Noname manuscript No.
(will be inserted by the editor)

A Correlation Study between Automated Program Repair and Test-Suite Metrics

Jooyong Yi[1] · Shin Hwei Tan[2] · Sergey Mechtaev[2] · Marcel Böhme[2] · Abhik Roychoudhury[2]

Received: date / Accepted: date

Abstract Automated program repair is increasingly gaining traction, due to its potential to reduce debugging cost greatly. The feasibility of automated program repair has been shown in a number of works, and the research focus is gradually shifting toward the quality of generated patches. One promising direction is to control the quality of generated patches by controlling the quality of test-suites used for automated program repair. In this paper, we ask the following research question: "Can traditional test-suite metrics proposed for the purpose of software testing also be used for the purpose of automated program repair?" We empirically investigate whether traditional test-suite metrics such as statement/branch coverage and mutation score are effective in controlling the reliability of generated repairs (the likelihood that repairs cause regression errors). We conduct the largest-scale experiments of this kind to date with real-world software, and for the first time perform a correlation study between various test-suite metrics and the reliability of generated repairs. Our results show that in general, with the increase of traditional test suite metrics, the reliability of repairs tend to increase. In particular, such a trend is most strongly observed in statement coverage. Our results imply that the traditional test suite metrics proposed for software testing can also be used for automated program repair to improve the reliability of repairs.

Keywords automated program repair · test suite · empirical evaluation · correlation

✉ Jooyong Yi
j.yi@innopolis.ru

Shin Hwei Tan
shinhwei@comp.nus.edu.sg

Sergey Mechtaev
mechtaev@comp.nus.edu.sg

Marcel Böhme
mboehme@comp.nus.edu.sg

Abhik Roychoudhury
abhik@comp.nus.edu.sg

[1] Innopolis University, Republic of Tatarstan, Russian Federation

[2] School of Computing, National University of Singapore, Singapore

1 Introduction

The idea of fixing bugs automatically is gaining traction, as evidenced by the emergence of a new research field of "automated program repair". Researchers have experimentally shown that automated program repair is possible for real-world large-scale software such as the PHP interpreter and Heartbleed-containing OpenSSL (Le Goues et al, 2012a; Nguyen et al, 2013; Le Goues et al, 2012b; Weimer et al, 2013; Nguyen et al, 2013; Kim et al, 2013; White et al, 2011; Dallmeier et al, 2009; Xuan et al, 2017; Assiri and Bieman, 2014; Pei et al, 2014; Debroy and Wong, 2010; Samimi et al, 2012; Qi et al, 2014, 2013; Mechtaev et al, 2016). Currently, the research focus is gradually shifting from the feasibility of automated program repair to the quality of generated patches (Assiri and Bieman, 2014; Smith et al, 2015; Qi et al, 2015; Long and Rinard, 2016a). In particular, these latest research results raise a question about how to generate a "correct" patch—a patch that not only passes all tests available to a repair system, but also indeed fixes the bug. Most of the automated program repair approaches use a test-suite as a proxy of software specification, since formal specification is hardly used in the industry. While the fact that software tests are widely available is advantageous, the fact that a test-suite is an incomplete specification can make a generated repair incomplete; there is generally no guarantee that no other new tests will fail a generated repair. This problem of automated program repair is akin to the problem of software testing; even if all available tests pass the software under test, there is generally no guarantee that no other new tests will fail the software under test. Despite this limitation, it is possible to improve software quality by improving the quality of a test-suite. Likewise, *is it possible to control the quality of automatically generated repair by controlling the quality of a test-suite?* This is our key high-level research question we aim to answer in this paper. Apart from this main research question, we also investigate how test-suite metrics affect repairability (repair success rate) and repair time.

To answer our main research question, we conduct large-scale experiments about the correlation between test-suite quality and automated program repair. Our subject programs contain four large-scale real-world programs such as a PHP interpreter and a TIFF image processing library, in addition to a well-known benchmark, SIR (Do et al, 2005). In comparison, previous studies (see Section 7.1) were conducted with small student programs or SIR subjects. As a result, we can provide stronger empirical evidences about the correlation between the quality of test-suites and the quality of automatically generated repairs than previous studies. Also, we for the first time compare various test-suite metrics such as statement coverage, branch coverage, test-suite size, and mutation score, with regard to their degrees of correlation (i.e., correlation coefficients) with repair quality. As a result, we can answer whether the traditional test suite metrics proposed for the purpose of software testing are also useful in the context of automated program repair, and which test-suite metric is most effective.

