Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results

Westley Weimer
Computer Science Department
University of Virginia
Charlottesville, VA, USA
weimer@cs.virginia.edu

Zachary P. Fry
Computer Science Department
University of Virginia
Charlottesville, VA, USA
zpf5a@cs.virginia.edu

Stephanie Forrest
Computer Science Department
University of New Mexico
Albuquerque, NM, USA
forrest@cs.unm.edu

***Abstract*—Software bugs remain a compelling problem. Automated program repair is a promising approach for reducing cost, and many methods have recently demonstrated positive results. However, success on any particular bug is variable, as is the cost to find a repair. This paper focuses on generate-and-validate repair methods that enumerate candidate repairs and use test cases to define correct behavior. We formalize repair cost in terms of test executions, which dominate most test-based repair algorithms. Insights from this model lead to a novel deterministic repair algorithm that computes a patch quotient space with respect to an approximate semantic equivalence relation. This allows syntactic and dataflow analysis techniques to dramatically reduce the repair search space. Generate-and-validate program repair is shown to be a dual of mutation testing, suggesting several possible cross-fertilizations. Evaluating on 105 real-world bugs in programs totaling 5MLOC and involving 10,000 tests, our new algorithm requires an order-of-magnitude fewer test evaluations than the previous state-of-the-art and is over three times more efficient monetarily.**

***Index Terms*—Automated program repair; mutation testing; program equivalence; search-based software engineering**

I. Introduction

Both the cost of program defects and the cost of repairing and maintaining programs remain high. For example, a 2013 Cambridge University study places the global cost of general debugging at US$312 billion annually and finds that software developers spend 50% of their programming time “fixing bugs” or “making code work” [6]. In the security domain, a 2011 Symantec study estimates the global cost of cybercrime as US$114 billion annually, with a further US$274 billion in lost time [49], and a 2011 Consumer Reports study found that one-third of households had experienced a malicious software infection within the last year [11]. Human developers take 28 days, on average, to address security-critical defects [48], and new general defects are reported faster than developers can handle them [3]. This has driven bug finding and repair tools to take advantage of cheap, on-demand cloud computing [32], [27] to reduce costs and the burden on developers.

Since 2009, when *automated program repair* was demonstrated on real-world problems (ClearView [40], GenProg [54]), interest in the field has grown steadily, with multiple novel techniques proposed (AutoFix-E [52], AFix [24], Debroy and Wong [12], etc.) and an entire session at the 2013 International Conference on Software Engineering (SemFix [34], ARMOR [9], PAR [26], Coker and Hafiz [10]). We categorize program repair methods into two broad groups. Some methods use stochastic search or otherwise produce multiple candidate repairs and then validate them using test cases (e.g., GenProg, PAR, AutoFix-E, ClearView, Debroy and Wong, etc.). Others use techniques such as synthesis (e.g., SemFix) or constraint solving to produce a single patch that is correct by construction (e.g., AFix, etc.). We use the term *generate-and-validate program repair* to refer to any technique (often based on search-based software engineering) that generates multiple candidate patches and validates them through testing. Although generate-and-validate repair techniques have scaled to significant problems (e.g., millions of lines of code [27] or Mozilla Firefox [40]), many have only been examined experimentally, with few or no explanations about how difficult a defect or program will be to repair.

A recent example is GenProg, which takes as input a program, a test suite that encodes required behavior, and evidence of a bug (e.g., an additional test case that is currently failing). GenProg uses genetic programming (GP) heuristics to search for repairs, evaluating them using test suites. A *repair* is a patch, edit or mutation that, when applied to the original program, allows it to pass all test cases; a *candidate repair* is under consideration but not yet fully tested. The dominant cost of such generate-and-validate algorithms is validating candidate patches by running test cases [16].

In this paper, we provide a grounding of generate-and-validate automated repair, and use its insights to improve performance and consistency of the repair process. We first present a formal cost model, motivated in part by our categorization: broadly, the key costs relate to how many candidates are generated, and how expensive each one is to validate. This model suggests an improved algorithm for defining and searching the space of patches and the order in which tests are considered. Intuitively, our new algorithm avoids testing program variants that differ syntactically but are semantically equivalent. We define the set of candidate repairs as a quotient space (i.e., as equivalence classes) with respect to an approximate program equivalence relation, such as one based on syntactic or dataflow notions. Further optimizations are achieved by eliminating redundant or unnecessary testing. By recognizing that a single failed test rules out a candidate repair, our algorithm prioritizes the test most likely to fail (and the patch most likely to succeed) based on previous observations. The result is a deterministic, adaptive algorithm for automated program repair backed by a concrete cost model.

We also highlight a duality between generate-and-validate program repair and mutation testing [23], explicitly phrasing program repair as a search for a mutant that passes all tests. Examining the hypotheses associated with mutation testing sheds light on current issues and challenges in program repair, and it suggests which advances from the established field of mutation testing might be profitably applied to program repair.

Based on these insights, we describe a new algorithm and evaluate it empirically using a large dataset of real-world programs and bugs. We compare to GenProg as a baseline for program repair, finding that our approach reduces testing costs by an order of magnitude.

The main contributions of this paper are as follows:

- A detailed cost model for generate-and-validate program repair. The model accounts for the size of the fault space, size of the *fix space* (Section III), the order in which edits are considered (repair strategy), and the testing strategy.
- A technique for reducing the size of the fix space by computing the quotient space with respect to an approximate program equivalence relation. This approach uses syntactic and dataflow analysis approaches to reduce the search space and has not previously been applied to program repair.
- A novel, adaptive, and parallelizable algorithm for automated program repair. Unlike earlier stochastic repair methods, our algorithm is deterministic, updates its decision algorithm dynamically, and is easier to reason about.
- An empirical evaluation of the repair algorithm on 105 defects in programs totaling over five million lines of code and guarded by over ten-thousand test cases. We compare directly to GenProg, finding order-of-magnitude improvements in terms of test suite evaluations and over three times better dollar cost.
- A discussion of the duality between generate-and-validate program repair and mutation testing, formalizing the similarities and differences between these two problems. This provides a lens through which mutation testing advances can be viewed for use in program repair.

II. Motivation

Generate-and-validate repair algorithms have been dominated historically by the time spent executing test cases. Consider GenProg, which uses genetic programming (GP) to maintain a population of candidate repairs, iteratively mutating and recombining them in a manner focused by fault localization information, until one was found that passed all tests [27]. The iterative, population-based GP search heuristic makes it difficult to predict the cost (number of test cases evaluated) in advance, but early empirical estimates placed the percentage of effort devoted to running tests at roughly 60% [16], and later experiments with test-suite sampling to reduce the number of test-case evaluations improved performance by 80% [15].

