94	3.66	4.54	-	-
APFT	Mean	4.26	1.63	3.11	1.88	3.19	1.59	1.16	0.82
	Median	3.53	0.44	2.09	1.80	1.87	0.97	0.30	0.30
	Std. Dev.	2.98	1.71	3.05	1.69	3.07	2.06	1.07	0.96
ANGV	Mean	18.73	22.29	16.94	13.54	14.96	7.02	6.07	6.14
	Median	17.86	19.14	15.46	13.15	14.24	7.32	4.30	4.12
	Std. Dev.	12.00	9.710	9.690	7.380	7.040	3.88	5.35	5.29
ATotalTime	Mean	3.52	3.73	3.77	5.00	5.050	4.31	3.86	4.43
	Median	2.61	2.09	2.73	3.65	4.100	3.77	1.88	1.86
	Std. Dev.	4.99	6.22	4.78	4.81	4.739	4.49	5.13	6.58

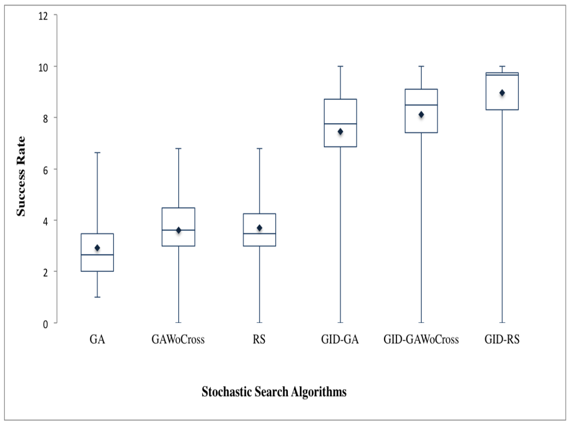
TABLE 5.13. Data from Table 5.12 transformed using the *sqrt* function to make data more normal.



(A) Average Success Rate Per Program



(B) Raw Data of Success Rate



(C) Transformed Data of Success Rate

FIGURE 5.17. MUT-APR effectiveness when applying different stochastic search algorithms. Higher success rate is better.

the most effective, and the difference in success rate between GID-RS and the other tested stochastic search algorithms is significant (p-values =0.00). Figure 5.17a graphically shows the average success rate for each faulty program, and Figure 5.17b and Figure 5.17c show the success rate distributions for each stochastic search algorithm using the raw data and data transformed using the *sqrt* function, respectively.

5.5.4.2. *RQ2: APR Repair Correctness.* We compared repair correctness by measuring the PFR and APFT. PFR indicates the relative number of failed potential repairs. A failed potential repair is one that fails at least one regression test (lower PFR is better). We excluded from this evaluation the faulty versions that had no potential repairs (success rate = 0). The average PFR for GA is 35%. The average PFR for GAWoCross and RS is 39.2% and 61.1%, respectively (lower PFR is better). Thus, the use of the selection algorithm and crossover operator in the genetic algorithm guides the search toward the validated repair, which produced more validated repairs; in other words, it produced fewer failing potential repairs. The difference between GA and GAWoCross is not significant (p-value = 0.363), but the difference between GA and RS is significant (p-value = 0.001) at the 0.95 confidence level. Thus, GA and GAWocross generated more validated repairs compared to RS.

Then, we compared the average PFR of the guided search algorithms. The average PFR for GID-GA is 42.3% , and the average PFR for GID-GAWoCross and GID-RS is 43.8% and 38.9%. The GID-RS produced more validated repairs (61.1%) compared to the other tested guided stochastic search algorithms, and the difference between GID-RS and other guided search algorithms (GID-GA and GID-GAWocross) is significant (p-values = 0.01 and 0.00; respectively). We also measured the statistical difference between GID-RS and GA since GA is the best original algorithm, and GID-RS is the best guided algorithm; the difference is not significant (p-value = 0.359) at the 0.95 confidence level. Lastly, we compared the average PFR of exhaustive search algorithms to those of stochastic search algorithms. BF generated an average of 43.9% failing potential repairs, and Ordered-BF generated an average of 36.5% failing potential repairs. The average PFR for Ordered-BF is between the average PFR for GA and GID-RS.

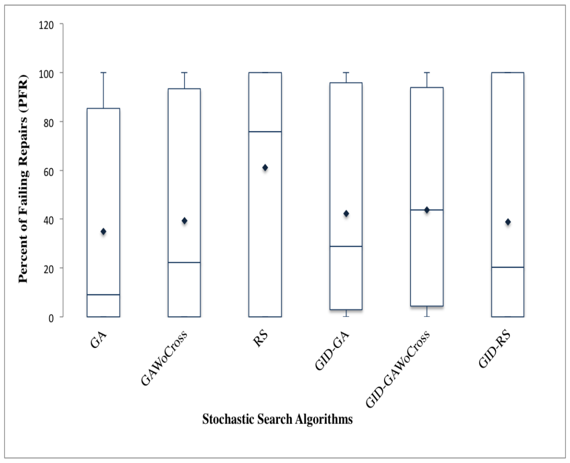
In summary, GA and GID-RS generated more validated repairs compared to all other tested search algorithms. The differences between GA and the other search algorithms, except RS, are not significant, but the difference between GID-RS and all the other search algorithms, except GA and GAWoCross, is significant. Figures 5.18a and 5.18b show the PFR distributions for stochastic search algorithms; we did not include exhaustive search algorithms because PFR measures the number of failing repairs for the total number of generated potential repairs when the algorithm runs for 100 time. The exhaustive search algorithms runs once and it either finds a repair or not.