With regard to the quality of automatically generated repairs, we focus on the reliability of a generated repair, that is, whether regressions occur in a repair. Judging whether a repair is correct is often subjective and difficult to be automated in the absence of formal specifications. Previous studies investigate the reliability of repairs instead, because whether a generated repair causes regressions can be checked in an automated way (Assiri and Bieman, 2014; Ke et al, 2015; Kong et al, 2015; Smith et al, 2015). That is, once a repair is generated, this repair can be tested with a test-universe (held-out test-suite) that contains tests that were not available at the time of generating the repair. If a failing test is found in the test-universe, it is considered that the repair causes regressions. As in previous studies, we also similarly investigate how often regressions occur to measure the quality of a repair. Meanwhile, we obtain automatically generated repairs by running GENPROG (Le Goues et al, 2012b; Weimer et al, 2013). In total, we collected 3818 repairs from 142 buggy versions of 10 different programs of various sizes (173–1046K LoC), using 14600 randomly sampled test suites. We sample test suites from the whole test cases available in our subjects. While we retrieve the main results from GENPROG-generated repairs, we also conduct smaller scale experiments with another repair tool SEMFIX (Nguyen et al, 2013) to see whether our main results extend beyond GENPROG. GENPROG and SEMFIX are first search-based and constraint-based repair tools, respectively. Search-based repair tools navigate a set of repair candidates through a search algorithm until a repair is found, while constraint-based repair tools first construct repair constraints that should be satisfied by a repair and symbolically search for a repair satisfying the repair constraint using a theorem prover. While our experiments may not generalize to all other repair tools, GENPROG, the repair system we mainly use in our study, has been used in many previous studies on automated program repair (Smith et al, 2015; Kong et al, 2015; Qi et al, 2015; Le Goues et al, 2012a,b; Weimer et al, 2013; Le Goues et al, 2013). Our experimental results obtained from GENPROG complement the results from earlier studies.

Our results show that in general, the traditional metrics of test-suites, that is, statement coverage, branch coverage, test-suite size, and mutation score, are *negatively correlated* with the likelihood that a generated repair causes a regression. In other words, as the traditional metrics of a test-suite increase, generated repair tend to cause regressions less often. Our result implies that the traditional test suite metrics proposed for software testing can also be used for automated program repair. Among the test-suite metrics we investigate, statement coverage is shown to be most strongly correlated with regression ratio. A practical implication is that to reduce regression ratio, increasing statement coverage is likely to be more effective than improving the other test-suite metrics such as branch coverage. However, it should be noted that the highest correlation of statement coverage does not necessarily imply that a statement coverage-adequate test-suite is better than a branch coverage-adequate test-suite.

In summary, the main contributions of this paper are:

– We for the first time conduct a correlation study of automated program repair with various test-suite metrics such as statement coverage, branch coverage, test-suite size, and mutation score. According to our study, traditional test-suite metrics proposed for software testing are negatively correlated with the likelihood that a generated repair causes regressions. Therefore, improving a test-suite based on traditional test-suite metrics is beneficial both for software testing and automated program repair. Among test-suite metrics we investigate, statement coverage is shown to be most strongly correlated.
– We conduct the largest experiments to date about the correlation between test-suite quality and the performance of automated program repair (in particular, the reliability of repairs). Our subject programs contain four large-scale real-world programs. Our experimental results provide strong empirical evidences that repair quality problem is indeed quite severe (the average regression ratio of 3818 repairs repairs we obtained from GENPROG is 40%), and traditional test suite metrics can be used to control the quality of automatically generated repairs.