An early and simplistic cost model for GenProg related the number of complete test suite evaluations to the size of the parts of the program implicated by fault localization [29, Fig. 9]. This is intuitive, but incomplete because it ignores the test-suite sampling discussed earlier, and it ignores the order in which candidate repairs are evaluated (e.g., if a high-probability candidate were validated early, the search could terminate immediately, reducing the incurred cost) and the number of possible repair operations (edits) that can be considered. However, GenProg demonstrates high variability, both across individual trials and among programs and defects. For example, in one large study, GenProg's measured ability to repair a defect varied from 0–100% with no clear explanation [28, Fig. 4]. In light of such results, a more powerful explanatory framework is desired.

A second cost arises from syntactically distinct but semantically equivalent program variants. This overhead is real [5], [35] but completely ignored by cost models that consider only test case evaluations. In a generate-and-validate repair framework, equivalent programs necessarily have equivalent behavior on test cases, so the size of this effect can be estimated by considering the number of candidate repairs that have exactly the same test case behavior. To this end, we examined the test output of over 500,000 program variants produced by GenProg in a bug repair experiment [27]. For a given bug, if we group variants based on their test output, 99% of them are redundant with respect to tested program behavior, on average. Although not all programs that test equally are semantically equivalent [44], this suggests the possibility of optimizing the search by recognizing and avoiding redundant evaluations.

We thus desire a more descriptive cost model as well as a search-based repair algorithm that explicitly considers program equivalence and the order in which tests and edits are explored.

III. Cost Model

This section outlines a cost model for generate-and-validate repair to guide the algorithmic improvements outlined in Section IV. We assume a repair algorithm that generates and validates candidate repairs using test cases, and tests are the dominant cost. We acknowledge many differences between approaches but believe the description in this section is sufficiently general to encompass techniques such as GenProg [27], ClearView [40], Debroy and Wong [12], and PAR [26].

Broadly, the costs in a generate-and-validate algorithm depend on generation (how many candidates are created) and validation (how each one is tested). Without optimization, the number of tests executed equals the number of candidate repairs considered by the algorithm times the size of the test suite. Fault localization identifies a region of code that is likely associated with the bug. Fault localization size refers to the number of statements, lines of code, or other representational unit manipulated by the repair algorithm. A candidate repair (i.e., a patch) typically modifies only the code identified by fault localization, but it can also include code imported from another part of the program [27], synthesized [34] or instantiated from a template [26], [40]. The number of first-order candidate repairs is the product of the fault localization size and the size of the *fix space* [27, Sec. V.B.3], where fix space refers to the atomic modifications that the algorithm can choose from. In addition, the repair algorithm could terminate when it finds and validates a repair, so the enumeration strategy—the order in which candidate repairs are considered—can have significant impact. Similarly, a non-repair may be ruled out the first time it fails a test case, and thus the testing strategy also has a significant impact.

Equation 1 shows our cost model. Fault localization size is denoted by Fault. The number of possible edits (mutations) is denoted by Fix. Note that the model structure has each component depending on the previous components. For example, Fix depends on Fault (e.g., some templates or edit actions might not apply depending on the variables in scope, control flow, etc.). The size of the test suite is denoted by Suite. The order in which the algorithm considers candidate repairs is $\mathsf{RepairStrat}$, and $\mathsf{RepairStratCost}$ denotes the number of tests evaluated by $\mathsf{RepairStrat}$, which ranges from $1/(\mathsf{Fault} \times \mathsf{Fix})$ (an optimal strategy that selects the correct repair on the first try) to 1 (a pessimal strategy that considers every candidate). Finally, given a candidate repair, the order in which test cases are presented is given by $\mathsf{TestStrat}$, and the number of tests evaluated by $\mathsf{TestStrat}$ is given by $\mathsf{TestStratCost}$, which ranges from $1/\mathsf{Suite}$ (optimal) to 1 (worst case).

$$\begin{aligned}\mathsf{Cost} \;&= \mathsf{Fault} \times \mathsf{Fix}(\mathsf{Fault}) \times \mathsf{Suite}(\mathsf{Fault}, \mathsf{Fix}) \\ &\times \mathsf{RepairStratCost}(\mathsf{Fault}, \mathsf{Fix}, \mathsf{Suite}) \\ &\times \mathsf{TestStratCost}(\mathsf{Fault}, \mathsf{Fix}, \mathsf{Suite}, \mathsf{RepairStratCost})\end{aligned} \tag{1}$$

By contrast, earlier algorithms defined the search space as the product of Fault and Fix for a given mutation type [28, Sec. 3.4]. GenProg's policy of copying existing program code instead of inventing new text corresponds to setting Fix equal to Fault (leveraging existing developer expertise and assuming that the program contains the seeds of its own repair) for $\mathcal{O}(N^2)$ edits, while other techniques craft repairs from lists of templates [40], [52], [26]. Although fault localization is well-established, *fix localization*, identifying code or templates to be used in a repair, is just beginning to receive attention [26].

In our model, Fix depends on Fault, capturing the possibility that operations can be avoided that produce ill-typed programs [39] or the insertion of dead code [5], [35]. Suite depends on Suite so the model can account for techniques such as impact analysis [42]. The $\mathsf{RepairStrat}$ term expresses the fact that the search heuristic ultimately considers candidate repairs in a particular order, and it suggests one way to measure optimality. It also exposes the inefficiencies of algorithms that re-evaluate semantically equivalent candidates. The $\mathsf{TestStrat}$ term depends on the repair strategy, allowing us to account for savings achieved by explicit reasoning about test suite sampling [16], [29]. Note that Suite optimizations remove a test from consideration entirely while $\mathsf{TestStrat}$ optimizations choose remaining tests in an advantageous order.

In the next section, we use the structure of the cost model, with a particular emphasis on the repair and test strategy terms, to outline a new search-based repair algorithm.

IV. Repair Algorithm

We introduce a novel automated program repair algorithm motivated by the cost model described in Section III. Based on the observation that running test cases on candidate repairs is time-consuming, the algorithm reduces this cost using several approaches. First, it uses an approximate program equivalence relation to identify candidate repairs that are semantically equivalent but syntactically distinct. Next, it controls the order in which candidate repairs are considered through an adaptive search strategy. A second adaptive search strategy presents test cases to candidate repairs intelligently, e.g., presenting test cases early that are most likely to fail. Although each of these components adds an upfront cost, our experimental results show that we achieve net gains in overall time performance through these optimizations. To highlight its use of *A*daptive search strategies and program *E*quivalence, we refer to this algorithm as "AE" in this paper.

We first describe the algorithm and then provide details on its most important features: the approximate program equivalence relation and two adaptive search strategies.