For each failing potential repair, we measured the average percent of failing regression tests (APFT) (lower APFT is better). This measure estimates how far a failing potential repair is from being a valid repair. We included all faulty versions with at least one failing potential repair. APFT values for each search algorithm are shown in Table 5.12 and Table 5.13. The exhaustive search algorithms generated potential repairs that failed fewer regression tests (the mean APFT for BF is 2.4% and the mean APFT for Ordered-BF is 1.5%). To measure the statistical difference, we applied the Mann-Whitney U Test (Table 5.14). The difference between Ordered-FB and GA and GID-GA is significant at the 0.95 confidence level, and the difference between Ordered-BF and RS and GID-GAWoCross is significant at the 0.80 confidence level. However, the difference between Ordered-BF, BF, GID-RS and GAWoCross is not significant . Thus, we conclude that using the BF, Ordered-BF, GID-RS, or GAWoCross algorithms produced potential repairs that failed fewer regression tests (low APFT) than GA, RS, GID-GA, and GID-GAWoCross. Figure 5.18c and Figure 5.18d show the APFT distributions for each search algorithm.

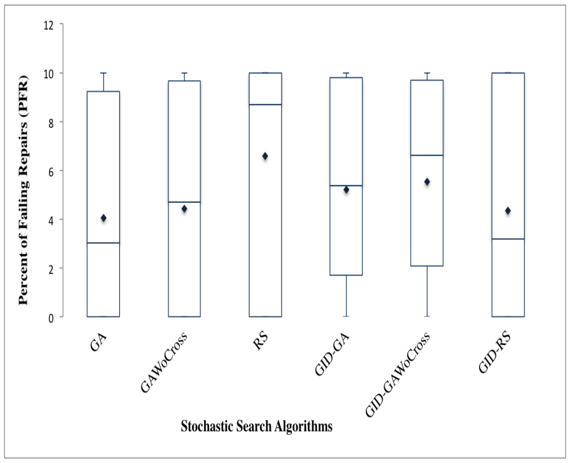
5.5.4.3. *RQ3: APR Performance.* To assess APR performance, we compared the ANGV and the ATotalTime required to fix faults for *N* repairs for each faulty version (lower ANGV

Search Algorithms	APFT P-Values
Ordered-BF and BF	0.430
Ordered-BF and GA	0.002
Ordered-BF and GAWoCross	0.770
Ordered-BF and RS	0.137
Ordered-BF and GID-RS	0.676
Ordered-BF and GID-GA	0.001
Ordered-BF and GID-GAWocross	0.157

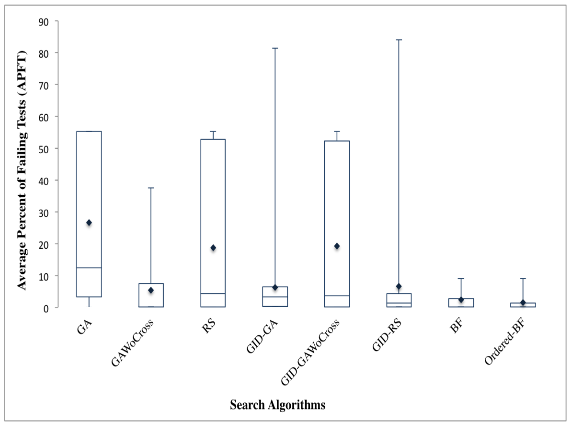
Table 5.14. P-values of applying the Mann-Whitney U Test for APFT of different search algorithms at 0.95 confidence level.



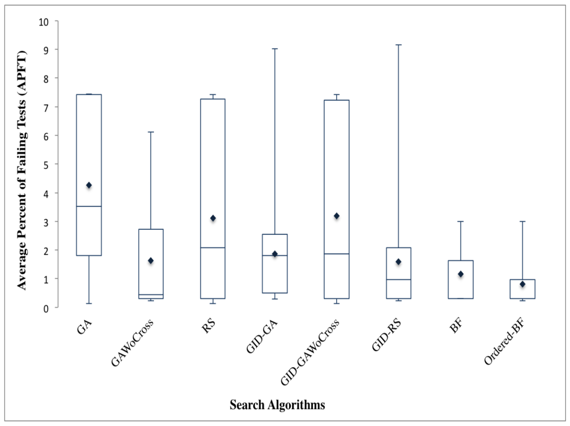
(A) Raw PFR Data



(B) Transformed PFR Data



(C) Raw APFT Data



(D) Transformed APFT Data

Figure 5.18. MUT-APR repair correctness when applying different search algorithms. Lower PFR and APFT is better.

and ATotalTime is better). We excluded faulty versions with success rate = 0. We first examined the ANGV values. GA generated an average of 491.5 variants until a potential repair is found, and GAWoCross and RS generated an average of 588.7 and 378.7 variants, respectively. Thus, the RS algorithm generated fewer variants before finding potential repairs compared to GA and GAWocross. Our results confirm those found by Qi et al. [87, 31]; for 23 out of 24 faults, random search required fewer variants to find potential repairs.

Then, we compared APR performance of the guided search algorithm. The ANGV for GID-RS is 63.9, while the ANGV for GID-GA and GID-GAWoCross is 236.6 and 272 variants, respectively. The guided search algorithms generated fewer number of variants before finding a potential repair compared to the corresponding original algorithm (e.g., GA required an average of 491.5 variants to find a potential repair, while GID-GA required an average of 236.6 variants to find a potential repair). The difference between the guided search algorithm and its corresponding original algorithm is significant (p-value = 0.00) using the Wilcoxon Signed-Rank Test at the 0.95 confidence level.

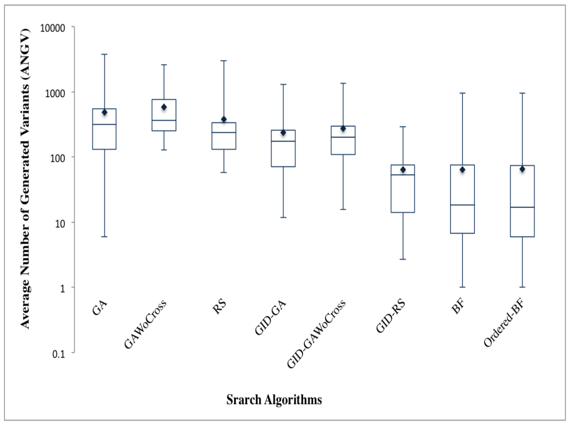
For exhaustive search algorithms, the ANGV for BF and Ordered-BF is 64.8 and 64.9 variants. Thus, GID-RS generated the fewest number of variants to find a potential repair compared to all other tested search algorithms. We analyzed the results by applying the Wilcoxon Signed-Rank Test at 0.95 confidence level. The difference between GID-RS and other stochastic search algorithms (GA, GAWocross, RS, GID-GA, and GID-GAWocross) is significant (p-values = 0.000); however, the difference between GID-RS and the two exhaustive search algorithms (BF and Ordered-BF) is not significant (p-values = 0.23 and 0.29, respectively). Thus, GID-RS , BF, and Ordered-BF algorithms improved APR performance by generating fewer variants to find potential repairs. Figure 5.19a and Figure 5.19b show the ANGV distributions for each search algorithm.