Apart from our main contributions, we also report other noteworthy results in this paper. We for the first time investigate the correlation between mutation score and repair quality (regression ratio of repairs). Despite the conceptual similarity of automated program repair to mutation testing, the correlation of mutation score with repair quality is not observed to be stronger than the correlation of coverage-based metrics. Our new mutation-based metric, capable-tests ratio, is observed to be more strongly correlated with the reliability of repairs in real-world subjects than mutation score. We also investigate how test-suite metrics affect repairability (repair success rate) and repair time. While we could not find a correlation pattern consistent across all subjects, positive correlations between repairability and test-suite metrics (as test-suite metrics increase, repairability increases) and negative correlations between repair time and test-suite metrics (as the test-suite metrics increase, repair time decreases) were observed in some subjects.

2 Background

2.1 Automated Program Repair

Test-driven program repair tools take as input a buggy program and a test-suite in which at least one of tests fails in the given buggy program. Such tests that fail in the buggy program are called *negative tests*, whereas those that pass are called *positive tests*. The immediate goal of test-driven repair tools is to find an edit of the buggy program that passes all the tests in the provided test-suite. The goal is achieved by first, localizing suspicious program locations, and next, modifying these suspicious parts of the program. This general framework is shared between various test-driven repair tools.[1] The main differences between these repair tools lie in how edits are constructed and how the final edit constituting a repair is searched for. For example, GenProg and SemFix, the two repair tools we use in our study, use two different repair techniques, namely search-based and constraint-based techniques, respectively. A search-based technique constructs edits by syntactically changing the program in various ways (e.g., deleting a statement or copying an existing statement in another program location), and searches for a repair using a search-based technique. Meanwhile, a constraint-based technique first constructs repair constraints that should be satisfied by a repair, and then synthesizes a repair satisfying the repair constraint using a theorem prover (typically, an SMT solver). Description about other repair tools are provided in Section 7.2.

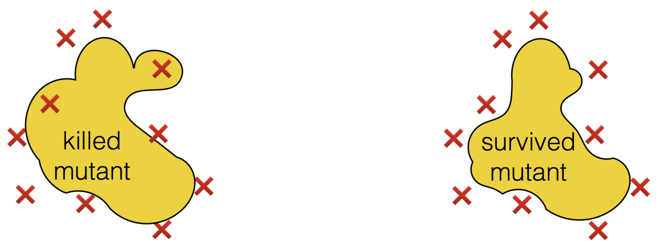
2.2 Repairs Causing Regressions

Typically, a test-suite used for a test-driven program repair tool consists of a small number of negative tests and a relatively large number of positive tests. While negative tests set a goal of repairing the given buggy program, positive tests serve as anti-goals; they filter out hazardous repair candidates, that is, those that pass negative tests but fail one of positive tests. Still, due to the incomplete nature of tests (not all desirable behaviors of software are tested), a test-driven repair tool runs the risk of generating repairs causing regressions.

2.3 On Mutation Testing and Automated Program Repair

Mutation testing is a systematic method to measure the fault detection ability of a test-suite (Jia and Harman, 2011). In mutation testing, a given program is modified (mutated) in various ways by applying mutation operators at various program locations. These modified

[1] One exception is DirectFix (Mechtaev et al, 2015) where fault localization and edit parts are fused.



(a) A killed mutant **(b)** A survived mutant

Fig. 1: The cross marks represent the tests in a test-suite. Crossing over a cross mark means that the mutant fails the corresponding test.

programs (mutants) represent potential faulty programs. Then, the fault detection ability of a test-suite TS is measured as the *mutation score* of TS, which is the ratio of the number of killed mutants[2] over the total number of non-equivalent (i.e., semantically different) mutants.

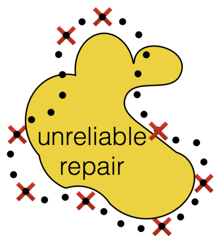
Automated program repair has similarities to mutation testing. It can be viewed that automated program repair "mutates" the original program, this time in an attempt to find a repair. As in mutation testing, mutants that fail to pass all tests in the provided test-suite are considered buggy (hence, incorrect repairs). This conceptual similarity between mutation testing and automated program repair suggests the plausibility of using the mutation score to measure the quality of a test-suite not only for mutation testing but also for automated program repair. Just as a higher mutation score is associated with a better fault-detection ability in mutation testing, it appears plausible to associate a higher mutation score with a better ability to guide a reliable repair.

