An Investigation into the Use of Mutation Analysis for Automated Program Repair

Christopher Steven Timperley[1], Susan Stepney[2], and Claire Le Goues[1]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] University of York, York, UK

Abstract. Research in Search-Based Automated Program Repair has demonstrated promising results, but has nevertheless been largely confined to small, single-edit patches using a limited set of mutation operators. Tackling a broader spectrum of bugs will require multiple edits and a larger set of operators, leading to a combinatorial explosion of the search space. This motivates the need for more efficient search techniques. We propose to use the test case results of candidate patches to localise suitable fix locations. We analysed the test suite results of single-edit patches, generated from a random walk across 28 bugs in 6 programs. Based on the findings of this analysis, we propose a number of mutation-based fault localisation techniques, which we subsequently evaluate by measuring how accurately they locate the statements at which the search was able to generate a solution. After demonstrating that these techniques fail to result in a significant improvement, we discuss why this may be the case, despite the successes of mutation-based fault localisation in previous studies.

Keywords: automated program repair, mutation analysis, fault localisation

1 Introduction

The worldwide cost of debugging and repairing software bugs is estimated to be $312 billion per year; on average, programmers spend roughly 50% of their time finding and fixing bugs [1]. Research in *automated program repair* (APR) seeks to tackle this problem. Generate-and-validate (G&V) is one approach to APR, also known as search-based program repair, which uses meta-heuristics—such as random search [18] or genetic programming [2,8]—to discover patches that lead a program to pass a given set of test cases. At a high level, G&V begins with *fault localisation*, followed by continual processes of *generation* and *validation*. Fault localisation is typically performed using spectra-based fault localisation techniques (SBFL) [25]. SBFL assigns suspiciousness values to statements in the program, based on their dynamic association with the failing tests. Patches are generated by selecting statements according to their suspiciousness, and sampling *edits* at those statements from the repair space. This repair space is defined by a set of transformation schemas, describing transformation shapes (e.g., insert statement, tighten if condition, replace call argument), and transformation

ingredients, supplying the parameters necessary to complete shapes (e.g. a particular statement). Candidate patches are evaluated for correctness by running the patched program on the original test suite; repair is indicated by passing all of the tests.

Different G&V approaches vary in their mutation operators and traversal techniques. For example, GenProg [8] constructs patches that may append, replace or delete statements within the program, reusing existing statements within the program as fix ingredients. Other transformation schemas have been proposed based on human-produced patches [6] or a value search to reduce the cost of patch evaluation [10]. Search space traversal schemes employed include genetic programming [8], random search [18], and a deterministic walk [23].

Despite promising early results, most G&V techniques are currently limited to generating patches for a relatively small sub-set of single-line bugs [23,18,11]. To repair a wider variety of bugs, techniques will need to use richer, more granular transformation schemas, and to construct multiple-line patches. However, this produces a combinatorial explosion in the size of the search space. This motivates a need for methods to prune the exploded search space.

Inspired by recent work in mutation testing [16,14], we propose to use candidate test suite evaluations to identify suitable fix locations *online*. Mutation-based fault localisation show promising results when ranking statements as candidates for human modification; We explicitly evaluate their utility in assigning suspiciousness scores to candidate repair locations, the key concern in localisation for repair. To determine whether the results of candidate patch evaluations may be used to localise the fault, we first perform a mutation analysis on a sample of a particular G&V repair search space across 28 bugs in six real-world C programs. We use the same ground truth as previous studies on fault localisation, assuming the location(s) of the human-written repair or the injected fault to be a suitable fix location [14,16,25]. For the sake of convenience, we refer to these locations as "faulty"; non-modified statements are considered to be "correct".

We find that faulty locations exhibit a different average rate of passing-to-failing tests across their mutants than statements assumed to be correct. We also observe that an average of 30.07% of (compiling) mutants have no impact on the outcomes of the test suite, mirroring earlier findings by Schulte et al. [20]. Similarly, we find that, on average, 26.44% of mutants covered by at least one previously passing test fail all of their covering tests. These results suggest a largely all-or-nothing search space, in which most mutants either pass all of their (covering) tests, or none at all.