A. High-level Description

The high-level pseudocode for AE is given in Figure 1. It takes as input a program P, a test suite Suite that encodes all program requirements and impact analyses, a conservative approximate program equivalence relation $\sim$, an edit degree parameter k, an edit operator Edits that returns all programs resulting from the application of kth order edits, and the two adaptive search strategies $\mathsf{RepairStrat}$ and $\mathsf{TestStrat}$. The algorithm is shown enumerating all kth-order edits of P on line 3. In practice, this is infeasible for $k > 1$, and operations involving $\mathit{CandidateRepairs}$ should be performed using lazy evaluation and calculated on-demand. On line 5 the $\mathsf{RepairStrat}$ picks the candidate repair deemed most likely to pass all tests based on Model, the observations thus far. On line 8 and 9 the quotient space is computed lazily: A set of equivalence classes encountered thus far is maintained, and each new candidate repair is checked for equivalence against a representative of each class. If the new candidate is equivalent to a previous one, it is skipped. Otherwise, it is added to the set of equivalence classes (line 9). Candidates are evaluated on the relevant test cases (line 10) in an order determined by $\mathsf{TestStrat}$ (line 13), which uses information observed thus far to select the test deemed most likely to fail. Since successful repairs are run on all relevant tests regardless, $\mathsf{TestStrat}$ affects performance (opting out after the first failure) rather than functionality, and is thus chosen to short-circuit the loop (lines 12–17) as quickly as possibly for non-repairs. If all tests pass, that candidate is returned as the repair. If all semantically distinct candidates have been tested unsuccessfully, there is no k-degree repair given that program, approximate equivalence relation, test suite and set of mutation operators.

```
Input: Program $P$ : Prog
Input: Test suite Suite : Prog $\to \mathcal{P}$(Test)
Input: Equivalence relation $\sim$ : Prog $\times$ Prog $\to \mathcal{B}$
Input: Edit degree parameter $k : \mathcal{N}$
Input: Edit operator Edits : Prog $\times \mathcal{N} \to \mathcal{P}$(Prog)
Input: Repair strategy RepairStrat : $\mathcal{P}$(Prog) $\times$ Model $\to$ Prog
Input: Test strategy TestStrat : $\mathcal{P}$(Test) $\times$ Model $\to$ Test
Output: Program $P'$. $\forall t \in$ Suite($P'$). $P'(t) =$ true
 1: let $Model \leftarrow \emptyset$
 2: let $EquivClasses \leftarrow \emptyset$
 3: let $CandidateRepairs \leftarrow$ Edits($P, k$)
 4: repeat
 5:     let $P' \leftarrow$ RepairStrat($CandidateRepairs, Model$)
 6:     $CandidateRepairs \leftarrow CandidateRepairs \setminus \{P'\}$
 7:     // "Is any previously-tried repair equivalent to $P'$?"
 8:     if $\neg\exists$ $Previous \in EquivClasses$. $P' \sim Previous$ then
 9:         $EquivClasses \leftarrow EquivClasses \cup \{P'\}$
10:         let $TestsRemaining \leftarrow$ Suite($P'$)
11:         let $TestResult \leftarrow$ true
12:         repeat
13:             let $t \leftarrow$ TestStrat($TestsRemaining, Model$)
14:             $TestsRemaining \leftarrow TestsRemaining \setminus \{t\}$
15:             $TestResult \leftarrow P'(t)$
16:             $Model \leftarrow Model \cup \{\langle P', t, TestResult\rangle\}$
17:         until $TestsRemaining = \emptyset \ \vee \ \neg TestResult$
18:         if $TestResult$ then
19:             return $P'$
20:         end if
21:     end if
22: until $CandidateRepairs = \emptyset$
23: return "no $k$-degree repair"
```

Fig. 1. Pseudocode for adaptive equivalence ("AE") generate-and-validate program repair algorithm. Candidate repairs P' are considered in an order determined by RepairStrat, which depend on a $Model$ of observations ($Model$ may be updated while the algorithm is running) and returns the edit (mutation) deemed most likely to pass all tests. Candidate repairs are compared to previous candidates and evaluated only if they differ with respect to an approximate program equivalence relation ($\sim$). TestStrat determines the order in which tests are presented to P', returning the test on which P' is deemed most likely to fail. The first P' to pass all tests is returned.

The cost model identifies five important components: Fault, Fix, Suite, RepairStrat, and TestStrat. We leave fault localization (Fault) as an orthogonal concern [25], although there is some recent interest in fault localization targeting automated program repair rather than human developers [41]. In this paper, we use the same fault localization scheme as GenProg [54] to control for that factor in our experiments. Similarly, while we consider impact analysis [42] to be the primary Suite reduction, we do not perform any such analysis in this paper to admit a controlled comparison to GenProg, which also does not use any. Finally, one cost associated with testing is compiling candidate repairs; compilation costs are amortized by bundling multiple candidates into one executable, each selected by an environment variable [50]. In the rest of this section we discuss the other three components.

B. Determining Semantic Equivalence

To admit a direct, controlled comparison we form Edits as a quotient space of edits produced by the GenProg mutation operators "*delete* a potentially faulty statement" and "*insert* after a potentially faulty statement a statement from elsewhere in the program." This means that any changes to the search space of edits are attributable to our equivalence strategies, not to different atomic edits or templates. GenProg also includes a "*replace*" operator that we view as a second-degree edit (delete followed by insert); in this paper we use edit degree $k = 1$ unless otherwise noted (see Section V for a further examination of degree).

If two deterministic programs are semantically equivalent they will necessarily have the same test case behavior.[1] Thus, when we can determine that two different edits applied at the same fault location would yield equivalent programs, the algorithm considers only one. Since general program equivalence is undecideable, we use a sound approximation $\sim$: $A \sim B$ implies that A and B are semantically equivalent, but our algorithm is not guaranteed to find all such equivalences. We can hope to approximate this difficult problem because we are not dealing with arbitrary programs A and B, but instead A and A', where we constructed A' from A via a finite sequence of edits applied to certain locations. Although our algorithm is written so that the quotient space is computed lazily, for small values of k it can be more efficient to compute the quotient space eagerly (i.e., on line 3 of Figure 1).

In this domain, the cost of an imprecise approximation is simply the additional cost of considering redundant candidate repairs. This is in contrast with mutation testing, where the equivalent mutant problem can influence the quality of the result (via its influence on the mutation score, see Section VI). Drawing inspiration from such work, we determine semantic equivalence in three ways: syntactic equality, dead code elimination, and instruction scheduling.

a) Syntactic Equality: Programs often contain duplicated variable names or statements. In techniques like GenProg that use the existing program as the source of insertions, duplicate statements in the existing program yield duplicate insertions. For example, if the statement `x=0` appears k times in the program, GenProg might consider k separate edits, inserting each instance of `x=0` after every implicated fault location. Template-based approaches are similarly influenced: if `ptr` is both a local and a global variable and a null-check template is available, the template can be instantiated with either variable, leading to syntactically identical programs. Programs that are syntactically equal are also semantically equal, so $A =_{text} B \implies A \sim B$.

b) Dead Code Elimination: If `lval` is not live at a proposed point of insertion, then a write to it will have no effect on program execution (assuming `rval` has no side-effects [33]). If k edits $e_1 \ldots e_k$ applied to the program A yield a candidate repair $A[e_1 \ldots e_k]$ and e_i inserts dead code, then $A[e_1 \ldots e_k] \sim A[e_1 \ldots e_{i-1} e_{i+1} \ldots e_k]$. As a special common case, if e_1 inserts dead code then $A[e_1] \sim A$. Dataflow analysis allows us to determine liveness in polynomial time, thus ruling out insertions that will have no semantic effect.