There is not only similarity but also duality between mutation testing and automated program repair. As pointed out by Weimer et al (2013), "our confidence in mutant testing increases with the set of *non-redundant mutants* considered, but our confidence in the quality of a program repair gains increases with the set of *non-redundant tests*." Note that mutation score measures the non-redundancy of killed mutants, not the non-redundancy of tests capable of killing mutants. We introduce a new metric called *capable-tests ratio* in the next section that measures the non-redundancy of capable tests.

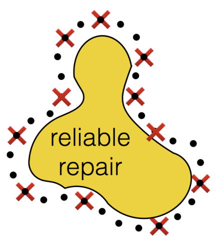
3 Research Questions

The key high-level research question of this study is whether it is possible to control the quality of automatically generated repair by controlling the quality of a test-suite. To address this question quantitatively, we investigate the *correlation* between the quality of a test-suite and the quality of an automatically generated repair. If a positive correlation is found, the use of a high-quality test suite is likely to lead to a high-quality repair. Thus, our first research question is:

[2] A mutant m is considered killed when the test result of m for at least on test in the provided test-suite is different from the test result of the original program for the same test.



(a) An unreliable repair



(b) A reliable repair

Fig. 2: The dots represent the tests in the test-universe, while the cross marks represent the tests in a test-suite used to guide automated program repair. Crossing over a dot or a cross mark means that the repair fails the corresponding test.

Research Question 1: Is there a positive correlation between the quality of a test-suite and the quality of an automatically generated repair?

However, this research question should be refined, because it does not state how to measure the quality of a test-suite and the quality of a generated repair. We first describe how we measure them, before refining the research question.

Measuring Test-Suite Quality with Traditional Metrics and Capable-Tests Ratio. We measure the quality of a test-suite using the following five kinds of test-suite metrics: (1) statement coverage, (2) branch coverage, (3) test-suite size, (4) mutation score, and (5) capable-tests ratio. All metrics except for the last one are traditional test-suite metrics. The last metric—capable-tests ratio—is a new metric we introduce in this paper to complement a potential shortcoming of the mutation score which will be described shortly. Fig. 1 pictorially describes mutation testing. The tests in a provided test-suite (represented with cross marks in the figure) guard the program against regression-causing changes (represented with a mutant that crosses over a cross mark). We hypothesize that the more the provided test-suite TS contains tests that kill one of mutants (we call such tests *capable tests*), the more likely TS can prevent regression-causing repairs. However, mutation score does not measure the percentage of capable tests in a test-suite; it only shows the percentage of killed mutants, and adding or removing tests killing no mutant does not change the mutation score of the test-suite. To complement this shortcoming of the mutation score, we introduce a new metric *capable-tests ratio* we define as the following: The capable-tests ratio of a test-suite TS is the ratio of the number of capable tests in TS, that is the number of tests that kill at least one mutant, over the total number of tests in TS.

Measuring Repair Quality. Meanwhile, we measure the quality of repair from the perspective of reliability. We deem a repair R to be reliable if there is no regression detected when testing R with its test-universe.[3] Note that the test-universe is the superset of a test-suite used to drive automated repair. In Fig. 2, both repairs (the shaded areas) are valid because they pass all tests in a given test-suite, represented with the cross marks in the figure. However, the one in Fig. 2(a) is unreliable because it fails some tests in the test-universe (the dots).

[3] Only positive tests are considered; an output change for negative tests is not a regression.