Based on the findings of our analysis, we evaluate a number of alternative fault localisation techniques in terms of their ability to localise statements at which a fix was found during the search (as opposed to assessing how well they localise the location of the human repair). We show that little benefit is gained by incorporating the results of candidate patch evaluations into the fault localisation, and that any gains are not particularly consistent. The best localisation approach using this information outperformed GenProg's default approach on just over half of the cases. To benefit from the knowledge of candidate eval-

uations, a more granular repair model is needed—to allow subtle faults to be detected—as well as a more effective means of aggregating different sources of fault localisation information. Overall, our primary contributions are:

- A detailed mutation analysis sampled from GenProg's search space, covering 28 bugs across six real-world C programs. Our results show that statement-level mutation operators used in many search-based program repair techniques can identify the code that humans modify to fix bugs.
- An evaluation of several alternative fault localisation techniques which use the test case outcomes of mutants produced during the search. Our results show that, although informative in terms of human-modified bug-fixing code, online mutation-based fault localisation does not definitively outperform existing offline SBFL approaches.
- An informed discussion of the limitations of GenProg's statement-level mutation operators in identifying faulty locations, and how these limitations might be addressed by alternative mutation operators.

2 Fault Localisation in Search-Based Program Repair

In this study, we examine the search space of GenProg, an established generate-and-validate APR technique, with a publicly-available implementation, based on genetic programming. We focus this discussion primarily on GenProg's approach to fault localisation, for illustration, but the principles generalise to most existing techniques in APR.

GenProg assigns suspiciousness values to each program statement based on their coverage by the failing and passing test cases. In the initial formulation, statements executed exclusively by the failing test cases are assigned a weight of 1.0; those executed by both failing and passing tests, 0.1; those not executed by a failing test, 0.0. Alternative weighting schemes have been explored since [9], including those that draw directly on advances in spectrum-based fault localisation [19]. Statements are sampled in proportion to their weight.

We now address Mutation-Based Fault Localisation (MBFL), a relatively new approach based on mutation analysis [14,16]. This analysis generates and evaluates a large number of mutants on the test cases. Each mutant is a variant of the original program, obtained by applying a traditional mutation testing operator (e.g., flip comparison operator) at a single location [16]. Two seminal approaches to MBFL are MUSE [14] and Metallaxis [16]. Both of these approaches share a common intuition: mutants generated at the fault location should exhibit different test suite outcomes to those generated at non-faulty locations. Despite sharing this intuition, each technique generates its suspiciousness values according to contradictory set of assumptions.

Prior to computing suspiciousness values for each statement $s \in S$, Metallaxis first computes explicit suspiciousness values for each mutant $m \in M$. The suspiciousness of a mutant is given by the similarity of its behaviour to that of the original, faulty program, measured using a variant of the Ochiai suspiciousness metric [25], given below, where $\#K$ is the number of tests that "kill" the

mutant (i.e., the tests which the mutant fails), $\#K_n$ is the number of previously failing tests that kill the mutant, and $\#K_p$ is the number of previously passing tests that kill the mutant:

$$\mu_{Metallaxis}(m) = \frac{\#K_n}{\sqrt{\#K \cdot (\#K_p + \#K_n)}} = \frac{\#K_n}{\#K} \tag{1}$$

To determine the suspicious of a statement, the set of mutants at that statement M_s is aggregated as follows:

$$\mu_{Metallaxis}(s; M) = \begin{cases} \max_{m \in M_s} \mu_{Metallaxis}(m) & M_s \neq \emptyset \\ 1.0 & \text{otherwise} \end{cases} \tag{2}$$

MUSE, on the other hand, computes statement suspiciousness directly, based on the average passing-to-failing rate $p2f$ and failing-to-passing $f2p$ rate of mutants at that statement. $p2f$ describes the fraction of previously passing tests that are failed by the mutant; $f2p$ describes the fraction of previously failing tests that are passed by the mutant. MUSE discards all of neutral mutants (i.e., mutants whose test outcomes are the same as the original program), and computes suspiciousness as:

$$\mu_{MUSE}(s) = \frac{1}{\#M_s} \cdot \sum_{m \in M_s} (f2p(m) - \alpha \cdot p2f(m)) \tag{3}$$

where α compensates for the greater likelihood that a previously passing test will fail than a previously failing test will pass:

$$\alpha = \frac{f2p_{all}}{\#M \cdot |T_{Pass}|} \cdot \frac{\#M \cdot |T_{Fail}|}{p2f_{all}} \tag{4}$$

where $f2p_{all}$ and $p2f_{all}$ give the number of tests, across all mutants, whose outcomes change, and T_{Fail} and T_{Pass} denote the set of initially failing and passing test cases, respectively.