We compared APR performance when applying different search algorithms using the ATotalTime metric. GA and RS required the least time to fix faults; both took an average of 36.6 seconds, and GAWocross took an average of 51.6 seconds. We studied the statistical difference in ATotalTime between GA, GAWocross, and RS. The difference between GA and RS is not significant (p-value = 0.22), but the difference between GAWocross and the other two algorithms (GA and RS) is significant (p-values = 0.00) at the 0.95 confidence level.

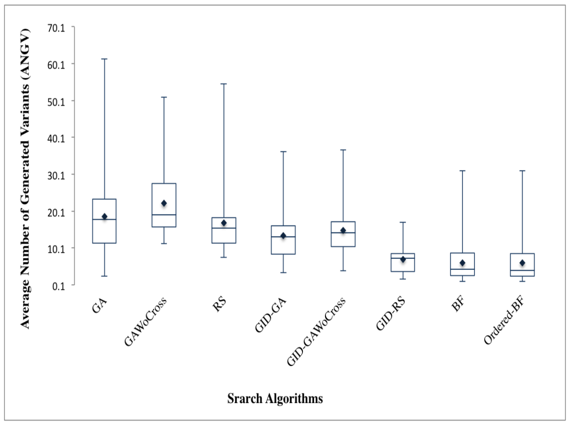
Then, we compared the ATotalTime for the guided search algorithms. GID-RS took an average of 38.3 seconds to find potential repairs, while GID-GA and GID-GAWocross took an average of 47.5 and 44.31 seconds, respectively. The difference between the three guided search algorithms is significant at the 0.95 confidence level.

In summary, GA and RS required the least time in average to find potential repairs. The statistical difference in ATotalTime between GA and all other search algorithms, except RS, is significant, and the statistical difference between RS and other search algorithms, except GA and BF, is significant at the 0.95 confidence level. However, one thing to note here is that the difference between the ATotalTime for GID-RS and both GA and RS is an average of 1.7 seconds (GID-RS took an average of 38.3 seconds to find potential repairs), and Ordered-BF search algorithm required the longest ATotalTime to find potential repairs was an average of 61.7 seconds. Figures 5.19c and 5.19d show the average ATotalTime distributions for each search algorithm.

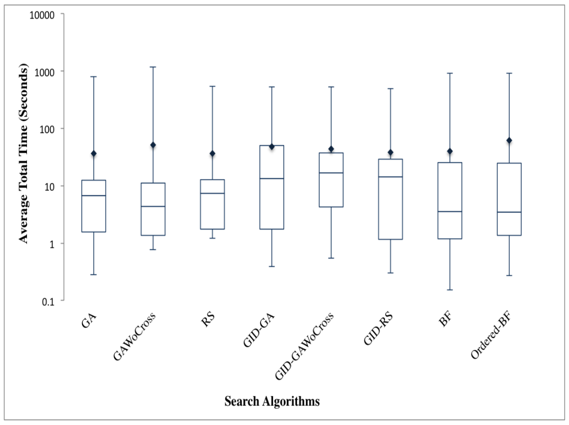
5.5.5. Conclusions and Lessons Learned. We found that guided random search (GID-RS) had the best success rate, and produced more validated repairs significantly compared to all other tested stochastic search algorithms except GA and GAWoCross. The Ordered-BF search algorithm generated potential repairs that failed fewer regression tests (APFT), but the difference between GID-RS, GAWocross, BF and Ordered-BF is not significant. In addition, GID-RS and the exhaustive search algorithms improved APR performance by generating fewer variants to find a potential repair compared to all other tested search



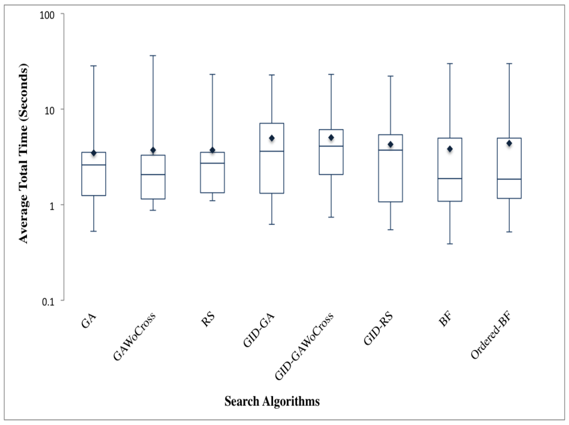
(A) Raw ANGV Data



(B) Transformed ANGV Data of



(C) Raw ATotalTime Data



(D) Transformed ATotalTime Data

FIGURE 5.19. MUT-APR performance when applying different search algorithms. Lower ANGV and ATotalTime is better.

algorithms. These results relate to the nature of GID-RS, BF, and Ordered-BF search algorithms. They create variants by adding a single change (PMO) to the original faulty program, and the generated variant is considered a potential repair, or another variant is created using the original faulty program. In addition, GID-RS search algorithm selects PMO from a small pool, thus there is a higher chance to select PMO that fixes faults, which in response generates more validated repairs.

On the other hand, GA and RS algorithms required the least average total time to fix faults compared to all other tested search algorithms. However, the different in ATotalTime between GA, RS, and GID-RS is an average of 1.7 seconds, and the least efficient algorithm, which is Ordered-BF, took an average of 61.7 seconds to find potential repairs. The conflicting conclusions using ANGV and ATotaltime metrics supports our observation in Section 5.4.5, where we found there is no correlation between the two metrics.