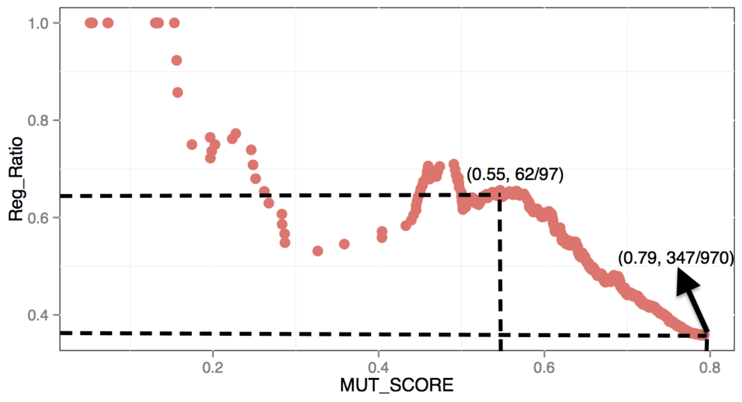


Fig. 3: A scatter plot that illustrates the correlation between the mutation score (the MUT_SCORE axis) and the regression ratio (the Reg_Ratio axis). Coordinate (0.55, 62/97) of the plot describes the following. First, among test-suites with which repairs are successfully generated, there are 97 test-suites whose mutation scores are not greater than 0.55. Second, out of these 97 cases, a regression is detected in 62 cases. Similarly, coordinate (0.79, 347/970) describes that there are 970 test-suites whose mutation scores are not greater than 0.79, and a regression is detected in 347 cases.

Based on the preceding concept of a reliable repair, we evaluate the reliability of repairs through *regression ratio*, which is computed as the number of regression-causing repairs over the total number of repairs obtained with the test-suites under investigation. Note that for each pair of a test-suite TS and a repair R generated with TS, we record whether regression is observed in R when tested against the test-universe. Our primary goal in this paper is to examine the correlations between the reliability of repairs and various test-suite metrics. We compute regression ratio at each metric score as follows. First, we collect repairs, each of which is generated with a test-suite whose metric is not greater than the score under investigation; for example, to compute the regression ratio at mutation score 0.5, we collect repairs generated with test-suites whose mutation scores are not greater than 0.5. Subsequently, we proceed to count how many of these repairs cause regressions when tested with the test-universe. Formally, the following formula is used to calculate the regression ratio at metric score s.

$$\frac{\mid \{TS \mid repaired(TS) \wedge metric \leq s \wedge regression(TU)\} \mid}{\mid \{TS \mid repaired(TS) \wedge metric \leq s\} \mid},$$

where TS and TU represent a test-suite and a test-universe, respectively (note that $TS \subseteq TU$). In the formula, predicate $repaired(TS)$ represents that a repair is successfully generated within the timeout when TS is used to guide automated program repair. Another predicate $regression(TU)$ means that a regression error is observed when testing the obtained repair with TU—in other words, $regression(TU)$ is true if there is a test $t \in TU \setminus TS$ for which the obtained repair fails. By tracking the regression ratio at different metric scores, we can retrieve the correlation between the regression ratio and a test-suite metric. Note that the lower the regression ratio is, the higher reliability of repairs.

Fig. 3 shows how the regression ratio (the Reg_Ratio axis) changes as the mutation score (the MUT_SCORE axis) changes in one of our subjects, tcas. For example, among test-suites with which repairs are successfully generated, there are 97 test-suites whose mutation score is not greater than 0.55, and a regression is detected in 62 cases out of those 97 cases. By increasing the mutation score threshold to 0.79, we can consider 873 more test-suites, in the majority of which a regression is not detected, as evidenced by a lower regression ratio there (347/970).

Now that we described how we measure the quality of a test-suite and a repair, we now refine our Research Question 1 as follows:

Research Question 1 (Refined): Is there a *negative correlation* between the *metrics* of a test-suite and the *regression ratio* of automatically generated repairs? In other words, are generated repairs *less likely* to cause regressions, as test-suite metrics increase?

Correlation analysis can not only show a general tendency about how test-suite metrics are associated with the quality of repairs, it can also be used to see which test-suite metric is most strongly associated with the quality of repairs, by comparing the correlation coefficients of different test-suite metrics. We thus ask the following research question.

Research Question 2: Which test-suite metric is most strongly correlated with the regression ratio of automatically generated repairs?

Answering this research question can be practically important. Imagine that a test-suite available for automated program repair is neither statement coverage-adequate nor branch coverage-adequate. Which would be more beneficial between improving statement coverage and branch coverage, in terms of improving the likelihood of obtaining a regression-free repair? It would be more cost-effective to improve a test-suite metric which is more strongly negatively correlated with the regression ratio of repairs.