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

- According to Metallaxis, a statement is suspicious as its most suspicious mutant. This behaviour assumes that the search landscape is mostly composed of non-neutral mutants. If the search space contains a large number of neutral mutants [20], Metallaxis assigns the maximum suspiciousness value to most statements. In contrast, MUSE actively discards its neutral mutants, and computes the suspiciousness of a statement as the average suspiciousness of its mutants, indicating a robustness to sampling noise.
- Whereas MUSE significantly increases the suspiciousness of statements containing mutants which pass previously failing test cases, Metallaxis implicitly decreases the suspiciousness of such statements. These behaviours highlight a contradiction between the techniques' underlying assumptions: MUSE sees

partial solutions as signs of a repair, whilst Metallaxis views them as either irrelevant, or the result of overfitting.

Both techniques have demonstrated significant improvement over previous fault localisation approaches. However, evaluations have been limited to manually-seeded faults in small-to-medium sized programs, and use metrics that have been shown to be inappropriate for automated program repair [19], where the degree of difference in suspiciousness is more important than rank.

3 Analysis

The mutation-based fault localisation approaches described in the previous section suggest a natural overlap with search-based program repair, which effectively produces a large number of candidate mutants throughout the generate-and-validate process. This suggests a mechanism for *online* fault localisation that leverages the existing mutation approach presented by the underlying repair process. However, the raw suspiciousness scores produced by a fault localisation technique are more important than the ranks. Thus, the utility of this approach depends more on the discriminatory power of the $p2f$ and $f2p$ scores than on their raw accuracy (evaluated in the traditional way).

Thus, to determine whether such mutation analysis may be used to improve fault localisation in this context, we first analyse whether mutants at (assumed) faulty and correct statements exhibit different test suite outcomes to one another. Given the experimental parameters enumerated in Section 3.1, we begin by answering the following research questions (Section 3.2):

- **RQ1:** Can statements that were modified by the human fix be discriminated from those that were not, on the basis of the $p2f$ rates of their mutants?
- **RQ2:** Can human-modified statements be distinguished from non-modified statements, based on the fail-to-passing rates $f2p$ of their mutants?

As a potential means of improving offline fault localisation, we also ask:

- **RQ3:** Are statements covered by the fewest number of previously passing tests the most likely to contain the fault?

Based on the results of this analysis, we propose and evaluate two new fault localisation strategies for search-based program repair (Section 3.3). However, we find that these new strategies do not offer significant improvement over previous SBFL strategies. We provide insight as to why not, as well as implications, supported by additional findings; these are detailed in Section 3.4.

3.1 Experimental Setup

We analyse test case results for a sample of single-edit mutants taken from 28 bugs across 6 real-world C programs. Some of these defects have been previously studied in the context of automated program repair; all are provided by the the RepairBox platform.[3] 15 of these bugs are artificial, injected into 3 small-to-medium sized programs sourced from the Software Infrastructure Repository

[3] `https://github.com/squaresLab/RepairBox`

[3]—the same source used to evaluate MUSE and Metallaxis. We include these bugs to determine whether GenProg's repair operators may be used to perform MBFL, rather than traditional mutation testing operators, used by existing approaches [16,14]. To determine whether MBFL remains effective when applied in the wild, we supplement this dataset with 13 real-world bugs across 3 large-scale, real-world programs. Table 1 summarises the studied programs.

