Preprint: published in Proc. IEEE Int. Conference on Software Testing, Verification, and Validation (ICST), April, 2014.

An Assessment of the Quality of Automated Program Operator Repair

Fatmah Yousef Assiri, James M. Bieman
Computer Science
Colorado State University
Fort Collins, USA
{*fatmahya,bieman*}*@cs.colostate.edu*

***Abstract*—Automated program repair (APR) techniques fix faults by repeatedly modifying suspicious code until a program passes a set of test cases. Although generating a repair is the goal of APR, a repair can have negative consequences. The quality of a repair is reduced when the repair introduces new faults and/or degrades maintainability by adding irrelevant but functionally benign code. We used two APR approaches to repair faulty binary operators: (1) find a repair in existing code by applying a genetic algorithm to replace suspicious code with other existing code as done by GenProg, and (2) mutate suspicious operators within a genetic algorithm. Mutating operators was clearly more effective in repairing faulty operators than using existing code for a repair. We also evaluated the approaches in terms of two potential negative effects: (1) the introduction of new faults and (2) a reduction of program maintainability. We found that repair processes that use tests that satisfy branch coverage reduce the number of new faults. In contrast, repair processes using tests that satisfy statement coverage and randomly generated tests introduce numerous new faults. We also demonstrate that a mutation-based repair process produces repairs that should be more maintainable compared to those produced using existing code.**

***Keywords*-automated program repair; repair maintainability; repair quality; test coverage**

I. Introduction

Producing and maintaining bug-free software generally requires time and labor-intensive debugging. The cost of testing, debugging, and verification have been estimated to be 50% to 70% of total development costs [1]. Automated approaches promise to reduce debugging cost.

Recent work has been directed towards automatic fault fixing. Automatic program repair (APR) techniques take a faulty program and a set of test inputs. APR techniques fix faults by modifying the faulty program until it passes the test inputs. Debroy and Wong [2] use mutation and fault localization technique to automate fault fixing using brute-force search. Wei et al. [3] fix faults in Eiffel programs equipped with contracts. Object states are derived for passing and failing runs, then the states are compared to study the abnormal behaviour. A behavioural object model is generated to change the object state from a failing state into a passing one. Parkins et al. fix faults without interrupting the execution by generating repairs as patches in deployed software [4], and Kim et al. [5] developed a tool to repair faults by using built-in patterns. Ten patterns were created based on common patches written by humans. Evolutionary computing and genetic programming were adapted to repair faults in C [6], [7], [8], [9], Java [10], [11], and Python [12] software, and was also used to help satisfy non-functional requirements [13], [14].

Of particular note is the GenProg tool, which uses genetic programming to modify a program until it finds a variant that passes all tests [6], [7], [8], [9]. It mutates the program to generate a variant, then the variant runs against the passing and failing tests to measure its fitness. GenProg can fix faults in C programs including infinite loops and segmentation faults. It also fixed 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 [15]. The GenProg approach is described in Section II.

Repair quality is one of the challenges of the automated repair process. Weimer et al. [7] describe repair quality as the ability of repairs to "compile, fix the defects, and avoid compromising required functionality." Perkins et al. [4] and Le Goues et al. [9] studied repair quality in terms of the introduction of new security vulnerabilities, and Jin et al. [16], and Liu et al. [17] studied repair quality in concurrent software in terms of the introduction of deadlock. We study repair quality in terms of repair correctness, and repair maintainability.

Repair correctness concerns how well a repaired program retains the required functionality without the introducing of new faults, while repair maintainability concerns how easy it is to understand and maintain the generated repair. We introduce different measures to indicate repair correctness and maintainability. Percent of failed repairs (PFR) and average percent of failed tests (PFT) are metrics for evaluating repair correctness. PFR is the percentage of repairs produced by APR that fail when tested on regression tests, and PFT is the average percent of regression tests that fail for individual repairs. To measure repair maintainability, we compute the size of repairs. The number of lines of code changed (LOCC) is the number of LOC modified, deleted, or added until a repair is found. We also analyze repairs to check the distribution of modifications. New code that is scattered may reduce software maintainability.

Le Goues et al. [9] found that "test suite selection is thus important to both scalability and correctness." Nguyen

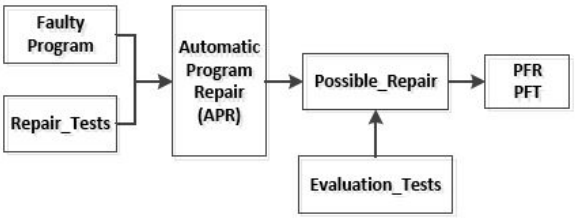

Figure 1. Steps to study repair correctness.

et al. [18] studied the effectiveness of an automated approach with different test sizes. They found that a large test suite decreases the success rate. Fast et al. [19] studied test suite sampling algorithms to improve fitness function performance, and found a sampling algorithm that improved performance of APR by 81%. Le Goues et al. [20] identified the need to assess and improve repair quality. We study repair correctness using different test suite coverage criteria as a step toward higher quality repairs without the need to use all regression tests.