Meanwhile, a higher-quality test suite may impose stricter restrictions on repair generation. In other words, it may be more difficult to find a repair that satisfies more stricter constraints imposed by a higher-quality test suite. We therefore evaluate whether the repairability of automated program repair is negatively correlated with test-suite metrics.

Research Question 3: Is there a *negative* correlation between the *metrics* of a test-suite and the *repairability* of automated program repair? In other words, should repairability be sacrificed in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

We compute repairability at each metric score in a similar way to how we compute regression ratio. First, we collect test-suites, each of which has a metric not greater than the score under investigation; for example, to compute the repairability at mutation score 0.5, we collect test-suites whose mutation scores are not greater than 0.5. Subsequently, we proceed to count how many of these test-suites succeed to generate a repair within the time budget. Formally, the following formula is used to calculate repairability at metric score s.

$$\frac{\mid \{TS \mid metric \leq s \wedge repaired(TS)\} \mid}{\mid \{TS \mid metric \leq s\} \mid},$$

where predicate $repaired(TS)$ represents that a repair is successfully generated within 1 hour when test-suite TS is used to guide automated program repair.

Table 1: Subjects of our experiments (6 SIR subjects in the top and 4 non-SIR subjects in the bottom); Column "LOC" shows the lines of code, column "Versions" the numbers of buggy versions, column "Test-Universe Size" the number of tests in the test universe of each subject, column "Test-Suites" the number of test suites we constructed by sampling the test universe, and column "Test-Suite Size" the range of the number of tests in the test-suite of each subject.

Subject	LOC	Versions	Test-Universe Size	Test-Suites	Test-Suite Size
tcas	173	41	1608	4100	1–100
tot_info	565	23	1052	2300	1–100
print_tokens	726	7	4130	700	1–100
print_tokens2	570	10	4115	1000	1–100
schedule	412	9	2650	900	1–100
schedule2	374	9	2710	900	1–100
php	1046K	21	200	2100	1–100
libtiff	77K	11	78	1100	1–78
grep	9.4K	5	1582	900	1–100
findutils	18K	6	82	600	1–82
Total		142	18207	14600	

In addition to repairability, repair time—the time taken to generate a repair—may be affected by the quality of a test-suite. We thus ask the following similar research question.

Research Question 4: Is there a *negative* correlation between the *metrics* of a test-suite and repair time? In other words, should more time be spent in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

4 Experimental Methodology

4.1 Subjects, Test-Universes and Test-Suites

Table 1 shows our 10 subject C programs of various sizes ranging from 173 LOC to 1046K LOC, as shown in the "LOC" column. Our subjects consist of 6 well-known Siemens programs collected from Software-artifact Infrastructure Repository (SIR) constructed by Do et al (2005), 2 real-world programs (php and libtiff) previously used to evaluate GenProg (Le Goues et al, 2012a; Weimer et al, 2013),[4] and another 2 real-world programs (grep and findutils) taken from CoREBench (Böhme and Roychoudhury, 2014).[5] We collect real-world subjects that have a large number of tests and multiple buggy versions, and are amenable for automated fix; in php and libtiff, GenProg are reported to generate repairs in many buggy versions in the previous study (Le Goues et al, 2012a). Similarly, buggy versions of grep and findutils are repaired by GenProg in our pilot experiment. The number of buggy versions of each subject is shown in the "Versions" column. Our subjects contain in

[4] We used the original GenProg benchmark. At the time of writing this paper, the benchmark was updated after a few problems in the test scripts of php and libtiff are reported in (Qi et al, 2015).

[5] The grep subject in CoREBench contains real errors unlike the grep in SIR that contains seeded errors.

total 142 buggy versions, among which the 6 SIR subjects have 99 buggy versions and the 4 non-SIR subjects have 43 buggy versions.

Each of our subjects has a relatively large number of tests, which is our test-universe. The test-universes of our large subjects consist of developer-written tests. Assuming that these tests are well-maintained (which appears to be the case), the tests in the test-universe are likely to be different from each other. In other words, the risk that some tests are identical to each other—which is undesirable because then, the training test-suite and the held-out test-suite can have the same test—is low. Meanwhile, the test-universes of our small subjects are extracted from the SIR benchmark (Do et al, 2005), which are generally considered high-quality and were used in numerous previous studies.