Source	Program	Scenarios	kLOC	Tests	Artificial?
SIR	gzip	6	6	104	✓
	grep	2	10	146	✓
	sed	7	14	255	✓
RBX	OpenSSL	5	248	77	✗
	Python	4	446	344	✗
	PHP	4	789	8597	✗

Table 1: Subjects under study. "Source" indicates the benchmark source (SIR the Software Infrastructure Repository; RBX, RepairBox); "Scenarios" refers to the number of independent defective versions considered per program; "kLOC" measures the number of thousands of lines of C code in the program; "Tests" indicates the average number of tests over all scenarios for a program.

We use GenProg, a search-based program repair technique with well-established and commonly-used mutation operators, to focus this evaluation. However, we anticipate the results can generalise to any G&V technique following a similar paradigm. To collect the necessary data for the analysis, we first generated a list of all the single-edit patches within GenProg's search space, before randomly shuffling that list and evaluating as many patches as possible within a 12-hour window. This 12-hour random walk was repeated for each of the bugs within the dataset.[4] We restrict the generation of mutants to the sub-set of statements covered by at least one of the failing test cases. For historical reasons, and given its similarity to traditional mutation testing operators, we also ensured that a deletion was attempted at each statement. For the purposes of balancing replication with performance, we performed each run using a minimal, purpose-built Docker container, provided by RepairBox. We used a C4.Large instance on Amazon EC2 for the artificial bugs, and a DS1_V2 instance on Microsoft Azure for the real-world bugs.

3.2 Analysis

A brief summary of the results of the mutation analysis is given in Table 2. We observe that most mutants exhibit an all-or-nothing behaviour: either their test

[4] Source code and a Docker image for the version of GenProg used by this study is publicly available at: `https://bitbucket.org/ChrisTimperley/gp3`

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

Table 2: Mutation analysis results for each bug scenario. “Mutants” shows number of mutants generated within the 12-hour random walk. “Sample Rate” is the average number of mutants per suspicious statement. “Compiling” shows the percentage of mutants that successfully compiled. “Neutral” shows the percentage of mutants with no effect on test outcomes, whereas “Lethal” shows the percentage of mutants that fail all covering tests.

outcomes remain wholly unchanged, or all of their covering tests are failed. We also observe low sample rates (< 1 mutant per suspicious statement) for most Python and PHP scenarios, because of substantial compilation overhead and, for statements covered by many tests, the cost of evaluating hundreds or thousands of test cases per mutant.

RQ1: Can statements that were modified by the human fix be discriminated from those that were not, on the basis of the $p2f$ rates of their mutants?

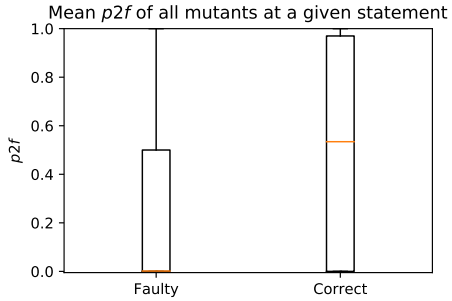

Fig. 1: We observe different mean $p2f$ distributions for faulty and correct statements ($KS2 = 0.301; p = 0.003, \hat{A} = 0.679$ [medium effect]).

To avoid misleading results, we omit mutants that do not compile, and those that are not covered by any of the passing tests. In line with our expectations and previous findings, we observe different mean $p2f$ rates between (assumed) faulty and correct statements, Figure 1.

Using a two-way Kolmogorov-Smirnov test, we reject the null hypothesis ($p < 0.05$) that the samples for the faulty and correct statements are drawn from the same distribution. Between the $p2f$ distributions for correct and faulty statements, we find an effect size of 0.679 (measured by the Vargha-Delaney $\hat{A}$ measure [22]), indicating that correct statements tend to have a higher mean $p2f$ than faulty statements. This finding supports the intuition that modifications to correct statements are likely to result in a greater degree of functionality loss.

RQ2: Can human-modified statements be distinguished from non-modified statements, based on the fail-to-passing rates $f2p$ of their mutants?

To answer this question, we first removed all non-compiling mutants from consideration. We then removed all mutants corresponding to acceptable solutions, giving us the most complete information possible without knowledge of a solution. Figure 2 compares the mean $f2p$ for faulty and correct statements.