To conclude, GID-RS improved success rate, generated more validated repairs, produced potential repairs that failed fewer regression tests, and required fewer variants to find potential repairs. On the other hand, the exhaustive search algorithms is guaranteed to find potential repairs that fail fewer regression tests. However, the number of generated variants to find potential repairs with exhaustive search algorithms can increase as the program size increases. Thus, the use of GID-RS can be more efficient especially as the program size increases.

5.6. Limitations and Threats to Validity

Our results shows that focusing on specific fault types can be very effective in repairing those faults. Although MUT-APR is limited to fixing single operator faults, it can be easily adapted to fix additional fault types.

Although our results show that faulty operators can be automatically fixed by MUT-APR, there are threats to validity.

Internal Validity: All the tool parameters are set heuristically as done by Weimer et al. [7] and represent a threat to internal validity. To mitigate this threat we selected the parameters that improved the success rate and the repair time in previous work [43, 42]. The selection of faulty versions with operator faults can bias the apparent effectiveness of fault repair approaches. To reduce this bias we used a C mutation tool to seed faults; we selected many faulty versions of each subject program.

To study repair correctness, we selected the repair tests randomly from the set of tests that satisfy the test methods of interest to avoid biasing the results toward one test method. To study repair correctness of different repair test suite methods, we used MUT-APR which produced more validated repairs than GenProg, thus reducing the effect of the PMOs used on PFR and APFT, and ensuring that the values of PFR and APFT are due to the selected test method.

The impact of different FL techniques on the performance and effectiveness of automated program repair was studied using a brute-force APR to eliminate the randomness of the search algorithm used in a previous study [27], thus mitigating the threats to internal validity by removing the dependency between FL technique accuracy and other factors such as the randomness of search algorithms.

External Validity: External validity relates to the ability to generalize the results. An important external threat to our study is the use of small programs in Section 5.2 and Section 5.3. Four different C programs with different faulty versions were used. The results might not generalize to larger programs. To mitigate external threats, we selected different faulty versions in which faults were seeded in different statement types. We seeded faults with a C mutation tool to create additional benchmarks with different faulty operators, and we also used subject programs that contain code to correct faults.

To mitigate threats to external validity when we studied repair quality, we used a large numbers (thousands) of generated potential repairs, and to limit the threats to the external validity when different FL techniques and search algorithms were studied, our evaluation consists of large C programs (more than 14K LOC). Even though we included larger programs, they are still not as large as real-world programs. In addition, our results might not be generalized to other fault types such as real-wold bugs, or to programs from other domains.

Construct Validity: In our evaluations, we use a set of repair tests to fix faults. One threat to construct validity relates to the properties of repair test suites; repair test suites determine the number of generated repairs and their correctness. To mitigate this threat we used repair tests suites that satisfy branch coverage which decreased the introduction of new faults, as shown by our evaluation in Section 5.3.

To measure repair correctness, we identified two metrics: PFR and APFT. APFT metric is a threat to the construct validity, APFT does not directly measure the number of introduced new faults. Rather it identifies failures caused by introduced new faults. A single fault can cause multiple tests to fail. Also, the accuracy of PFR and APFT depends on the quality of the regression tests; different regression tests might produce different results. Thus, PFR and APFT are only estimates. The use of the distribution of changes and LOCC as surrogate measures for maintainability also represent threats to construct validity. These measures quantify important aspects of maintainability, but they are not complete and comprehensive maintainability indicators.

The accuracy of FL techniques depends in part on the repair tests used to identify faulty statements, which is a threat to construct validity. To mitigate this threat we used five different repair test suites. For the small programs, we selected repair test suites randomly from the set of suites provided by the SIR. For the large programs, we created independent repair test suites for each faulty version with no less than 20 passing test cases to achieve the best accuracy as reported in [21]. However, we did not use repair test suites that satisfy branch coverage due to the unavailability of different branch repair test suites that include at least one failing and one passing tests for each faulty version in the SIR repository.

Total time is a metric we used to measure APR performance, but time can be affected by other factors such as the number of repair tests, and the compilation and execution time of variants. In addition, time is platform dependency and it can be affected by the resources used to run the experiments. Thus, total time is another threat to the constant validity. To mitigate this threat, we ran all experiments using the same machine.

Conclusion Validity: Conclusion validity relates to the statistical significant of the results. We used the *sqrt* function to transform data to make it more normal. We applied the Shapiro-Wilk normality Test [88] to check the normality of the data. We studied the consistency of the results across all faulty versions, and measured the statistical difference between the variables by applying statistical tests. We applied Paired T-Test with normally distributed data, and non-parametric statistical tests— the Mixed Model, Wilcoxon Signed-Rank Test, and Mann-Whitney U Test—when the data is not normally distributed. To limit threats to conclusion validity, we studied the relation between variables using the same set of parameters to decrease the error variance. We also used the same evaluation settings such as faulty versions, repair tests, FL technique and search algorithm when studying the impact of APR components on effectiveness, repair quality, and performance. In addition, we ensured randomness in the experimental setting when selecting benchmarks and repair test suites.

CHAPTER 6

Conclusion

The overall goal in the research presented in this dissertation is to evaluate APR components and mechanisms to optimize APR effectiveness, repair quality, and performance. In this chapter we summarize our results based on a comprehensive evaluation, and we discuss the remaining many research opportunities to further improve APR techniques.

6.1. Results

We evaluated and compared options for APR components and mechanisms.

Program modification operators (PMOs): Simple PMOs that insert a syntactic change to the code (e.g., transform > into alternatives: ==, !=, >=, <, and <=) improve APR effectiveness, and the quality of potential repairs in fixing faulty operators compared to the GenProg APR approach that uses existing code (e.g., delete a statement or swap statements) to fix faults [80]: MUT-APR fixed 48.03% while GenProg fixed 31.48% of operator faults, and 72.64% of MUT-APR generated repairs are validated repairs, while 3.06% of GenProg generated repairs are validated repairs.