The "Test-Universe Size" column of Table 1 shows the total number of tests in the test-universe of each subject.[6] We construct a large number of test-suites by randomly selecting tests from these test-universes (without replacement) at each test-suite-construction iteration. The "Test-Suites" column shows how many test-suites we constructed for each subject. For each buggy version of a subject, we constructed 100 test-suites such that it contains at least one failing test case. Only in grep, we constructed 180 test-suites to collect more repairs; due to the smallest number of buggy versions of grep (5 versions), less number of repairs are obtained from grep. The size of each test-suite is chosen uniformly at random between 1 and 100, except for in libtiff and findutils where the maximum size is the size of the test-universe, that is, 78 and 82, respectively. Note that in our study, we compare experimental results across test-suite metrics, not across subjects. In each subject, we compute diverse test-suite metrics for each test-suite we construct. In total, we prepare 14600 test-suites.

We acknowledge that our simple random test-suite construction method in itself does not distinguish the effect of test-suite size and the effect of other test-suite metrics such as coverage (coverage tends to increase as the size of the test-suite increases). An alternative more controlled test-suite construction method is to construct a set of test-suites of identical size with different levels of coverage, and similarly a set of test-suites of identical coverage with different sizes, although Namin and Andrews (2009) reported in their study that it is difficult to obtain such a more ideal set of test-suites. To compensate the shortcoming of our test-suite construction method, we perform ANCOVA (analysis of covariance) and separate out the effect of coverage, similar to the work of Namin and Andrews (2009).

4.2 Automated Repair Algorithm

To obtain repairs, we mainly used GENPROG (Le Goues et al, 2012a; Weimer et al, 2013), which is also used in previous studies (Kong et al, 2015; Smith et al, 2015). We fed GENPROG with the 14600 test-suites we prepared. When running GENPROG, we used almost the same configuration parameters as those that were used in an earlier GENPROG experiment (Le Goues et al, 2012a). One noteworthy difference is that we used the deterministic repair algorithm of GENPROG (Weimer et al, 2013) to minimize randomness during experiments. All experiments were performed by distributing the load on 10 machines, each of which has two Intel Xeon E5520 2.2GHz processors and 24GB of main memory. To obtain a large number of repairs, which is essential for our study, we used relatively short timeout, 1 hour. We obtained in total 3818 repairs.

While GENPROG is the main repair tool we used, we also conducted smaller scale supplementary experiments with another repair tool SEMFIX (Nguyen et al, 2013) to see

[6] While php contains 8471 tests, we randomly selected 200 tests out of them to deal with long running time of the php tests.

whether the results we obtained extend beyond GenProg. We chose SemFix because the repair algorithm of SemFix is fundamentally different from that of GenProg. Essentially, SemFix extracts from the runs of a test-suite a set of constraints in the form of logical formulas, and subsequently solves these constraints to obtain a repair. Such a deductive style of repair of SemFix is in contrast to GenProg's generate-and-validate approach; GenProg repeats the loop of generating a repair candidate and validating it until a repair is found.

In our experiments with SemFix, we used the same test-suites as used for our main experiments with GenProg. We collected repairs using SemFix from the same SIR subjects as used in our main experiments except for tot_info which does not work with the current version of SemFix.[7] Meanwhile, testing our non-SIR subjects requires running non-trivial test-scripts written in scripting languages. To deal with these non-SIR subjects with the current version of SemFix, it is necessary to transform these test-scripts into corresponding C program statements. This is because SemFix extracts logical formulas through a symbolic-execution tool, KLEE (Cadar et al, 2008) that currently cannot handle scripting languages. Still in an attempt to deal with a large subject at least partially, we manually transformed the test-scripts of 4 versions of libtiff (i.e., 01209c9, 3af26048, d13be72c, and 0661f81).

4.3 Measuring Test-Suites Metrics

It is well known that computing mutation score typically takes a long time due to the high volume of mutants. Each and every mutant should be tested with the test suite under investigation, resulting in running the same test repeatedly for different mutants. For large programs, obtaining mutation score is particularly challenging because there are too many mutants to be tested within a reasonable time budget. To alleviate the problem, it is customary to sample parts of the mutants, and compute the mutation score only with the sampled mutants.