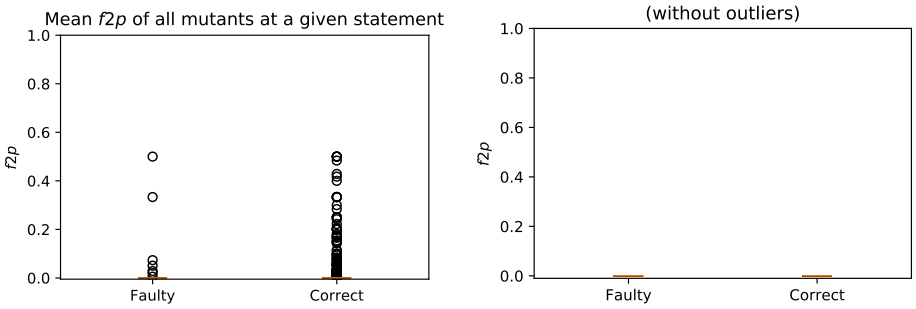

Fig. 2: We observe similar distributions of mean $f2p$ values for faulty and correct statements ($KS2 = 0.185; p = 0.177$). In both cases, more than half of the mutants at each statement did not pass any of the previously failing tests.

We find that the mean $f2p$ is zero for the majority of statements, regardless of whether or not they are assumed to be correct. On closer inspection we find that only 2.09% of mutants have any impact on the outcomes of the previously failing tests. This suggests that $f2p$ information may not be particularly effective at distinguishing faulty and correct statements.

RQ3: Are statements covered by the fewest number of previously passing tests the most likely to contain the fault?

When the search is restricted to the sub-set of statements covered by all of the failing tests, most SBFL techniques become partly redundant, as the number of (non-)executed failing tests is no longer relevant. Instead, it may be preferable to measure statement suspiciousness as a function of the number of executed and non-executed passing test cases.

Using the results of the analysis, we measure the passing test coverage at each statement where a repair was found, and investigate whether (assumed) faulty statements are covered by fewer passing tests. Accounting for the varying sizes of the test suites, we measure the fraction of passing tests that cover each each statement, rather than the number. To determine coverage relative to other statements in the program, we compute the adjusted coverage as:

$$\text{AdjustedCoverage(s)} = \frac{\#T_{Pass}(s) - \text{MinCoverage}}{\#T_{Pass}} \tag{5}$$

where $T_{Pass}(s)$ is the set of passing tests covered by statement s, and MinCoverage is the number of passing tests that cover that least covered statement.

Measuring the adjusted coverage of each scenario, we find that 90% are covered by fewer than 2% of the previously passing tests, supporting our intuition and the intuition of the original suspiciousness metric used in GenProg.

3.3 Fault Localisation

Using the knowledge gained from our mutation analysis, we propose and evaluate two fault localisation strategies for G&V program repair, which may be aggregated. We aggregate localisations by computing their product. To use these layers in a noisy, online context, we ensure that each is numerically stable and that none assigns a suspiciousness of zero to any statement covered by a previously failing test. We consider:

- **Coverage, μ_{Cov}:** produces a 90% probability that a statement with less than 2% adjusted coverage will be selected, and a 10% probability that a statement with a greater level of coverage will be chosen.
- **Pass-to-Fail:** This layer assigns values between zero and one to each statement, based on the pass-to-fail rates $p2f$ of its mutants:

$$\mu_{p2f}(s) = \frac{1}{\#M_s + 1} \cdot \left[1 + \sum_{m \in M_s} (1 - p2f_m)\right] \tag{6}$$

To assess the effectiveness of the fault localisation approaches that combine these layers, we use the mutants of the analysis—excluding any solutions, to avoid potential biases—to generate a set of suspiciousness values μ for each of the bug scenarios. We then measure the accuracy of a fault localisation technique as the probability of selecting a statement that contains a fix found during the random walk.[5]

Table 3 compares the effectiveness of our proposed fault localisation strategies against a selection of existing strategies, across the 11 bugs for which solutions were found during the random walk. We find that no strategy, whether mutation- or spectrum-based, is dominant: Jaccard, the previously reported [19] best SBFL strategy for APR, is outperformed by GenProg's default strategy in 6/11 cases; our Adjusted Coverage strategy beats GenProg in 6/11 cases, but is also beaten by Jaccard in 6/11 cases. Neither Metallaxis nor MUSE dominate GenProg's default strategy: GP beats MUSE in 7 cases (and draws in 2) and Metallaxis in 8 cases.