Figure 1 describes steps that we use to study repair correctness. APR requires a set of test inputs to find a repair (*Repair_Tests*). We generate repair test inputs based on three test methods: branch coverage, statement coverage, and random test suites. We use a mutation-based repair process to fix faults when branch coverage, statement coverage, and random tests are used. To evaluate the repair correctness of different test suite selection criteria, we executed the generated repair (*Possible_Repair*) on a set of regression tests (*Evaluation_Tests*), and compute PFR and PFT.

Another important factor of repair quality is the nature of mutation operators used to fix faults. We compare the repair correctness and maintainability of two different APR techniques: (1) APR that depends on existing code to repair faults as done by GenProg, and (2) an APR that transforms faulty operators into their alternatives. The latter is similar to the approach of Debroy and Wong [2]. Debroy and Wong applied a brute-force search method to repair faults, while we combined the mutation of operators with genetic programming (GP). GP modifies the code by applying two operators: mutation and crossover. Mutation operators are injected to the code as done by Debroy and Wong [2], then a crossover operator is applied to combine information from the best selected variants.

We developed a prototype tool *MUT-APR* that injects mutation operators through genetic programming to repair faults in the subject programs. Our mutation operators can fix faulty operators in several different statement types: *if*, *return*, *assignment*, and *loop*. The MUT-APR approach is described in Section III.

Our focus on single faults is consistent with the competent programmer hypothesis that programmers "create programs that are close to being correct" [21]. We evaluated our work on Siemens Suite programs [22], and found that branch coverage repair test suites improve repair correctness compared to statement and random repair test suites. Repairs that are generated using tests that satisfy branch coverage have lower PFR and PFT. Unlike mutation-based APR, the use of existing code to fix faults generate repairs include many irrelevant changes to the code (an average of 28.68 LOCC) thus reducing maintainability. The evaluation is described in Section IV, and the threats to validity are described in Section V.

The main contributions of this paper are the following:

- An evaluation of the nature of mutation operators used to fix faults in terms of PFR and PFT to measure repair correctness, and LOCC to measure repair maintainability. The evaluation study shows that the mutation of operators produce higher quality repairs than the use of existing code as done by GenProg.
- The prototype tool, MUT-APR, that fixes faulty operators by constructing new operators within a genetic algorithm. MUT-APR injects fifty-eight mutation operators to fix binary operator faults in C programs.
- An evaluation of the effectiveness of different test suite selection criteria on repair quality. Compared to the use of random tests and statement coverage tests, the use of branch coverage tests when repairing faults improves repair correctness. Our results show that the use of the branch coverage criteria gives lower PFR and PFT values.

II. The GenProg Approach

Genetic programming (GP) is an evolutionary computing method that evolves computer software. GenProg is a tool developed by Weimer and his colleagues [6], [7], [8], [9] that uses genetic programming to repair faults. Figure 2 describes the GenProg appraoch.

GenProg consists of two steps: fault localization and a genetic algorithm. It takes as input a C program and a set of test inputs (passing and failing tests). To locate faults, an instrumented version of a faulty program is executed against the test inputs. Statement IDs are recorded for statements that are executed by the passing and the failing tests to create a list of statements that are more likely to contain the faults. It creates a weighted path file, which is an ordered list of pairs of statements and corresponding weights. Each statement is assigned a weight based on its execution (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). The statements in the list are ordered based on their assigned weights, and the statement list is passed to the genetic algorithm for modification. Statements are selected sequentially from the list. Mutation and crossover operators modify the program creating a new variant.

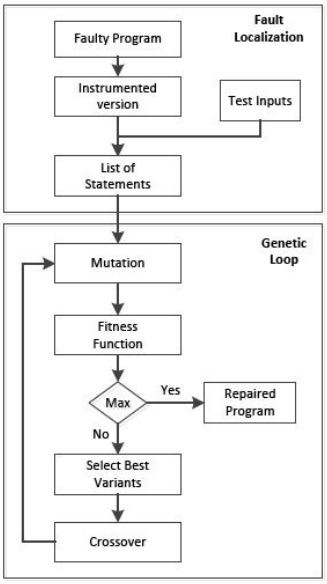

Figure 2. GenProg Approach.

Three mutation operators — *insert*, *delete* and *replace* statements — make use of similar code in the subject program. Variants are created by introducing one mutant. Each variant is executed against the tests to compute its fitness. Variants that do not compile or 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 operation combines information from two parent variants to create two new child variants. The genetic algorithm stops when a variant that maximizes the fitness function is found, or when the algorithm exceeds the upper bound of the predefined parameters.