Repair test suites selection methods: We compared the impact of three test methods to select repair tests with the following results:

- Repair test suites that satisfy branch coverage test criteria improved the average success rate significantly (42.3%) compared to repair test suites that satisfy statement coverage test criteria (41.46%) and repair test suites that are randomly selected (37.92%).
- Branch coverage repair test suites improved repair quality significantly by generating fewer failing potential repairs (12.77%) compared to statement coverage repair test

suites (26.58%) and random repair test suites (24.04%), and branch coverage repair test suites generated potential repairs failed fewer regression tests (an average of 3.2% of regression tests), while statement coverage repair test suites failed an average of 4.9% of regression tests and random repair test suites failed an average of 4.8% of regression tests [80].

- Repair test suites that are randomly selected improved APR performance by decreasing the average number of generated variants (ANGV) to 176.5 and the average total time (8.23 seconds) compared to branch coverage repair test suites that required an average of 187.34 variants and 14.25 seconds, and statement coverage repair test suites that required an average of 186.71 variants and 11.87 seconds to find potential repairs; however, the difference in ANGV is not significant between the three test methods.

Repair test suites size: We evaluated the impact of small and large repair test suites, and we found the following:

- Using small repair test suites improved APR effectiveness significantly by increasing success rate (average success rate for small repairs test suites is 42.2% vs. 19% for large repair test suites). These results are consistent with those found by Nguyen et al. [4].
- APR repair quality and performance are improved significantly using small repair test suites. Small repair test suites generated more validated repairs (68.7% of potential repairs are validated repairs) compared to large repair test suites (10.9%), and they also generated potential repairs that failed fewer regression tests (3.5% vs. 13.7% APFT). Using small repair tests improved APR performance significantly by

generating fewer variants and required less time to find potential repairs compared to large repair test suites.

Fault localization techniques: Fault localization techniques compute a score to identify statements that are more likely to hide faults. The Optimal FL technique had the best APR performance by requiring less time (an average of 28.6 seconds) and generating fewer variants (an average of 34.4 variants) until finding potential repairs. However, APR is least effective when Optimal FL technique was used since it failed to identify actual faulty statement in two trials. The Ochiai fault localization technique improves APR effectiveness since Ochiai always identified actual faulty statements, and it assigned an equal or lower LPFS rank to actual faulty statements compared to other FL techniques. Our results are different that those found by Qi et al. [27, 28], which identified Jaccard as the best FL technique when used by GenProg.

Search algorithms: We compared the impact of stochastic and exhaustive search algorithms on APR, and we found the following:

- The guided random search algorithm (GID-RS) improved APR effectiveness (an average success rate of 83%) significantly compared to all other tested stochastic search algorithms. We suggest using GID-RS algorithm when a large number of PMOs are used to fix faults in large programs with APR techniques.
- The genetic algorithm (GA) and the guided random search algorithm (GID-RS) produced more validated repairs (an average of 65% and 61.1% of potential repairs are validated repairs for GA and GID-RS, respectively) compared to other tested stochastic search algorithms, but the ordered brute-force (Ordered-BF) generated potential repairs that failed fewer regression tests (an average of 1.5% of regression

tests are failed) compared to other tested search algorithms; however, the difference between Ordered-BF, BF, GID-RS, and GAWoCross is not significant.

- The guided random search algorithm (GID-RS) improved APR performance by decreasing the average number of generated variants (ANGV) to 63.9 compared to other tested search algorithms. The difference between GID-RS and other tested stochastic search algorithms is significant, but the difference between GID-RS and the two exhaustive search algorithms is not significant. Our results are similar to those Qi et al. [87] obtained by applying random search algorithm on GenProg. However, their RS algorithm is different than our GID-RS algorithm in that: (1) our GID-RS search runs for one generation to ensure each variant contains one change, while their RS search runs for multiple iterations, (2) our algorithm computes a fitness function to validate variants, while their algorithm runs variants on repair tests and discard variants as soon as one test fails, and (3) GID-RS selects a PMO from a small pool of operators that contains the alternatives of faulty operators, while their algorithm selects a PMO from a larger pool. On the other hand, GA and RS required the least time to find potential repairs; however, GID-RS required an average of two more seconds, and the maximum average time required to find potential repairs was 61.7 seconds.
- There is no advantage of using the ordered brute-force algorithm (Ordered-BF) since the difference in repair quality and performance between BF and Ordered BF is not significant. Other ordering heuristic should be applied and evaluated.

Our results suggest using simple program modification operators, small branch coverage repair test suites, Ochiai fault localization technique, and the guided random search algorithm to improve APR effectiveness, the quality of generated repairs, and APR performance when fixing faulty operators in C programs.

6.2. Publications

Publications resulting from this work:

- Fatmah Yousef Assiri and James M. Bieman,"An assessment of the quality of automated program operator repair," In Proceedings of the 2014 ICST Conference (ICST).
- Fatmah Yousef Assiri and James M. Bieman. "Toward Improved APR Techniques and Tools," The 2014 Rocky Mountain Celebration of Women in Computing (RM-CWiC). Best Poster Award.
- Fatmah Yousef Assiri and James M. Bieman. "The impact of search algorithms in automated program repair". To Appear in the 2015 International Conference on Soft Computing and Software Engineering (ScSe).

Other publication:

- Fatmah Yousef Assiri and James M. Bieman,"Improving Testability of OO programs through Design by Contracts," The 2010 Colorado Celebration of Women in Computing (CCWIC).

6.3. Future Work

Our work can be extended to improve APR effectiveness and performance by fixing more types of faults, and applying methods to decrease the required number of repair tests and PMOs applied to find valid repairs. In addition, methods can be applied to fix multiple faults. APR techniques can be also developed to fix faults in software for different application domain such as scientific software without oracles.