Our $p2f$ strategy is beaten by GenProg in 8/11 cases, indicating that passing-to-information, when used alone, is not particularly effective at identifying suitable fix locations. When the Adjusted Coverage and $p2f$ strategies are aggregated by computing their product, the resulting hybrid beats GenProg and Jaccard in 6 and 8 cases, respectively. If the online modifications to μ_{p2f} are removed and $1 - p2f(s)$ is used to compute suspiciousness instead, the resulting localisation outperforms GenProg's fault localisation in all cases.

We experimented with ways of using $f2p$ information, but found the approach either attained near-perfect accuracy (since the only mutants to pass any of the previously failing tests were at statements where a solution was found), or

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

Program	ID	Cov	$p2f$	Cov $\times$ $p2f$	GP	Jac.	MUSE	MXS
OpenSSL	0a2dcb6	0.14	0.91	0.05	1.23	1.82	1.38	0.53
	6979583	75.26	15.96	79.75	37.68	25.00	8.33	8.46
	8e3854a	0.11	1.01	0.10	0.71	0.61	0.95	0.73
	eddef30	76.04	45.92	77.58	66.67	54.55	41.67	41.67
gzip	v1-KP_1	4.46	4.67	6.49	6.63	4.63	2.44	2.60
	v1-TW_3	14.21	4.69	15.32	9.20	6.60	1.94	2.10
	v4-KL_1	2.73	1.72	3.08	2.69	3.03	0.94	0.99
	v5-KL_1	0.33	0.36	0.39	0.40	0.34	0.26	0.27
	v5-KL_8	6.59	0.34	4.67	2.17	4.08	0.30	0.89
sed	v2-AG_17	0.02	0.21	0.02	0.19	0.15	0.19	0.53
	v3-AG_11	6.95	0.61	3.09	0.57	1.92	0.57	0.81

Table 3: Comparison of fault localisation accuracies achieved by different approaches, where accuracy is measured by the probability of sampling a statement containing a fix from the resulting distribution. "Cov", "Jac.", "GP" and "MXS" refer to Adjusted Coverage, Jaccard, GenProg and Metallaxis, respectively.

substantially worse accuracy (since all mutants, other than the solutions at the faulty statements, failed all of the previously failing tests).

3.4 Discussion

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

- **Lack of mutants:** for a number of bug scenarios, we find that excessively long test suite evaluation and compilation times prevent the search from producing an adequate sample of mutants at each statement. In previous research, Moon et al. [15] show that the performance of MUSE is sensitive to the number of samples—at low sample rates (i.e., the average number of samples per statement), MUSE is outperformed by offline SBFL techniques.
- **Lack of passing test coverage:** in cases where most statements are not covered by any passing tests, these statements will be assigned a suspiciousness score of 1.0 by $\mu_{P2F}(s)$; Metallaxis will also assign maximal suspiciousness to such statements. As a result, if the faulty statements are covered by any passing tests, they will be suppressed; if not, the fault localisation will fail to identify the faulty statements amongst the many statements that are not covered by the passing tests.
- **All-or-nothing $f2p$ response:** within GenProg's search space, we observe erratic negative test suite behaviour. In some cases, the only statements with mutants that passed a previously failing test were those where a repair was

found. In other cases, previously failing tests were frequently passed, except at the statements where a repair was found. If one knew which type of $f2p$ response one was dealing with, a more accurate fault localisation might be possible. In the future, we plan to explore whether the rarity of passing previously failing tests might be used to determine whether a given failing-to-passing event is a coincidence or indicative of a potential repair at that statement.