III. Prototype Tool: MUT-APR

Our approach constructs new operators to replace faulty ones within a genetic programming algorithm. In order to fix a fault, potential faulty locations are identified, then the algorithm runs many times. Each run generates a new copy of the program with one change in an operator that is picked randomly. The new program *variant* is compiled and executed against the test inputs. If the variant passes all test inputs, it is deemed to be a possible repair. If not, the algorithm runs for more iterations to find a repair. We set an upper bound on the number of cycles that is equal to a specified maximum value.

A. Mutation Operator

Our mutation operators change each operator into its alternatives. We focus on fixing binary operator faults including relational operators, arithmetic operators, bitwise operators, and shift operators (Table I). The mutation operators are as follow: (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. We use a random selection approach by assigning an equal probability to all mutation operators.

Table I
Mutation operators supported by our approach.

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

Algorithm 1 is used in the implementation of our mutation operators. MUT-APR selects the first faulty statement from the weighted path, and selects a mutation operator 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 mutation operator (line 7) (e.g., *stmti* contains $>$, and the selected mutation is one of five operators that change the operator $>$ 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* in the WP (line 10), and a new variant is created (line 14).

Algorithm 1 Mutation Operator Pseudocode

```
1:  Inputs: Program P and weighted path WP
2:  Output: mutated program
3:  for all statements stmti in the weighted path do
4:    let mOp = choose(ChangeOp1ToOp2),
5:    if stmti contains an operator then
6:      let stmtOp = checkOperator(stmti)
7:      if stmtOp = Op1 then
8:        let stmtType = checkStmtType(stmti)
9:        let stmtj= apply(stmti,mOp)
10:       substitute stmti with stmtj in WP
11:     end if
12:   end if
13: end for
14: return P with stmti substituted by stmtj
```

B. The Repair Algorithm

To fix a fault, potential faulty locations are identified. We use the same fault localization method employed by GenProg. Then the tool modifies the program. A genetic programming algorithm mutates the program under test to generate an initial population (set of variants). A selection algorithm is used to select the best variants. Then a crossover operator combines information from the best variants to create a population for the next generation. A fitness function is computed for each variant until a repair that maximizes

the fitness function is found. Equal probability is assigned to all mutation operators and one of them is selected randomly.

Algorithm 2 describes how the genetic algorithm fixes faults. MUT-APR applies the set of mutation operators described in Table I. MUT-APR takes a faulty C program, a set of tests, a weighted path, and a set of parameters (population size, number of generations, and maximum fitness). The initial population is created by mutating the faulty statements (line 3). If no repair is found in the initial population, variants with zero fitness are discarded (line 6). If the number of remaining variants is less than half the population size $pop\_size/2$ (line 7), variants are duplicated (line 8) so that the number of variants is equal to a $pop\_size/2$ for use by the crossover operator (line 10-13). We applied the one-point crossover, which selects a random cut-off point, and swaps the parents statements after the selected point to create children variants. For all variants, a mutation operator is picked randomly (line 15) to create a new population (line 16). The process continues until a variant that maximizes the fitness is found (line 19), or until the process exceeds the upper bound set as a system parameter.

Algorithm 2 Genetic programming Pseudocode: *MUT-APR*

```
1: Inputs: Program P, max, pop_size
2: Output: variant
3: let pop = initial_pop(P,pop_size)
4: let fitness = ComputeFitness (pop)
5: repeat
6:    let variants = select (pop)
7:    if size(variants) < pop_size/2  then
8:       let variants = double(variants)
9:    end if
10:   for all two variants p1 and p2 ∈ variants do
11:      let newVariants(c1,c2) = crossover(p1,p2)
12:      let newPop = c1, c2, p1, p2
13:   end for
14:   for all variant in newPop do
15:      let mutOp= choose(mutationOperator)
16:      let pop = apply(variant,mutOp)
17:      let fitness = ComputeFitness (pop)
18:   end for
19: until fitness = max
20: return variant
```

C. Motivation example

We use a faulty Euclid's greatest common divisor adapted from an example used by Weimer et al. [7] to illustrate 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 1.

The *gcd* program in Figure 3 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 1). In order to fix the fault, the operator in line 1 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. *gcd.c*

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

The 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 faulty locations plus two other statements. Since the statement in line 1 is the first statement, it is selected first for mutation. A mutation operator 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 mutation operator that changes $<$ to $>$ for line 1, a new variant is created by constructing a new operator for line 1. The variants are compiled and executed against the tests. This variant fails two test inputs: (0,55) and (55,0). Since the variant did not pass all tests, it is not a repair and the process continues. Another statement and mutation operator are selected. MUT-APR will successfully repair the fault in line 1 when it selects the correct mutation operator.