6.3.1. Program Modification Operators.

6.3.1.1. *Implement additional PMOs.* The ability of APR to fix faults depends in part on the set of PMOs used to change faulty programs to create variants. Our current work is limited to fixing relational operators, arithmetic operators, shift operators and bitwise operators. Additional PMOs can be implemented to target unary operators, as well as those injected by C mutation tools, such as transforming a relational operator into arithmetic operator. In addition, MUT-APR can be combined with GenProg, the state-of-the-art APR tool, PMOs to target different/additional fault types.

6.3.1.2. *Going beyond simple operator faults.* Fixing multiple faults requires applying more than one PMO for each variant. That process can be implemented by applying all possible combinations of single PMO in which each combination fixes a single fault. However, this approach can be infeasible since the number of PMO can increase exponentially as the number of PMOs increase. Another approach to fixing multiple faults is to apply PMOs with a search algorithm that runs for multiple iterations, or a search algorithm that applies a crossover operator to combine the changes from two variants into one. In addition, properties of repair tests suites need to be investigated when fixing multiple faults. For example, repair test suites must contain at least one failing test that executes each fault, and the passing and failing tests should not overlap. We demonstrated this idea using the tcas subject program. In a preliminary study, we injected two operator faults, and we created repair test suites following the provided criteria. MUT-APR successfully fixed both faults.

Focusing on specific fault types can be very effective in repairing those faults. In order to make this approach more effective, we need to identify a wide variety of fault types that occur in potentially faulty statements and apply appropriate PMOs for that type of faults.

6.3.1.3. *Prioritize PMOs.* The large number of PMOs used by many APR techniques can reduce APR performance. Prioritizing PMOs in order to apply the operator that is most likely to fix faults before other operators can potentially improve performance. To prioritize PMOs, scores can be assigned to each PMO by adapting the subsumtion relationship that was introduced by Ammann et al. [89]. A mutation operator subsumes another one if the test cases that kill the first one also will kill the second one. Thus, the first mutation operator is applied first.

6.3.2. Repair Tests.

6.3.2.1. *Determine the number of passing and failing repair tests.* We studied two properties of repair test suites: selection method and suite size. We found that small repair test suites are better when repairing faults compared to large repair test suites. More investigation is needed to determine the number of passing and failing tests that improve repair quality. We can first study the correlation between repair quality and the number of passing and failing tests in repair test suites. Then, we can investigate the impact of the number of passing/failing tests by increasing/decreasing the number of passing/failing tests and studying the quality of generated repairs by computing the percent of failing potential repairs (PFR) and the average percent of failing regression tests (APFT).

6.3.2.2. *Apply test prioritization techniques.* Repair tests suites guide the search toward potentially faulty locations, and protect other locations in the code providing program functionality from unwanted modification. There are ideas in the literature that propose to prioritize repair tests that we can implement within MUT-APR. Weimer et al. [46] executed failing tests before passing ones. As soon as a test fails, there is no need to execute the remaining tests. Qi et al. [45] proposed an algorithm that orders repair tests based on their effectiveness in detecting invalid variants. First, each repair test is assigned a weight. Negative repair tests instantiate with a higher weight (weight of one) than passing tests which instantiate with a weight of zero. Then, when each variant is executed against repair tests to determine if it is a potential repair or not, weights are adjusted. If a variant fails a passing test *t*, its weight is changed to *initial weight + 1.* Then, the repair tests are reorders based on their weight. At any given point, if one test fails, a variant is discarded and another variant is created.

Another idea that can be investigated is to assign different weights to failing tests. For example, assigning lower weights to failing tests executes non-essential functionality, and higher weight to failing tests executes essential functionality. Different weights can also be assigned to passing tests. Such an approach can address a current problem with the fitness function: it cannot distinguish between two variants that produce the same fitness values, even if one of them is closer to the actual repair [25].

6.3.3. List of Potentially Faulty Statements.

6.3.3.1. *Shrinking the* $|LPFS|$. To fix faults, APR modifies potentially faulty statements sequentially from the LPFS until a potential repair is found or a set of parameters reached a limit. When stochastic search algorithms were applied, the number of variants generated in each iteration (population size) was equal to the number of statements in the LPFS ($|LPFS|$). To improve APR performance, we suggest decreasing the number of generated variant to be equal to $|LPFS|/2$, since actual faulty statements were ranked in the top half of the LPFS in 78% of faulty versions.

6.3.4. Applications.

6.3.4.1. *Evaluate on real-world software/bugs.* Our evaluation was performed using small C programs and two larger programs from the SIR repository [50]. Additional evaluations need to be performed using real-world software/bugs. The evaluation should also include large programs that simulate those used in industry.

6.3.4.2. *Different application domain.* Existing APR techniques are applied on software with available test oracles. APR techniques can be extended to address other domains such as scientific software with no oracles. One way to fix faults in theses applications by adapting metamorphic testing (MT) [90, 91] to determine if a repair is found or not. MT used to create new test cases from the initial test cases using metamorphic relations (MRs). MRs determine the relation between input values and their outputs. For example, if the test input of a *sum* function is {1,3,2} and the output is 6. The *permutative* relation, which randomly permute the order of test inputs, identifies the relation between the input-output pair. Thus, an MR creates new test inputs {3,1,2} and the correct output for the new input can be determined using the initial output and the MR. In this case, the initial output and the follow-up outputs should be the same because permutating the order of test inputs does not change the output of the sum function.

Bibliography

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 65 –74, april 2010.

[3] V. Debroy and W. E. Wong, “Combining mutation and fault localization for automated program debugging,” *Journal of Systems and Software*, vol. 90, pp. 45–60, 2014.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 772–781, IEEE Press, 2013.

[5] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA ’10, (New York, NY, USA), pp. 61–72, ACM, 2010.

[6] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 802–811, IEEE Press, 2013.

[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, (Washington, DC, USA), pp. 364–374, IEEE Computer Society, 2009.

[8] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 87–102, ACM, 2009.

[9] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 782–791, IEEE Press, 2013.

[10] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, (New York, NY, USA), pp. 947–954, ACM, 2009.

[11] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, pp. 109–116, may 2010.

[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54 –72, 2012.