[1]Excluding non-functional requirements, such as execution time or memory use. We view such non-functional program properties as a separate issue (i.e., compiler optimization).

c) Instruction Scheduling: Consider the program fragment `L1: x=1; L2: y=2; L3: z=3;` and the Fix mutation "insert `a=0;`". GenProg might consider three possible insertions: one at `L1`, one at `L2` and one at `L3`. In practice, all three insertions are equivalent: `a=0` does not share any read-write or write-write dependencies with any of those three statements. More generally, if `S1; S2;` and `S2; S1;` are semantically equivalent, only one of them need be validated. One type of instruction scheduling compiler optimization moves (or "bubbles") independent instructions past each other to mask latencies or otherwise improve performance. We use a similar approach to identify this class of equivalences quickly.

First, we calculate effect sets for the inserted code and the target code statements (e.g., reads and writes to variables, memory, system calls, etc.). If two adjacent instructions reference no common resources (or if both references are reads), reordering them produces a semantically equivalent program. If two collocated edits e and e' can be instruction scheduled past each other, then $A[\ldots ee' \ldots] \sim A[\ldots e'e \ldots]$ for all candidate repairs A. This analysis runs in polynomial time.

Precision in real applications typically requires a pointer alias analysis (e.g., must `*ptr=0` write to `lval` and/or may `*ptr` read from `lval`). For the experiments in this paper, we implement our flow-sensitive, intraprocedural analyses atop the alias and dataflow analysis framework in CIL [33].

C. Adaptive Search Strategies

The repair enumeration loop iterates until it has considered all atomic edits, stopping only when (and if) it finds one that passes all tests. Similarly, the test enumeration loop iterates through all the tests, terminating only when it finds a failing test or has successfully tested the entire set. In this subsection we discuss our algorithmic enhancements to short-circuit these loops, improving performance without changing semantics.

There are many possible strategies for minimizing the number of interactions in both loops. For the experiments in this paper we use a simple, non-adaptive RepairStrat as a control: as in GenProg, edits are preferred based on their fault localization suspiciousness value. By contrast, for TestStrat we favor the test that has the highest historical chance of failure (in the $Model$), breaking ties in favor of the number of times the test has failed and then in favor of minimizing the number of times it has passed. Although clearly a simplification, these selection strategies use information that is easy to measure empirically and are deterministic, eliminating algorithm-level variance.

Although these strategies are quite simple, they are surprisingly effective (Section V). However, we expect that future work will consider additional factors, e.g., the running time of different test cases, and could employ machine learning or evolutionary algorithms to tune the exact function.

V. EXPERIMENTS

We present experimental results evaluating the algorithm (referred to as *AE* to highlight its use of adaptive search and program equivalence) described in Section IV, using the ICSE 2012 dataset [27] as our basis of comparison. We focus on the following issues in our experiments:

1) Effectiveness at finding repairs: How many repairs from [27] are also found by AE?
2) Search-space efficiency: How many fewer edits are considered by AE than GenProg?
3) Cost comparison: What is the overall cost reduction (test case evaluations, monetary) of AE?
4) Optimality: How close is AE to an optimal search algorithm? (Section IV-C)?

A. Experimental Design

Our experiments are designed for direct comparison to previous GenProg results [27], and we use the publicly available benchmark programs from this earlier work. This data set contains 105 high-priority defects in eight programs totaling over 5MLOC and guarded by over 10,000 tests.

We provide grounded, reproducible measurements of time and monetary costs via Amazon's public cloud computing service. To control for changing prices, we report values using Aug-Sep 2011 prices [27, Sec. IV.C] unless otherwise noted. These GenProg results involved ten random trials run in parallel for at most 12 hours each (120 hours per bug). Since AE is deterministic, we evaluate it with a single run for each bug, allowing for up to 60 hours per bug.

B. Success Rates, Edit Order, Search-Space Size

Table I shows the results. k indicates the maximum possible number of allowed mutations (edits). Ignoring the crossover operator, a 10-generation run of GenProg could in principle produce an individual with up to 10 accumulated mutations. However, this is an extremely rare event, because of the small population sizes used in these results (40) and the effects of finite sampling combined with selection.

The "Defects Repaired" column shows that AE, with $k = 1$ and restricted to 60 CPU-hours, finds repairs for 53 of the 105 original defects. GenProg, with $k \leq 10$ and 120 CPU-hours, repairs 55. This confirms earlier GenProg results using minimization that show a high percentage (but not all) of the bugs that GenProg has repaired can also be repaired with one or two edits. As a baseline to show that this result is not specific to AE, we also consider a version of GenProg restricted to one generation ($k = 1$): it finds 37 repairs.

The remaining experiments include just the 45 defects that both algorithms repair, allowing direct comparison. The "Search Space" column measures the number of possible first-order edits. Higher-order edits are too numerous to count in practice in this domain: first-order insert operations alone are $\mathcal{O}(n^2)$ in the size of the program, and 10 inserts yields $\mathcal{O}(n^{20})$ options. The results show that using program equivalence (Section IV) dramatically reduces the search space by 88%, when compared with GenProg.

TABLE I

COMPARISON OF AE AND GENPROG ON SUCCESSFUL REPAIRS. AE denotes the "adaptive search, program equivalence" algorithm described in Figure 1 and columns labeled GP reproduce previously published GenProg results [27]. k denotes the maximum number of edits allowed in a repair. The first three columns characterize the benchmark set. The "Search Space" columns measure number of first-order edits considered by each method in the worst case. The "Defects Repaired" columns list the number of valid patches found: only 45 repairs are found in common by both algorithms (e.g., there are no shared repairs for **python**). The "Test Suite Evals." column measures the average number of test suite evaluations on those 45 repairs. The monetary cost column measures the average public cost of using Amazon's cloud computing infrastructure to to find those 45 repairs.