D. Implementation

Our implementation of MUT-APR was built by adapting the GenProg Version 1 framework, which is implemented in OCaml. Like GenProg, MUT-APR requires a faulty C program and a set of test inputs (passing and failing tests). We turned off the original GenProg mutation operators and inserted fifty-eight new mutation operators that represent all binary operator transformations. For each operator (e.g., $>$), we implemented an OCaml class for each alternative. Each class changes the operator into one of its alternatives (e.g., $>=$, $<$, $<=$, $==$, $!=$). Therefore, each class checks the type of the statement that contains the operator. If the statement type is supported by our tool, a new statement is constructed with a new operator.

Table II
BENCHMARK PROGRAMS. EACH *Program* IS AN ORIGINAL PROGRAM FROM THE SIEMENS SUITE [22]. *LOC* IS THE NUMBER OF LINES OF CODES. *#Faulty Versions* IS THE NUMBER OF FAULTY VERSIONS. *#Tests* IS THE NUMBER OF REGRESSION TESTS.

Program	LOC	# Faulty Versions	# Test
tcas	173	14	1608
replace	564	21	5542
schedule2	374	8	2650
tot_info	565	11	1052

A fitness function is computed for each variant to determine if the generated variant is a repair or not. It is computed by running all tests against the created variant, and the fitness value is cached.

MUT-APR changes only one statement in each iteration to create a new variant. If a variant is similar to a previously created variant, the fitness will not be recalculated. Thus, MUT-APR reduces the number of fitness evaluations during the process.

IV. EVALUATION

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

- **RQ1 Repair correctness of a test suite**: Does the selection of test suite type (branch coverage, statement coverage, and random test) affect repair correctness?
- **RQ2 Repair correctness of an APR**: Does the nature of mutation operators used to modify the program improve repair correctness?
- **RQ3 Repair maintainability of an APR**: Does the nature of mutation operators used to modify the program improve repair maintainability?

A. Evaluation Design

To validate our approach, we used subject programs from the Siemens Suites in the Software artifacts infrastructure repository [22]. Since the ability to fix faults depends on the type of mutation operators that are supported by the approach, we only included programs with operator faults. For example we excluded print-tokens from our study because the seeded fault can be fixed by inserting or deleting a statement. We found different versions of each program with the fault of interest. We created additional faulty versions for each subject program by using Proteum/IM 2.0 [23], which is a C mutation tool, to inject additional mutations. The faults in the subject programs are seeded in different statements types: *if* statements, *return* statements, *loops*, and *assignments*. Table II consists of the benchmarks and the number of faulty versions for each program after removing equivalent mutants. This evaluation only deals with binary operator faults, and it includes four subject programs with 54 faults.

We compared the results of using three test selection methods: branch coverage, statement coverage, and random testing. Branch coverage and random test suites are from the SIR repository [22]. *Gcov* [24] is used to create statement coverage test suites. Test suites that satisfy both branch and statement coverage reached between 90% to 100% of the branches/statements of the programs under test, and each test 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 II that include randomly generated test suites. We included 25 faulty versions in total, and we ran MUT-APR on the selected programs.

MUT-APR takes the same set of parameters as GenProg. We used the default parameters from Weimer et al. [7]. Since MUT-APR and GenProg are random techniques, we ran each tool 100 times to ensure that it will generate a repair in at least one execution for each subject program. In each execution, the genetic loop runs for 10 generations, and each generation consists of a population size of 40 versions. The weights for the passing and the failing tests are 1 and 10 respectively.

Although we built our tool by adapting GenProg version.1, we also studied the ability of the latest version of GenProg version.2 to fix the faulty operators. For a valid comparison, we ran GenProg.v2 100 times on each subject program, and each run consists of 10 generations and a population size of 40. We set parameters in GenProg.v2 using the values specified in Le Goues et al. [25], [26].

B. Evaluation Results

We analyzed our results to answer the three research questions. To assess repair correctness, we executed the generated repair on regression tests that are taken from repository [22]. If the repair failed at least one regression test, that means the generated repair introduced new faults. To study repair maintainability, we compare the changes made by the mutation-based APR technique (MUT-APR) and the APR that uses existing code (GenProg) to fix faults.

As expected MUT-APR successfully repaired the faulty operators in 47 out of 54 (87.03%) of the programs. GenProg fixes many faults that MUT-APR cannot fix. However, GenProg repaired only 17 (31.48%) of the faulty operators (we excluded the faulty versions that GenProg could not repair). 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 they 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. We included 54 faulty programs in our study; however, only 17 of them were fixed by both techniques.

Table III shows the number of repairs found for each benchmark by running each faulty program on both MUT-APR and GenProg. MUT-APR was able to repair more

Table III
MUT-APR VS. GENPROG: NUMBER OF OPERATOR FAULTS FIXED FOR EACH SUBJECT PROGRAM.