- **Machine-generated repairs:** Across the random walk for each of the 28 bugs, we found fixes at a total of 48 different statements. Only 5 of these 48 statements were also modified by the original patch. This finding suggests that GenProg is crafting repairs unlike those that a human would make, supporting arguments made by Monperrus [13] that automated repair should consider alien-looking repairs, rather than restricting itself to producing human-like repairs. The disparity between the findings of RQ1, that faulty statements exhibit a different mean $p2f$ to correct statements, and the lack of improvement when mutant information is added to the fault localisation may suggest that locations patched by GenProg are less distinguishable by their mutants' behaviours.
- **Coarse-grained mutation operators:** one explanation for the relative lack of success from incorporating the results of the mutation analysis into the fault localisation may be due to the coarse-grained nature of the repair operators within GenProg's search space. With such actions, it may be difficult to expose subtle bugs within the statement, that might otherwise be identified using finer-grained mutation testing operators. In our analysis, we find that most repairs tend to either have no effect on the outcomes of the test suite, or to cause all of their covering tests to fail; this all-or-nothing behaviour may be a consequence of the granularity of the search operators.
- **Combining information:** to combine each of the proposed layers of fault localisation from our evaluation, we computed the product of each of the layers—a necessarily arbitrary decision. A meaningful and effective way of combining information from multiple sources, which may corroborate or conflict, is not immediately clear.

These results suggest a number of possibly fruitful directions to translate the potential of mutation analysis approaches such as MUSE into efficiency gains in automated program repair. Richer repair models, with lower-level repair operators (such as those traditionally used in mutation testing), may mitigate the all-or-nothing behaviour exhibited by mutants generated using GenProg's coarse-grained operators. Test outcomes for particular types of mutants may be informative in refining suspiciousness beyond the statement-level, and may even serve to predict the type of repair that might be needed. Finally, simple weighted averages or products, as we explored in Section 3.3, may be inadequate; it is possible that ensemble learning techniques could more effectively combine sources of information.

4 Related Work

In this section, we discuss previous research related to fault localisation within automated program repair, and other approaches to addressing the difficulties of scaling to larger repairs and search spaces.

Instead of tackling the problem of a growing search space by exploiting knowledge learnt online, several techniques have been proposed to learn the likelihood of candidate repairs based on their features by mining large collections of source code repositories [21,7,11]. A complementary approach to tackle the problem of the expanding search space is to reduce the cost of evaluating candidate patches, whether through test case prioritisation [17,18], test case sampling [4] or removal of redundant tests and (known) semantically equivalent mutants [23]. These approaches complement improved fault localisation for repair. In contrast to syntactic- or heuristic-based G&V repair, semantic repair techniques [12,5] infer partial specifications of desired behavior using test suites and then use synthesis to construct replacement code that satisfies them. These techniques also use test suites to localise faults and to validate patches, and thus could also benefit from improved fault localisation.

A large number of methods for automated debugging and fault localisation exist, including program slicing [24], delta-debugging [26] and various forms of spectra-based fault localisation [25]. To date, most automated repair techniques exclusively use SBFL; it is general and low-cost. SBFL approaches, to which GenProg's default fault localisation method belongs, use the test case coverage information for the program to assign suspiciousness values to each of its locations. Qi et al. [19] conduct a study of the effectiveness of various SBFL techniques when used with GenProg, finding that the Jaccard suspiciousness metric produced the best fault localisation information, as measured by the number of candidate repairs required to find a solution. In contrast to our study, we find no one approach to fault localisation is dominant.

Schulte et al. [20] conduct an empirical study of the robustness of 22 programs to mutation using GenProg's operators, finding that over 30% of generated mutants exhibit no change to the outcomes of their test suites. This behaviour may hinder the effectiveness of MBFL techniques. For instance, Metallaxis will assign maximal suspiciousness to statements with neutral mutants, causing the faulty statements to be suppressed.

5 Conclusions

Although mutation analysis can distinguish between human-modified and human-unmodified statements in a bug-fixing context, these results do not translate directly into clear gains as a fault localisation technique for the purposes of program repair. However, our results provide insight into why this may be the case, and suggest several possibly fruitful future directions for fault localisation for search-based repair. Given the previous successes of Metallaxis and MUSE with mutation testing operators, we believe GenProg's all-or-nothing search space,

in which most edits are either neutral or fail all of their covering tests, may be partly responsible for the lack of clear gains. Low levels of passing test coverage may also preclude the use of mutation-based fault localisation techniques.