			Order 1 Search Space		**Defects Repaired**			**Test Suite Evals.**		**US$ (2011)**	
			AE	GP	AE	GP	GP	AE	GP	AE	GP
Program	LOC	Tests	$k=1$	$k=1$	$k=1$	$k=1$	$k\leq 10$	$k=1$	$k\leq 10$	$k=1$	$k\leq 10$
fbc	97,000	773	507	1568	1	0	1	1.7	1952.7	0.01	5.40
gmp	145,000	146	9090	40060	1	0	1	63.3	119.3	0.91	0.44
gzip	491,000	12	11741	98139	2	1	1	1.7	180.0	0.01	0.30
libtiff	77,000	78	18094	125328	17	13	17	3.0	28.5	0.03	0.03
lighttpd	62,000	295	15618	68856	4	3	5	11.1	60.9	0.03	0.04
php	1,046,000	8,471	26221	264999	22	18	28	1.1	12.5	0.14	0.39
python	407,000	355	—	—	2	1	1	—	—	–	—
wireshark	2,814,000	63	6663	53321	4	1	1	1.9	22.6	0.04	0.17
weighted sum	—	—	922,492	7,899,073	53	37	55	186.0	3252.7	4.40	14.78

C. Cost

Since neither algorithm is based on purely random selection, reducing the search space by x does not directly reduce the expected repair cost by x. We thus turn to two externally visible cost metrics: test suite evaluations and monetary cost.

Test suite evaluations measure algorithmic efficiency independent of systems programming or implementation details. The "Test Suite Evals." column shows that AE requires an order of magnitude fewer test suite evaluations than GenProg: 186 vs. 3252. Two factors contribute to this twenty-fold decrease: search-space reduction and test selection strategy (see Section V-D).

Finally, we ground our results in US dollars using public cloud computing. To avoid the effect of Amazon price reductions, we use the applicable rate from the earlier GenProg evaluation ($0.074 dollars per CPU-hour, including data and I/O costs). For example, on the **fbc** bug, serial AE algorithm runs for 0.14 hours and thus costs $0.14 \times 0.074 = 0.01$. GenProg runs ten machines in parallel, stopping when the first finds a repair after 7.29 hours, and thus costs $7.29.52 \times 10 \times 0.074 =$ 5.40. Overall, AE is cheaper than GenProg by a factor of three ($4.40 vs. $14.78 for the 45 repairs found by both algorithms).

D. Optimality

The dramatic decrease in the number of test suite evaluations performed by AE, and the associated performance improvements, can be investigated using our cost model. Our experiments used a simple repair strategy (fault localization) and a dynamically adjusted (adaptive) test strategy. In the cost model, TestStrat depends on RepairStrat: Given a candidate repair, the test strategy determines the next test to apply. For a successful repair, in which n candidate repairs are considered, an *optimal* test strategy would evaluate $(n-1)+|\text{Suite}|$ test cases. In the ideal case, the first $n-1$ candidate repairs would each be ruled out be a single test and the ultimate repair would be validated on the entire suite.

We now measure how close the technique described in Section IV approaches this optimal solution in practice. For example, for the 20 **php** shared repairs, GenProg runs 1,918,170 test cases to validate 700 candidate repairs. If the average test suite size is 7,671, an optimal test selection algorithm would run $680 + 20 \times 7,671 = 154,100$ test cases. GenProg's test selection strategy (random sampling for internal calculations followed by full evaluations for promising candidates [16]) is thus $12\times$ worse than optimal on those bugs. On those same bugs, AE runs 163,274 test cases to validate $3,099$ mutants. Its adaptive test selection strategy is thus very near optimal, with a $0.06\times$ increase in testing overhead on those bugs. By contrast, the naive repair selection strategy evaluated is worse than GenProg's tiered use of fault localization, mutation preference, fix localization, and past fitness values. Despite evaluating $4\times$ times as many candidates, however, we evaluate $12\times$ fewer tests. The results on other programs are similar.

This analysis suggests that integrating AE's test selection strategy with GenProg's repair selection strategy, or enhancing AE's adaptive repair strategy, could lead to even further improvements. We leave the exploration of these questions for future work.

The difference between our test count reduction and our monetary reduction stems from unequal test running times (e.g., AE selects tests with high explanatory power but also above-average running times).

E. Qualitative Evaluation

Since GenProg has a strict superset of AE's mutation operators, any repair AE finds that GenProg does not is attributable to Fault, Fix, RepairStrat or TestStrat. We examine one such case in detail, related to command-line argument orderings and standard input in **gzip**.[2]

An exhaustive evaluation of all first-order mutations finds that only 46 out of GenProg's 75905 candidates are valid

[2]http://lists.gnu.org/archive/html/bug-gzip/2008-10/msg00000.html

repairs (0.06%). Worse, the weightings from GenProg's repair strategy heuristics (which tier edit types after fault localization) are ineffective in this case, resulting in a 0.03% chance of selecting such an edit. GenProg considered over 650 mutants per hour, but failed to find a repair in time. By contrast, AE reduced the search space to 17655 via program equivalence and was able to evaluate over 5500 mutants per hour by careful test selection. However, AE's repair strategy was also relatively poor, considering 85% of possible candidates before finding a valid one. These results support our claims that while program repair could benefit from substantial improvements in fault localization and repair enumeration, our program equivalence and test strategies are effective in this domain.

VI. Duality with Mutation Testing

At a high level, mutation testing creates a number of *mutants* of the input program and measures the fraction that fail (are *killed* by) at least one test case (the *mutation adequacy score*). Ideally, a high score indicates a high-quality test suite, and a high-quality test suite gives confidence about program correctness. By contrast, low scores can provide guidance to iteratively improve programs and test suites. Mutation testing is a large field, and characterizing all of the variations and possible uses of mutation testing is beyond the scope of this work; we necessarily adopt a broad view of the field and do not claim that our generalization applies in all cases. The interested reader is referred to Jia and Harman [23], to whom our presentation here is indebted, for a thorough treatment.

Broadly, we identify the mutants in mutation testing with the candidates in program repair. This leads to duality between the mutant-testing relationship (ideally all mutants fail at least one test) and the repair-testing relationship (ideally at least one candidate passes all tests).

A. Hypotheses

The competent programmer hypothesis (CPH) [13] and the coupling effect hypothesis [38] from mutation testing are both relevant to program repair. The CPH states that programmers are competent, and although they may have delivered a program with known or unknown faults, all faults can be corrected by syntactic changes, and thus mutation testing need only consider mutants made from such changes [23, p. 3]. The program is assumed to have no known faults with respect to the tests under consideration ("before starting the mutation analysis, this test set needs to be successfully executed against the original program ... if p is incorrect, it has to be fixed before running other mutants" [23, p.5]). In contrast, program repair methods such as GenProg and AE assume that the program is buggy and fails at least one test on entry. GenProg and AE also assume the CPH. However, they often use operators that make tree-structured changes [54] (e.g., moving, deleting or rearranging large segments of code) or otherwise simulate how humans repair mistakes [26] (e.g., adding bounds checks) without introducing new ones. Search-based program repair further limits the set of mutants (candidate repairs) considered by using *fault localization*, program analysis that uses information from successful and failing tests to pinpoint likely defect locations, e.g., [25]), while mutation testing can consider all visited and reachable parts of the program (although profitable areas are certainly prioritized).