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

operator faults than GenProg for each subject program. The number and the types of faults fixed depend on the mutation operators supported by the tool. Since this is an initial study of repair quality, we limit our tool to support a subset of mutation operators as shown in Table I. However, we plan to support more mutations (unary operators, and constants) to fix more faults.

We studied why MUT-APR 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 list of faulty statements and found that the faults were in statements that did not appear in the weighted path, and thus, were not treated as potentially faulty. A better fault localization technique would improve the effectiveness of our approach; we leave this for future work.

1) Repair Correctness depends on test suite type: Test coverage criteria specify the test requirements (e.g. branch coverage) that need to be satisfied during testing [27]. Coverage criteria can reduce testing cost by requiring an adequate set of tests that cover most parts of the program under test without the need to generate a large number of test inputs. We examine the use of the branch coverage and statement coverage criteria to select a small set of test inputs to generate higher quality repairs without the need to use all regression tests. Branch coverage test criteria require testing all feasible edges, and statement coverage test criteria require testing all feasible nodes in a control flow graph of the program under test.

To study repair correctness for repairs that are generated by the mutation-based approach (MUT-APR), we follow the steps described by Figure 1. We execute all repairs (*Possible_Repair*) that are generated using branch coverage (*Test_Repair1*), statement coverage (*Test_Repair2*), and random tests (*Test_Repair3*) on regression tests (*Evaluation_Tests*). Then we compute the percentage of failed repairs (PFR) for each subject program, and the average percent of failed regression tests (PFT) for individual repair. This evaluation includes 765 repairs that are generated using branch coverage test suites, 1234 repairs that are generated using statement coverage test suites, and 986 repairs that are generated using random tests (in total 2985 repairs).

Table IV summarizes the results of repairs that are generated using different test suite types. We found that an average of 12.77% of repairs that are generated using branch coverage test suites failed regression tests, while 26.58% and 24.04% of repairs that are generated using statement coverage and random test suites failed regression tests, respectively. For 38% of repaired faults, branch coverage test suites generate repairs with lower PFR than statement coverage test suites, and for 57% of repaired faults, branch coverage test suites generate repairs with lower PFR than random tests. For 50% of repaired faults, branch coverage test suites generate repairs that did not fail any regression test (with zero PFR). The largest PFR value using branch coverage test suites to produce repairs is 50%, while statement coverage and random test suites generate repairs that failed all regression tests (with PFR equal to 100).

Table IV
RESULTS SUMMARY OF REPAIRS THAT ARE GENERATED BY BRANCH COVERAGE, STATEMENT COVERAGE, AND RANDOM TEST SUITES ON BENCHMARKS.

	Branch Coverage	Statement Coverage	Random Tests
PFR mean	12.77	26.58	24.04
PFR std	18.32	31.94	28.46
PFT mean	3.2	4.9	4.8

Repairs that are generated using branch coverage failed in an average of 3.2% of regression tests, while repairs generated using statement coverage failed in an average of 4.9% of regression tests, and repairs generated using random test suites failed in an average of 4.8% of regression tests.

We applied Paired T-Test [28] to analyze the improvements of repair correctness when different test suite types were used. The difference is statistically significant between branch coverage and statement coverage (p-value = 0.03), and between branch coverage and random tests (p-value = 0.04) at the 0.95 confidence level. However, the difference is not statistically significant between statement coverage and random tests.

These results indicate that repairing faults can introduce new faults when the selected tests do not provide good coverage. Using branch coverage tests in the repair process can improve the quality of generated repairs significantly by generating more correct repairs (reduce PFR and PFT) compared to statement coverage and random tests.

2) Repair Correctness depends on the nature of mutation operators: To compare repairs correctness by two APR techniques, we only include subject programs that have a repair by both the mutation-based approach (MUT-APR) and the use of existing code to repair faults (GenProg). We include 648 repairs that are generated by MUT-APR, and 475 repairs that are generated by GenProg.

To determine which technique produces more correct repairs, we executed the generated repairs on a set of

regression tests, and computed the PFR and PFT. As shown in Figure 4, the PFR for repairs that are generated by MUT-APR is less than the PFR of repairs that are generated by GenProg. For ten out of 17 programs, all GenProg repairs failed regression tests.

We found that, for all the benchmarks, 27.36% of repairs that are generated by the MUT-APR technique failed regression tests (PFR=27.36%), while 96.94% repairs that are generated by GenProg failed regression tests (PFR=96.94%). We studied the difference between MUT-APR and GenProg using Paired T-Test. The PFR difference is statistically significant (*p-value* = 0.00) for all benchmarks at the 0.95 confidence level.