[13] A. Arcuri, "On the automation of fixing software bugs," in *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, (New York, NY, USA), pp. 1003–1006, ACM, 2008.

[14] C. Kern and J. Esparza, "Automatic error correction of Java programs," in *Proceedings of the 15th international conference on Formal methods for industrial critical systems*, FMICS'10, (Berlin, Heidelberg), pp. 67–81, Springer-Verlag, 2010.

[15] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, (New York, NY, USA), pp. 1427–1434, ACM, 2011.

[16] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 162 –168, june 2008.

[17] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *Evolutionary Computation, IEEE Transactions on*, vol. 15, no. 4, pp. 515–538, 2011.

[18] "Microsoft Zune affected by 'bug'." `http://news.bbc.co.uk/2/hi/technology/7806683.stm`, December 2008.

[19] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada*, pp. 71–75, Citeseer, 2001.

[20] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, pp. 467–477, ACM, 2002.

[21] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89–98, IEEE, 2007.

[22] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, (New York, NY, USA), pp. 177–187, ACM, 2012.

[23] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, pp. 1–23, 2013.

[24] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, pp. 929–948, Oct 2001.

[25] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, (New York, NY, USA), pp. 965–972, ACM, 2010.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, (New York, NY, USA), pp. 273–282, ACM, 2005.

[27] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, (New York, NY, USA), pp. 191–201, ACM, 2013.

[28] Y. Qi, X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Empirical effectiveness evaluation of spectra-based fault localization on automated program repair," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pp. 828–829, July 2013.

[29] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494 – 3514, 2011.

[30] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "Does genetic programming work well on automated program repair?," in *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on*, pp. 1875–1878, IEEE, 2013.

[31] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair.," in *ICSE*, pp. 254–265, 2014.

[32] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, pp. 833–839, 2001.

[33] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, pp. 11:1–11:61, Dec. 2012.

[34] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*, pp. 1–59, Springer, 2012.

[35] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 1–10, IEEE, 2011.

[36] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[37] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *Software Engineering, IEEE Transactions on*, vol. 31, pp. 511–526, 2005.

[38] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 11:1–11:32, Aug. 2011.

[39] G. Kaminski, P. Ammann, and J. Offutt, "Better predicate testing," in *Proceedings of the 6th International Workshop on Automation of Software Test*, pp. 57–63, ACM, 2011.

[40] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pp. 235–241, 2002.

[41] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, "Using execution paths to evolve software patches," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pp. 152–153, IEEE, 2009.

[42] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, (Piscataway, NJ, USA), pp. 3–13, IEEE Press, 2012.

[43] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, (New York, NY, USA), pp. 959–966, ACM, 2012.

[44] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, (New York, NY, USA), pp. 313–316, ACM, 2010.

[45] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 180–189, Sept 2013.

[46] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 356–366, IEEE, 2013.

[47] M. Martinez and M. Monperrus, "Mining repair actions for guiding automated program fixing," tech. rep., INRIA, Tech. Rep, 2012.

[48] K. Pan, S. Kim, and E. J. Whitehead Jr, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[49] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[50] "Software-artifact infrastructure repository." `http://sir.unl.edu/php/previewfiles.php`.

[51] W. Wong, J. Horgan, S. London, and A. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pp. 41–41, April 1995.

[52] B. Hofer and F. Wotawa, "Mutation-based spreadsheet debugging," in *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pp. 132–137, Nov 2013.

[53] "EUSES spreadsheets corpus." `https://dl.dropbox.com/u/38372651/Spreadsheets/EUSES_Spreadsheets.zip`.

[54] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[55] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, (New York, NY, USA), pp. 233–244, ACM, 2006.

[56] B. Elkarablieh and S. Khurshid, "Juzi," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pp. 855–858, IEEE, 2008.

[57] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 389–400, ACM, 2011.

[58] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *ACM Sigplan Notices*, vol. 44, pp. 25–36, ACM, 2009.

[59] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 299–309, IEEE Press, 2012.

[60] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[61] M. V. Iordache and P. J. Antsaklis, "Supervision based on place invariants: A survey," *Discrete Event Dynamic Systems*, vol. 16, no. 4, pp. 451–492, 2006.

[62] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 550–554, IEEE Computer Society, 2009.

[63] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *Computer*, vol. 42, no. 9, pp. 46–55, 2009.

[64] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pp. 282–291, IEEE, 2013.

[65] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 266–276, ACM, 2014.

[66] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, (New York, NY, USA), pp. 121–130, ACM, 2008.

[67] "National vulnerability database." `http://nvd.nist.org`.

[68] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 433–436, ACM, 2007.

[69] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[70] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pp. 143–151, IEEE, 1995.

[71] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *Software Engineering, IEEE Transactions on*, vol. 36, no. 4, pp. 528–545, 2010.

[72] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *ECOOP 2005-Object-Oriented Programming*, pp. 528–550, Springer, 2005.

[73] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings. 18th IEEE International Conference on Automated Software Engineering*, pp. 30–39, IEEE, 2003.

[74] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 34–44, IEEE, 2009.

[75] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 30–39, IEEE, 2003.

[76] D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[77] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[78] "CIL Intermediate Language." `http://kerneis.github.io/cil/`.

[79] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Systems*, vol. 9, no. 3, pp. 193–212, 1995.

[80] F. Y. Assiri and J. M. Bieman, "An assessment of the quality of automated program operator repair," in *Proceedings of the 2014 ICST Conference*, ICST '14, 2014.

[81] F. Y. Assiri and J. M. Bieman, "The impact of search algorithms in automated program repair." Submitted to the 2015 International Conference on Soft Computing and Software Enineering, (SeSe'15), 2015.

[82] M. E. Delamaro and J. C. Maldonado, "Proteum/im 2.0: An integrated mutation testing environment," in *Mutation testing for the new century*, pp. 91–101, Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[83] T. Cleophas and A. Zwinderman, "Non-parametric tests," in *Statistical Analysis of Clinical Data on a Pocket Calculator*, pp. 9–13, Springer Netherlands, 2011.