The coupling effect hypothesis (CEH) states that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults" [38]. Thus, even if mutation testing assesses a test suite to be of high quality using only simple mutants, one can have confidence that the test suite will also be of high quality with respect to complex (higher-order) mutants. For example, tests developed to kill simple mutants were also able to kill over 99% of second- and third-order mutants historically [38]. Following Offutt, we propose the following dual formulation, the search-based *program repair coupling effect hypothesis*: "complex faults are coupled to simple faults in such a way that a set of mutation operators that can repair all simple faults in a program will be able to repair a high percentage of the complex faults." This formulation addresses some observations about earlier repair results (e.g., "why is GenProg typically able to produce simple patches for bugs when humans used complex patches?" [27]).

Whether or not this program repair CEH is a true claim about real-world software system is unknown. While some evidence provides minor support (e.g., in Section V, many faults repaired with higher-order edits can also be repaired with first-order edits), there is a troubling absence of evidence regarding repairs to complex faults. Broadly, current generate-and-validate program repair techniques can address about one-sixth to one-half of general defects [12], [26], [27]. It is unknown whether the rest require more time (cf. [28]) or better mutation operators (cf. [26]) or something else entirely. Since fixing (or understanding why one cannot fix) these remaining bugs is a critical challenge for program repair, we hope that this explicit formulation will inspire repair research to consider this question with the same rigor that the mutation testing community has applied to probing the CEH [37], [38].

B. Formulation

We highlight the duality of generate-and-validate repair and mutation testing in Figure 2, which formalizes ideal forms of mutation testing and program repair. Both mutation testing and program repair are concerned with functional quality and generality (e.g., a test suite mistakenly deemed adequate may not detect future faults; a repair mistakenly deemed adequate may not generalize or may not safeguard required behavior) which we encode explicitly by including terms denoting future (held-out) tests or scenarios.

Given a program P, a current test suite Test, a set of non-equivalent mutants produced by mutation testing operators MTMut, and a held-out future workload or test suite $\mathsf{FutureTest}$, we formulate mutation testing as follows: Under idealized mutation testing, a test suite is of high quality if $MT(P, \mathsf{Test}) = \mathsf{true}$ holds. That is, if P passes all tests in Test and every mutant fails at least one test. In practice, the

$$MT(P, \mathsf{Test}) = \mathsf{true} \text{ iff}$$

$$\begin{pmatrix} (\forall t \in \mathsf{Test}.\ t(P)) \ \wedge \\ (\forall m \in \mathsf{MTMut}(P).\ \exists t \in \mathsf{Test}.\ \neg t(m)) \end{pmatrix} \implies \qquad \forall t \in \mathsf{FutureTest}.\ t(P)$$

$$PR(P, \mathsf{Test}, \mathsf{NTest}) = m \text{ iff}$$

$$\begin{pmatrix} (\forall t \in \mathsf{Test}.\ t(P)) \ \wedge \ (\forall t \in \mathsf{NTest}.\ \neg t(P)) \\ \wedge \ m \in \mathsf{PRMut}(P) \ \wedge \ (\forall t \in \mathsf{Test} \cup \mathsf{NTest}.\ t(m)) \end{pmatrix} \implies \begin{pmatrix} (\forall t \in \mathsf{FutureTest}.\ t(P) \implies t(m)) \\ \wedge \ (\forall t \in \mathsf{FutureNTest}.\ t(m)) \end{pmatrix}$$

Fig. 2. Dual formulation of idealized mutation testing and idealized search-based program repair. Ideally, if mutation testing indicates that a test suite is of high quality ($MT(P, \mathsf{Test}) = \mathsf{true}$) then that suite should confer high confidence of the program's correctness: passing that suite should imply passing all future scenarios. Dually (and ideally), if program repair succeeds at finding a repair ($PR(P, \mathsf{Test}) = m$) then that repair should address all present and future instances of that bug (pass all negative tests) while safeguarding all other behavior: if the original program would succeed at a test, so should the repair. The right-hand-side consequent clauses encode quality: a low-quality repair (perhaps resulting from inadequate Test) will appear to succeed but may degrade functionality or fail to repair the bug on unseen future scenarios, while low-quality mutation testing (perhaps resulting from inadequate MTMut) will appear to suggest that the test suite is of high quality when it fact it does not predict future success.

equivalent mutant problem implies that MTMut will contain equivalent mutants preventing a perfect score.

Similarly, given a program P, a current positive test suite encoding required behavior Test, a current negative test suite encoding the bug NTest, a held-out future workload or test suite FutureTest, and held-out future instances of the same bug FutureNTest, we formulate search-based program repair. Idealized program repair succeeds on mutation m ($PR(P, \mathsf{Test}, \mathsf{NTest}) = m$) if all 4 hypotheses (every positive test initially passes, every negative test initially fails, the repair can be found in the set of possible constructive mutations (edits), and the repair passes all tests) imply that the repair is of high quality. A high quality repair retains functionality by passing the same future tests that the original would, and it defeats future instances of the same bug.

A key observation is that our confidence in mutant testing increases with the set non-redundant mutants considered (MTMut), but our confidence in the quality of a program repair gains increases with the set of non-redundant tests (Test).[3] We find that |MTMut| is much greater than |Test| in practice. For example, the number of first-order mutants in our experiments typically exceeds the number of tests by an order of magnitude, as shown in Table I. Thus, program repair has a relative advantage in terms of search: not all of PRMut need be considered as long as a repair is found that passes the test suite. Similarly, the dual of the basic mutation testing optimization that "a mutant need not be further tested after it has been killed by one test" is that "a candidate repair need not be further tested after it has been killed by one test". These asymmetrical search conditions (the enumeration of tests can stop as soon as one fails, and the enumeration of candidate repairs can stop as soon as one succeeds) form the heart of our adaptive search algorithm (see Section IV-A).

[3]Our presentation follows the common practice of treating the test suite as an input but treating the mutation operators as part of the algorithm; this need not be the case, and mutation testing is often parametric with respect to the mutation operators used [22].