We also computed the percent of failed tests for each repair. Repairs that are generated by MUT-APR failed in an average of 6.65% of regression tests (PFT=6.65%), while repairs that are generated by GenProg technique failed in an average of 19.05% of the tests (PFT=19.05%).

In summary, repairs of operator faults produced by mutations (MUT-APR) tend to be correct more often than those produced by the use of existing code (GenProg).

3) Repair Maintainability: A study by Fry et al. [29] compares the software maintainability of human-written and machine-generated patches. They found that machine-generated patches reduce software maintainability by making the program less understandable. Maintainability is measured in terms of acceptability [5] and readability [30]. Buse and Weimer [31] found a correlation between software readability and its quality. Le Goues et al. [20] identify the need to develop more 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 [30]. We define two metrics to estimate repair maintainability. First, we define a static code measure for repair maintainability based on the size of a 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. A wider distribution of repair modifications can have a negative impact on software maintainability.

To measure repair maintainability we compare the modifications made by the mutation-based repair technique (MUT-APR) and 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 [31].

Since we only modify an operator, our repair minimizes the changes inserted into the code which makes it similar to repairs done by humans, and should not reduce software maintainability. In contrast, the use of existing code to repair faults makes many irrelevant changes to the code which reduce maintainability [25]. When GenProg generates a repair, it 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 was repaired by both MUT-APR and GenProg. MUT-APR changed one statement as shown in Figure 5. On the other hand, GenProg fixed the faults by making three changes (Figure 6): (1) it added *a-=b* after declaring variable *a* (line 3), (2) it added an empty *else* block after the faulty *if* statement (line 9 and 10), and (3) it copied the whole *if* block after statement *a-=b* (lines 14-18).

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

Figure 5. MUT-APR repair for the fault in Figure 3

```
1. void gcd (int a, int b) {
2. { a = (double )tmp;
3.  a -= b; } //inserted a -= b
4. b = (double )tmp_0;
5. if (a < (double )0) {
6.   printf("%g\n", b);
7.   return (0);
8.   }
9. else {  //inserted empty block
10.    }
11. while (b != (double )0)
12.  { if (a > b)
13.   { a -= b;
14.     if (a > b)  //inserted if block
15.       a -= b;
16.     else
17.       b -= a;
18.   }
19.   else
20.     b -= a;
21. }
22. printf("%g\n", a);
23. return (0);}
```

Figure 6. GenProg repair for the fault in Figure 3

We compared LOCC when both MUT-APR and GenProg generated repairs. We used the Linux *diff* command to count the LOCC, and we checked the changes distributions. If changes are scattered in the code, the code will be harder to understand and read.

We compare the differences between the original code, and repairs generated by MUT-APR and by GenProg. We include 648 repairs that are generated by the MUT-APR, and 475 repairs that are generated by GenProg for the faulty subject programs. MUT-APR repairs change an average of l.72 LOC in all subject programs, while the GenProg 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.

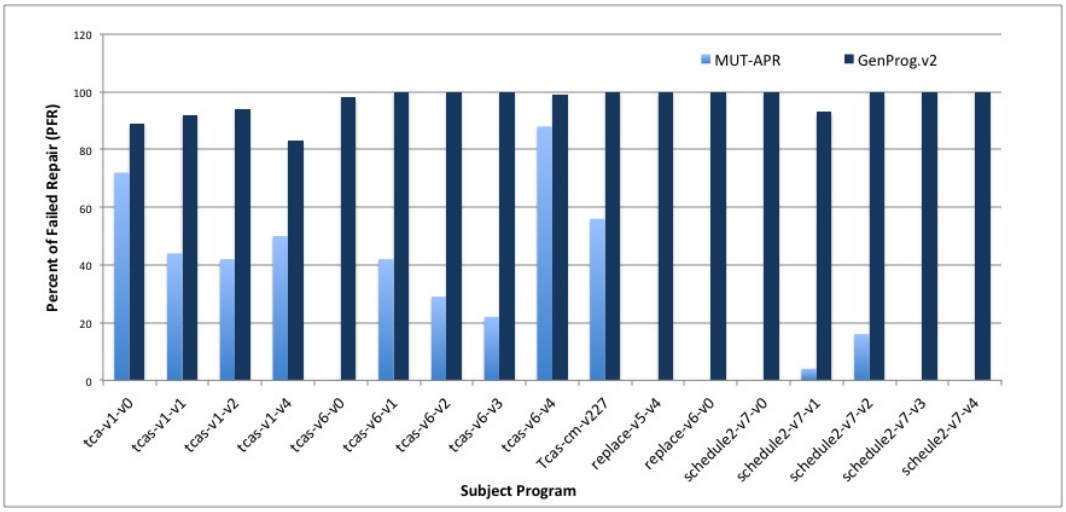

Figure 4. Percent of Failed Repairs (PFR) by mutation-based technique (MUT-APR) and the use of existing code (GenProg).