To benefit from the knowledge of test suite outcomes for candidate patches, we believe a set of more finely grained mutation operators are required—a requirement that will most likely allow a larger number of bugs to be solved at the same time.

To encourage further investigation, all results from this study, together with the files used to conduct it, are available at:

`https://bitbucket.org/ChrisTimperley/ssbse-2017-data`.

6 Acknowledgements

This research was partially funded by AFRL (#FA8750-15-2-0075) and DARPA (#FA8750-16-2-0042), and an EPSRC DTG; the authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government. The authors additionally wish to thank the anonymous reviewers, whose comments were especially insightful and constructive.

References

1. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. `http://www.prweb.com/releases/2013/1/prweb10298185.htm`, Accessed April, 2017
2. Arcuri, A.: Evolutionary Repair of Faulty Software. Applied Soft Computing 11(4), 3494–3514 (2011)
3. Do, H., Elbaum, S., Rothermel, G.: Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering 10(4), 405–435 (2005)
4. Fast, E., Le Goues, C., Forrest, S., Weimer, W.: Designing Better Fitness Functions for Automated Program Repair. In: Genetic and Evolutionary Computation Conference. pp. 965–972. GECCO '10 (2010)
5. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing Programs with Semantic Code Search. In: Automated Software Engineering. pp. 295–306. ASE '15 (2015)
6. Kim, D., Nam, J., Song, J., Kim, S.: Automatic Patch Generation Learned from Human-written Patches. In: International Conference on Software Engineering. pp. 802–811. ICSE '13 (2013)
7. Le, X.B.D., Lo, D., Goues, C.L.: History driven program repair. In: International Conference on Software Analysis, Evolution, and Reengineering. SANER '16, vol. 1, pp. 213–224 (2016)
8. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering. pp. 3–13. ICSE '12 (2012)
9. Le Goues, C., Weimer, W., Forrest, S.: Representations and Operators for Improving Evolutionary Software Repair. In: Genetic and Evolutionary Computation Conference. pp. 959–966. GECCO '12 (2012)

10. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Joint Meeting on Foundations of Software Engineering. pp. 166–178. ESEC/FSE '15 (2015)
11. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Principles of Programming Languages. pp. 298–312. POPL '16 (2016)
12. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering. pp. 691–701. ICSE '16 (2016)
13. Monperrus, M.: A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In: International Conference on Software Engineering. pp. 234–242. ICSE '14 (2014)
14. Moon, S., Kim, Y., Kim, M., Yoo, S.: Ask the Mutants: Mutating Faulty Programs for Fault Localization. In: International Conference on Software Testing, Verification and Validation. pp. 153–162. ICST '14 (2014)
15. Moon, S., Kim, Y., Kim, M., Yoo, S.: Hybrid-MUSE: Mutating Faulty Programs for Precise Fault Localization. Tech. rep., KAIST (2014)
16. Papadakis, M., Le Traon, Y.: Metallaxis-FL: mutation-based fault localization. Software Testing, Verification and Reliability 25(5-7), 605–628 (2015)
17. Qi, Y., Mao, X., Lei, Y.: Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In: International Conference on Software Maintenance. pp. 180–189 (2013)
18. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The Strength of Random Search on Automated Program Repair. In: International Conference on Software Engineering. pp. 254–265. ICSE '14 (2014)
19. Qi, Y., Mao, X., Lei, Y., Wang, C.: Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. In: International Symposium on Software Testing and Analysis. pp. 191–201. ISSTA '13 (2013)
20. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genetic Programming and Evolvable Machines 15(3), 281–312 (2013)
21. Soto, M., Thung, F., Wong, C.P., Le Goues, C., Lo, D.: A deeper look into bug fixes: patterns, replacements, deletions, and additions. In: International Conference on Mining Software Repositories. pp. 512–515. MSR '16 (2016)
22. Vargha, A., Delaney, H.D.: A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
23. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: Models and first results. In: International Conference on Automated Software Engineering. pp. 356–366. ASE '13 (2013)
24. Weiser, M.: Program Slicing. In: International Conference on Software Engineering. pp. 439–449. ICSE '81 (1981)
25. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based Software Engineering. pp. 244–258. SSBSE '12 (2012)
26. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28(2), 183–200 (2002)