C. Implications

The formalism points to an asymmetry between the two paradigms, which we exploit in AE, namely, that the enumeration of tests can stop as soon as one fails (the mutation testing insight), and the enumeration of candidate repairs can stop as soon as one succeeds (the program repair insight). From this perspective, several optimizations in generate-and-validate repair can be seen as duals of existing optimizations in mutation testing, and additional techniques from mutation testing may suggest new avenues for continued improvements to program repair. We list five examples of the former and discuss the latter in Section VIII:

1) GenProg's use of three statement-level tree operators (mutations) to form PRMut is a dual of "selective mutation", in which a small set of operators is shown to generate MTMut without losing test effectiveness [30].
2) GenProg experiments that evaluate only a subset of PRMut with crossover disabled [28] are a dual of "mutant sampling", in which only a subset of MTMut is evaluated [31].
3) GenProg's use of multiple operations per mutant, gathered up over multiple generations, is essentially "higher-order mutation" [21]. Just as a subsuming higher-order mutation may be harder to kill than its component first-order mutations, so too may a higher-order repair be of higher quality than the individual first-order mutations from which it was constructed [23, p. 7].
4) Attempts to improve the objective (fitness) functions for program repair by considering sets of predicates over program variables instead of using all raw test cases [15] are a dual of "weak mutation" [20], in which a program is broken down into components, and mutants are only checked immediately after the execution point of the mutated component [23, p. 8].
5) AE's compilation of multiple candidate patches into a single program with run-time guards (see Section IV-A) is a direct adaptation of "super-mutant" or "schemata" techniques, by researchers such as Untch or Mathur, for

compiling all possible mutants into a single program (e.g., [50]).

Finally, our use of approximate program equivalence is directly related to the "equivalent mutant problem" [23, p. 9], where mutation-testing regimes determine if a mutant is semantically equivalent to the original. AE's use of dataflow analysis techniques to approximate program equivalence for detecting equivalent repairs is thus exactly the dual of Baldwin and Sayward's use of such heuristics for detecting equivalent mutants [5]. Offutt and Craft evaluated six compiler optimizations that can be used to detect equivalent mutants (dead code, constant propagation, invariant propagation, common subexpression, loop invariant, hosting and sinking) and found that such compiler techniques could detect about half [35]. The domains are sufficiently different that their results do not apply directly: For example, Offutt and Craft find that only about 6% of mutants can be found equivalent via dead code analysis, whereas we find that significantly more candidate repairs can be found equivalent via dead code analysis. Similarly, our primary analysis (instruction scheduling), which works very well for program repair, is not among those considered by early work in mutation testing. In mutation testing, the equivalent mutant problem can be thought of as related to result quality, while in program repair, the dual issue is one of performance optimization by search space reduction.

VII. Related Work

Related work from the subfields of program repair and mutation testing is most relevant to this paper. We characterize related repair work along a number of dimensions, summarize our differences, and then discuss mutation testing.

Domain-specific repair. Several repair methods target particular classes of bugs. AFix generates correct fixes for single-variable atomicity violations [24]. Jolt detects and recovers from infinite loops at runtime [8]. Smirnov *et al.* insert memory overflow detection into programs, exposing faulty traces from which they generate proposed patches [47]. Sidiroglou and Keromytis use intrusion detection to build patches for vulnerable memory allocations [46]. Demsky *et al.* repair inconsistent data structures at runtime via constraint solving and formal specifications [14]. Coker and Hafiz address unsafe integer use in C by identifying faulty patterns and applying template-style code transformations with respect to type and operator safety to correct erroneous runtime behavior [10].

General repair. Other approaches target software defects more generally. Arcuri proposed using GP to repair programs [4]; and several authors explore evolutionary improvements [55] and bytecode evolution [39]. ClearView notes errors at runtime and creates binary repairs that rectify erroneous runtime conditions [40]. The ARMOR tool replaces library calls with functionally equivalent statements: These differing implementations support recovery from erroneous runtime behavior [9]. AutoFix-E builds semantically sound patches using testing and Eiffel contracts [52]. SemFix uses symbolic execution to identify faulty program constraints from tests and builds repairs from relevant variables and constructive operators to alter the state of the program at the fault location [34]. Kim *et al.* introduced PAR, which systematically applies mined bug repair patterns from human-created patches to known faults, leveraging semantic similarities between bugs and human expertise [26]. Debroy and Wong [12] use fault localization and mutation to find repairs.

The work in this paper is related to generate-and-validate program repair techniques such as GenProg [27], [54], PAR, ClearView, and Debroy and Wong. However, it differs in several ways: We compute a quotient space of possible repairs using an approximation to program equivalence; we propose an adaptive test selection strategy, and we make explicit a duality with mutation testing. Unlike the work that targets a particular defect class, our approach is general and handles multiple types of defects without a priori knowledge. Unlike techniques that use synthesis (e.g., SemFix) or specifications (e.g., AutoFix-E), our approach has been demonstrated on orders-of-magnitude larger programs. As our cost model reveals, however, many aspects of of program repair are orthogonal. Our approach could easily "slot in" a better fault localization technique (as in SemFix or Debroy and Wong) or a mutation space that includes templates adapted from human repairs (as in PAR).

GenProg's RepairStrat uses the GP search heuristic which assumes implicitly that mutations of candidates that have previously passed the most test cases (i.e., have high fitness) are the most likely to be valid repairs. This approach has succeeded on problems, but no explanation has been offered for why it might hold, and other approaches are possible (e.g., perhaps mutations to previously unexplored parts of the program should be favored). Similarly, recent GenProg studies have used a TestStrat that "runs the candidate on a random 10% of the test cases; if it passes all of those, run it on the rest." Our explicit formulation in terms of an arbitrary RepairStrat and TestStrat generalizes GenProg's random strategy. Our use of a sound approximation to program equivalence could also be applied to other generate-and-validate program repair techniques (e.g., Debroy and Wong [12]).

Mutation testing. Mutation testing measures the adequacy of test suites, often highlighting hidden faults by revealing untested parts of a program [18], [23]. Mutation operators commonly mimic the types of errors a developer might make [13]. Untested parts of a system are exposed if the mutations change the program behavior in a meaningful way and the test suite fails to detect those changes. Critical challenges for mutation testing include the prohibitively large search space of possible mutants, the high cost of testing, and the difficulty in determining if a mutant actually changes program behavior (and thus should be caught by tests) or is equivalent to the original (and thus should not be) (cf. [44]).

Early work in mutation testing attempted to solve this problem by sampling mutants (e.g., [1], [7]) or systematically selecting which mutations to make (e.g., [30]). Previous work in detecting equivalent mutants considered many possible approaches: using compiler optimizations [5], [35], constraint solving [36], program slicing [19], [51], attempting to diversify mutants via program evolution [2], and code coverage [43].