V. Limitations and Threats to Validity

Although our tool fixes faults in binary operators, we were not able to fix faults in Logical Operators due to MUT-APR's use of CIL [32]. CIL is an intermediate language for C programs. It is used to manipulate source code and generate a simplified version of source code. It transforms logical operators into *if-then* and *if-else* blocks. Fixing faults in logical operators w.r.t. CIL would require making changes to the generated *if* blocks rather than the operators. Thus to use CIL to fix logical operators, we need to change generated *if* blocks. An alternative is to use a different framework that does not depend on CIL. Then we can change logical operators as we do with other binary operators.

The proposed strategy fixed single faulty operators that are related to the supported mutation operators. However, MUT-APR is limited to fix faults requiring one line modification, and to repair faults in relational operators, arithmetic operators, shift operators, and bitwise operators in most statement types. We will expand the capabilities of MUT-APR by adding additional mutation operators to fix faults in other program constructs such as constants and unary operators. We also plan to add mutation operators to support the transformation between different binary operators (e. g., convert relational operators into arithmetic operators).

Although our results show that faulty operators can be automatically fixed, there are threats to 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 [26], [25]. 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. In addition, to improve internal validity we selected the test inputs randomly from the set of tests that satisfies the test methods of interest. An important external threat to our study is the use of small programs. 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 also seeded faults with a C mutation tool to create additional benchmarks with different faulty operators. To mitigate threats to external validity, we used large numbers (thousands) of generated repairs.

One threat to construct validity relates to the test set used to produce repairs by different APR techniques. The selection and properties of test sets determine the number of generated repairs and their correctness. To mitigate this threat we used test inputs that satisfy branch coverage which decreased the introduction of new faults. PFT 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 PFT depends on the quality of the regression tests. Thus, PFT is only an estimate. The use of the number 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.

Another threat is conclusion validity. We applied Paired T-Tests to study the relation between the variables. To limit this threat we studied the relation between variables using the same set of parameters to decrease the error variance. In addition, we ensured randomness in the experimental setting when selecting benchmarks and tests inputs.

VI. Related Work

Automated fault fixing has attracted considerable attention [33], [13], [4], [34], [3], [35], [2], [16], [36], [5]. Our work is based on GenProg, a tool developed by Weimer et al. [6], [7], [8], [9], which uses genetic programming to repair faults in C programs. GenProg modifies a program until it passes all tests using small test sets.

In 2012, Le Goues et al. [9] reported results showing that GenProg can fix additional faults including remote heap buffer overflow to inject code and conduct denial of services attacks. In further work, Le Goues et al. [25], [26] improved GenProg (version 2) to scale to larger programs by representing repairs as patches rather than modification to the AST. They also changed the mutation operators, introduced *fix localization* (a list of statements used as the source of the fix w.r.t 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. Fast et al. [19] defined a new fitness function, which evaluates each variant against a sample of the passing tests and all failing tests, to efficiently apply the fitness function using larger test suites. A variant that passes all the selected tests is tested against the entire test suite. GenProg fixes a variety of faults; however, the generated "repairs" failed other test inputs (other than the one used to generate repairs).

The *py*EDB tool [12] automates repairs of Python software. It modifies a program as patches rather than changing the whole program. Possible changes for a location are chosen from look-up tables that are created before the evolutionary process based on general rewrite rules that map each value to all possible modifications. The *py*EDB tool can fix faults in relational operators and constants. It differs from our approach in many ways: (1) *py*EDB selects mutations sequentially while we use random selection, (2) we fix faults in C programs while *py*EDB fixes faults in Python applications, (3) *py*EDB creates look-up tables to determine the possible changes for each program beforehand but we select a change randomly during execution, and (4) *py*EDB uses Tarantula [37] to locate the faults while we use the weighting scheme used by the original GenProg team [7]. The repairs that are generated pyEDB repairs can introduce new faults.

SemFix [18] is a tool for fixing faults through semantic analysis. Faulty statements are ranked using Tarantula [37]; highly ranked statements are selected first. Then, constraints are derived for statements using symbolic execution, and a repair is generated through program synthesis. The main different between their approach and ours is how they repair the faults. We apply a genetic algorithm while they use semantic analysis. SemFix successfully fixed faults in constants, mathematical operators, and relational operators in conditional statements and assignments. However, it does not fix relational faults in return statements and loops, and it does reduce APR performance.

Debroy and Wong [2] applied a brute-force search method to repair faults. Tarantula [37], computes a suspiciousness score for each statement and ranks them. First-order mutation operators are applied one by one starting with the higher rank statement to create a unique mutant. Each mutant is checked through string matching. If a mutant matches the original program, it is considered a "potential fix", then the potential fix is re-tested against all tests. This work supports a number of mutation operators, which include arithmetic, increment/decrements, and logical operator replacement. The evaluation used the Siemens Suite. Only 18% of the faults were fixed. No evaluation of repair quality was included.