[84] "gcov: A test coverage program." `http://www.linuxcommand.org/man_pages/gcov1.html`.

[85] H. Brown and R. Prescott, *Applied mixed models in medicine.* John Wiley & Sons, 2006.

[86] P. E. McKnight and J. Najab, "Mann-whitney u test," *Corsini Encyclopedia of Psychology*, 2010.

[87] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair.," in *ICSE*, pp. 254–265, 2014.

[88] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, no. 3/4, pp. pp. 591–611, 1965.

[89] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pp. 21–30, IEEE, 2014.

[90] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," *Department of Computer Science, Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01*, 1998.

[91] T. Y. Chen, T. Tse, and Z. Quan Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, 2003.

APPENDIX A

Acronyms

Acronym	Description
AO	Arithmetic Operators.
AOR	Arithmetic Operator Replacement.
ANGV	Average Number of Generated Variants for N potential repairs for each faulty version; N is the number of potential repairs generated for 100 trials for each faulty version.
APFT	Average Percent of Failed regression Tests for N potential repairs for each faulty version; N is the number of potential repairs generated for 100 trials for each faulty version.
ATotalTime	Average Total Time for N potential repairs for each faulty version; N is the number of potential repairs generated for 100 trials for each faulty version.
APR	Automated Program Repair.
BF	Brute-Force.
BWO	BitWise Operators.
BWOR	BitWise Operator Replacement.
GA	Genetic Algorithm.
GAWoCross	Genetic Algorithm Without Crossover Operator.
GID-GA	Guided Genetic Algorithm.
GID-GAWoCross	Guided Genetic Algorithm Without Crossover Operator.
GID-RS	Guided Random Search Algorithm.
LPFS	List of Potentially Faulty Statements.
MUT-APR	Mutation-based Automated Program Repair.
LOCC	Lines of Code Changed.
Ordered-BF	Ordered Brute-Force.
PFR	Percent of Failed potential Repair for 100 trials for each faulty version.
PMO	Program Modification Operator.
RO	Relational Operators.
ROR	Relational Operator Replacement.
RS	Random Search.
SIR	Software-Artifacts Infrastructure Repository.
SO	Shift Operators.
SOR	Shift Operator Replacement.

Table A.1. Acronyms in alphabetical order

APPENDIX B

Program Modification Operators (PMOs)

PMO	Description
GtToGe	Change > operator into >=
GtToLt	Change > operator into <
GtToLe	Change > operator into <=
GtToEq	Change > operator into ==
GtToNe	Change > operator into !=
GeToGt	Change >= operator into >
GeToLt	Change >= operator into <
GeToLe	Change >= operator into <=
GeToEq	Change >= operator into ==
GeToNe	Change >= operator into !=
LtToLe	Change < operator into <=
LtToGt	Change < operator into >
LtToGe	Change < operator into >=
LtToEq	Change < operator into ==
LeToNe	Change < operator into !=
LeToLt	Change <= operator into <
LtToGt	Change <= operator into >
LtToGe	Change <= operator into >=
LtToEq	Change <= operator into ==
LtToNe	Change <= operator into !=
EqToNe	Change == operator into !=
EqToLt	Change == operator into <
EqToGt	Change == operator into >
EqToGe	Change == operator into >=
EqToLe	Change == operator into <=
NeToEq	Change != operator into ==
NeToLt	Change != operator into <
NeToGt	Change != operator into >
NeToGe	Change != operator into >=
NeToLe	Change != operator into <=

Table B.1. Relational Program Modification Operators.

PMO	Description
PlustToMinus	Change + operator into -
PlustToDiv	Change + operator into /
PlusToMult	Change + operator into *
PlusToMod	Change + operator into %
MinusToPlus	Change - operator into +
MinusToDiv	Change - operator into /
MinusToMult	Change - operator into *
MinusToMod	Change - operator into %
MultToPlus	Change * operator into +
MultToMinus	Change * operator into -
MultToDiv	Change * operator into /
MultToMod	Change * operator into %
DivToPlus	Change / operator into +
DivToMinus	Change / operator into -
DivToMult	Change / operator into *
DivToMod	Change / operator into %
ModToPlus	Change % operator into +
ModToMinus	Change % operator into -
ModToMult	Change % operator into *
ModToDiv	Change % operator into /

Table B.2. Arithmetic Program Modification Operators.

PMO	Description
BAndToBOr	Change & operator into ||
BAndToBXor	Change & operator into $\wedge$
BOrToBAnd	Change || operator into &
BOrToBXor	Change || operator into $\wedge$
BXorToBAnd	Change $\wedge$ operator into &
BXorToBOr	Change $\wedge$ operator into &

Table B.3. BitWise Program Modification Operators.

PMO	Description
BSRtToBSLt	Change >> operator into <<
BSLtToBSRt	Change << operator into >>

Table B.4. Shift Program Modification Operators

PMO	Description
LtToLe	Change < operator into <=
LtToNe	Change < operator into !=
GtToGe	Change > operator into >=
GtToNe	Change > operator into !=
LeToLt	Change <= operator into <
LeToEq	Change <= operator into ==
GeToGt	Change >= operator into >
GeToEq	Change >= operator into ==
EqToLe	Change == operator into <=
EqToGe	Change == operator into >=
NeToLt	Change != operator into <
NeToGt	Change != operator into >
LtToEq	Change < operator into ==
LtToGt	Change < operator into >
GtToEq	Change > operator into ==
GtToLt	Change > operator into <
LeToNe	Change <= operator into !=
LeToGe	Change <= operator into >=
GeToNe	Change >= operator into !=
GeToLe	Change >= operator into <=
EqToLt	Change == operator into <
EqToGt	Change == operator into >
NeToLe	Change != operator into <=
NeToGe	Change != operator into >=
LtToGe	Change < operator into >=
GtToLe	Change > operator into <=
LeToGt	Change <= operator into >
GeToLt	Change >= operator into <
EqToNe	Change == operator into !=
NeToEq	Change != operator into ==

Table B.5. ROR Program Modification Operators in order based on the heuristic in Section 3.4.3.2