In this paper we propose that an important subset of generate-and-validate program repair can formalized as a dual of mutation testing. In this ideal view, mutation testing takes a program that *passes* its test suite and requires that *all* mutants based on human *mistakes* from the *entire* program that are not equivalent *fail* at least *one* test. By contrast, repair takes a program that *fails* its test suite and requires that *one* mutant based on human *repairs* from the *fault localization* be found that *passes all* tests. The equivalent mutant problem relates to outcome quality for mutation testing (is the adequacy score meaningful); it relates only to performance for program repair (are redundant repairs considered). While both techniques are based on a competent programmer hypothesis, the coupling effect hypothesis for mutation testing relates to tests ("tests that detect simple faults will also detect many complex faults") while for program repair it relates to operators ("mutation operators that repair simple faults can also repair many complex faults"). The structure of this duality (e.g., program repair can short-circuit its search with one failing test or one passing repair) informs our proposed algorithm. It also explains AE's scalability: most mutation testing work presents evaluations using programs with less than 1,000 lines [23, Tab. IX], and the largest involves around 176,500 lines of code [43]. Comparatively, we evaluated on nearly 30 times the largest previously investigated amount of code, including one system that encompasses over 2.8 million lines of code itself: Program repair requires exhaustive *testing* to validate the impact of a repair while mutation testing desires exhaustive whole-program *mutation* to validate adequacy of a test suite. In addition, this duality highlights areas that may benefit from existing mutation testing techniques.

VIII. Future Work

The crucial issue of repair quality is not adressed here, but note that program repairs similar to those of AE have been successfully evaluated by Red Teams [40], held out test cases and fuzz testing [29], and human judgments of maintainability [17] and acceptability [26]. Even incorrect candidate patches cause bugs to be addressed more rapidly [53], so reducing the cost of repairs while maintaining quality is worthwhile.

There are several promising directions for future improvements to AE. "Mutant clustering" selects subsets of mutants using clustering algorithms [22] (such that mutants in the same cluster are killed by similar sets of tests): such a technique could be adopted for our repair strategy (cluster candidate repairs by testing behavior and prioritize repairs that differ from previously investigated clusters). "Selective mutation" finds a small set of mutation operators that generate all possible mutants, often by mathematical models and statistical formulations [45]. Such techniques are appealing compared to post hoc measurements of operator effectiveness [28] and suggest a path to principled, weighted combinations of simple mutations [54] and complex templates [26], both of which are effective independently. Finally, "higher-order mutation" finds rarer higher order mutants corresponding to subtle faults and finds that higher-order mutants may be harder to kill than their first-order component [21]. This is similar to the issue in repair where two edits may be required to fix a bug, but each reduces quality individually (e.g., consider adding a `lock` and `unlock` to a critical section, where adding either one without the other deadlocks the program. Insights such as these may lead to significant improvements for current program repair methods which succeed on about 50% of attempted repairs [34], [27].

Better equivalence approximations from mutation testing [2], [36], [43], [51] could augment our instruction scheduling heuristic. Just as the CPH encourages mutation testing to favor local operations corresponding to simple bugs [13], program repair may benefit from higher-level structural mutations (e.g., introducing new types, changing function signatures, etc.), which are integral to many human repairs.

IX. Conclusion

This paper formalizes the important costs of generate-and-validate program repair, highlighting the dependencies among five elements: fault localization, possible repairs, the test suite, the repair selection strategy, and the test selection strategy. We introduced a deterministic repair algorithm based on those insights that can dynamically select tests and candidates based on the current history of the run. The algorithm computes the quotient space of candidate repairs with respect to an approximate program equivalence relation, using syntactic and dataflow analyses to avoid superfluous test when the outcomes are provably already known. We evaluated the algorithm on 105 bugs in 5 million lines of code, comparing to GenProg. We find that our algorithm reduces the search space by an order of magnitude. Using only first-order edits, our algorithm finds most of the repairs found by GenProg, and it finds more repairs when GenProg is limited to first-order edits. The algorithm achieves these results by reducing the number of test suite evaluations required to find a repair by an order of magnitude and the monetary cost by a factor of three. Finally, we characterize generate-and-validate program repair as a dual of mutation testing, helping to explain current and past successes as well as opening the door to future advances.

Acknowledgments

The authors are sincerely grateful to J. Offutt and P. Amman for insightful discussions of the mutation testing aspect of this work. We also thank J. Dorn, M. Harman, J. Knight and C. Le Goues for thoughtful commentary. We acknowledge the partial support of NSF (CCF 0954024, CCF 0905373) AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231) and the Santa Fe Institute.

References

[1] A. T. Acree. *On Mutation.* PhD thesis, Georgia Tech, 1980.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference*, pages 1338–1349, 2004.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[4] A. Arcuri. On the automation of fixing software bugs. In *Doctoral Symposium — International Conference on Software Engineering*, 2008.

[5] D. Baldwin and F. Sayward. *Heuristics for Determining Equivalence of Program Mutations*. Department of Computer Science: Research report. Yale University, 1979.

[6] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging sfotware. Technical report, University of Cambridge, Judge Business School, 2013.

[7] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, Connecticut, 1980.

[8] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, 2011.

[9] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *International Conference on Sofware Engineering*, 2013.

[10] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *International Conference on Sofware Engineering*, 2013.

[11] Consumer Reports. Online exposure: social networks, mobile phones, and scams can threaten your security. Technical report, June 2011.

[12] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[13] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[14] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, 2006.

[15] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, pages 965–972, 2010.

[16] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[17] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[18] R. Geist, J. A. Offutt, and F. C. Harris Jr. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computing*, 41(5):550–558, 1992.

[19] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[20] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.

[21] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[22] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference*, pages 94–98, 2008.

[23] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[24] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

[25] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[26] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Sofware Engineering*, 2013.

[27] 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 *International Conference on Software Engineering*, pages 3–13, 2012.

[28] C. Le Goues, S. Forrest, and W. Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[30] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference*, pages 604–605, 1991.

[31] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Softw. Test., Verif. Reliab.*, 4(1):9–31, 1994.

[32] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *USENIX Security Symposium*, pages 67–82, 2009.

[33] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, pages 213–228, 2002.

[34] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[35] A. Offutt and W. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[36] A. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[37] A. J. Offutt. The coupling effect: fact or fiction. *SIGSOFT Softw. Eng. Notes*, 14(8):131–140, Nov. 1989.

[38] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, Jan. 1992.

[39] M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192, 2011.

[40] 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 *Symposium on Operating Systems Principles*, 2009.

[41] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[42] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Notices*, 39(10):432–448, 2004.

[43] D. Schuler and A. Zeller. (un-)covering equivalent mutants. In *International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[44] E. Schulte, Z. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, To Appear, 2013.

[45] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, 2008.

[46] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[47] A. Smirnov and T.-C. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*, 2005.

[48] Symantec. Internet security threat report. In *http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf*, Sept. 2006.

[49] Symantec. Internet security threat report. In *https://www4.symantec.com/mktginfo/downloads/21182883_GA_REPORT_ISTR_Main-Report_04-11_HI-RES.pdf*, Apr. 2011.

[50] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

[51] J. Voas and G. McGraw. *Software fault injection: inoculating programs against errors*. Wiley Computer Pub., 1998.

[52] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[53] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[54] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[55] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *Transactions on Evolutionary Computation*, 15(4):515–538, 2011.