Repair quality is one of the challenges of the APR process. To the best of our knowledge, none of the prior work identified quantitative measurements of repair quality. In addition, no prior study addressed the use of different coverage test inputs to generate repairs with higher quality.

VII. Conclusion

Quantitative measurements can compute repair correctness and maintainability of APR methods. We studied repair correctness when using branch coverage, statement coverage, and random test suites to generate a repair. We found that using branch coverage test suites improved the quality of generated repairs by reducing PFR and PFT values.

We then compared repair correctness and maintainability when using two automated repair techniques: use of existing code (GenProg) versus use of mutations (MUT-APR). We found that 27.36% of repairs that are generated by the mutation-based repair technique failed regression tests, while 96.94% repairs that are generated by using existing code failed regression tests.

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

We plan to identify additional measurements of repair quality, and use them to evaluate the repair quality of other automated repair techniques. We are studying techniques that can be adopted by automated program repair techniques to improve the quality of generated repairs.

Acknowledgment

This project was supported by the Ministry of Higher Education, Saudi Arabia. The authors would like to thank Westley Weimer and Claire Le Goues for their help with GenProg.

References

[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 *Proc. of the 3rd Intl. Conf. on Software Testing, Verification and Validation*, 2010, pp. 65 –74.

[3] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proc. of the 19th Int. Symp. on Software Testing and Analysis*, 2010, pp. 61–72.

[4] 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 *Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles*, 2009, pp. 87–102.

[5] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. of Int. Conf. on Software Eng.*, 2013, pp. 802–811.

[6] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. of the 11th Annual Conf. on Genetic and Evolutionary Computation*, 2009, pp. 947–954.

[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. of the 31st Int. Conf. on Software Eng.*, 2009, pp. 364–374.

[8] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54 –72, 2012.

[10] A. Arcuri, "On the automation of fixing software bugs," in *Companion of the 30th Int. Conf. on Software Eng.*, 2008, pp. 1003–1006.

[11] C. Kern and J. Esparza, "Automatic error correction of Java programs," in *Proc. of the 15th Int. Conf. on Formal Methods for Industrial Critical Systems*, 2010, pp. 67–81.

[12] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proc. of the 13th Annual Conf. on Genetic and Evolutionary Computation*, 2011, pp. 1427–1434.

[13] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. of the IEEE Congress on Evolutionary Computation*, 2008, pp. 162 –168.

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

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

[16] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2011, pp. 389–400.

[17] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. of the Int. Conf. on Software Eng.*, 2012, pp. 299–309.

[18] H. D. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proc. of the Int. Conf. on Software Eng.*, 2013, pp. 772–781.

[19] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proc. of the 12th Annual Conf. on Genetic and Evolutionary Computation*, 2010, pp. 965–972.

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

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

[22] Software-artifact infrastructure repository. [Online]. Available: http://sir.unl.edu/php/previewfiles.php

[23] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, "Proteum/im 2.0: An integrated mutation testing environment," in *Mutation Testing for The New Century*, 2001, pp. 91–101.

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

[25] 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 *Proc. of the Int. Conf. on Software Eng.*, 2012, pp. 3–13.

[26] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proc. of the 14th Int. Conf. on Genetic and Evolutionary Computation*, 2012, pp. 959–966.

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

[28] H. Hsu and P. A. Lachenbruch, "Paired t Test," *Wiley Encyclopedia of Clinical Trials*, 2008.

[29] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proc. of the Int. Symp. on Software Testing and Analysis*, 2012, pp. 177–187.

[30] K. Aggarwal, Y. Singh, and J. Chhabra, "An integrated measure of software maintainability," in *Proc. of Reliability and Maintainability Symp.*, 2002, pp. 235–241.

[31] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proc. of the Int. Symp. on Software Testing and Analysis*, 2008, pp. 121–130.

[32] CIL Intermediate Language. [Online]. Available: http://kerneis.github.io/cil/

[33] 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 *Proc. of the Int. Symp. on Software Testing and Analysis*, 2006, pp. 233–244.

[34] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proc. of the IEEE/ACM Int. Conf. on Automated Software Eng.*, 2009, pp. 550–554.

[35] J. L. Wilkerson and D. Tauritz, "Coevolutionary automated software correction," in *Proc. of the 12th Annual Conf. on Genetic and Evolutionary Computation*, 2010, pp. 1391–1392.

[36] R. Konighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *Formal Methods in Computer-Aided Design*, 2011, pp. 91–100.

[37] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Eng.*, 2005, pp. 273–282.