We measure the mutation score and the capable-tests ratio of each test-suite using Proteum (Maldonado et al, 2001).[8] To deal with enormous size of mutants generated from the 4 large subjects (php, libtiff, grep, findutils), we randomly sampled 1 – 3% of the total mutants, using the options Proteum provides. Although we do not distinguish equivalent mutants, note that the same mutant samples of program P are used across all test-suites for P in our experiments. Thus, the mutation scores of these test-suites are affected at the same rate by equivalent mutants that may exist, making the correlations between mutation scores of these test-suites and the reliability of repairs unaffected accordingly. Meanwhile, to measure the statement and branch coverage of our test-suites, we use `gcov`.[9] When running GenProg or SemFix, it is necessary to mark the source file(s) allowed to be repaired. Our measurements of mutation score and statement/branch coverage are performed on these marked files.

5 Experimental Results

In this section, we outline the results from our experiments with the repair tools GenProg and SemFix. We first present the results from our main experiments performed with GenProg. The results from SemFix is presented in Section 5.5.2.

[7] tot_info includes non-linear arithmetic expressions which are not currently supported by the underlying SMT solver SemFix uses.

[8] We extended its parser to handle the large subjects (php, libtiff, grep, and findutils).

[9] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Table 2: GenProg experiments: statistics for repairs and regressions

Subjects	Test-Suites	Repairs	Repair Ratio	Regressing	Regression Ratio
tcas	4100	972	24 %	348	36 %
tot_info	2300	137	6 %	17	12 %
print_tokens	700	28	4 %	0	0 %
print_tokens2	1000	235	24 %	9	4 %
schedule	900	37	4 %	37	100 %
schedule2	900	108	12 %	53	49 %
php	2100	1666	79 %	915	55 %
libtiff	1100	313	28 %	42	13 %
grep	900	128	14 %	41	32 %
findutils	600	194	32 %	83	43 %
Total	14600	3818	26 %	1545	40 %

5.1 Basic Statistics – Repair Ratio and Regression Ratio

Before investigating our research questions, we first present Table 2 which shows the basic statistics for our experiment such as how often repairs are generated (repair ratio) and how often regressions are observed (regression ratio). The “Test-Suites” column shows the number of test-suites in each subject, and the “Repairs” column the total number of obtained repairs for each subject. We obtain repairs by running GenProg for maximum 1 hour. In total, we obtained 3818 repairs out of 14600 trials, resulting in average repair ratio of 26%. The repair ratio of each subject is defined as the ratio of the total number of obtained repairs (available in the “Repairs” column) over the total number of repair trials (available in the “Test-Suites” column). Note that the total number of repair trials is equivalent to the number of test-suites, because we initiate a separate repair session for each test-suite.

For the obtained repairs, we investigate whether regressions are observed. We run each repaired program against its test universe, and observe whether regressions occur. That is, if a repaired program fails any of the previously passing tests in the test universe, we consider that a regression occurs. The “Regressing” column shows the number of repairs for which regressions are observed. For example, in tcas, out of the 972 repairs obtained, 348 of them are observed to be regression-causing repairs. The “Regression Ratio” column of Table 2 shows the regression ratio in each subject, which is defined as the ratio of the number of regression-causing repairs (available in the “Regressions” column) over the total number of repairs (available in the “Repairs” column). For example, the regression ratio in tcas is 348/972, which is about 36%. The overall regression ratio ranges from 0% of print_tokens to 100% of schedule. The average regression ratio of all subjects is 40%, as shown in the “Total” row.

Ideally, an automated program repair tool should generate a repair as often as possible (which can be indicated by a high repair ratio), and the generated repair should be regression-free as much as possible (which can be indicated by a low regression ratio). Table 2 indicates that the current repair tool does not achieve these goals yet. The overall repair ratio is as low as 26%, while the overall regression ratio is as high as 40%. In subject schedule, the repair ratio is only 4% and all generated repairs cause regressions. In subject php, while the repair ratio is relatively high reaching 79%, regression ratio is also quite high (55%).

Table 3: GENPROG experiments: correlations between the regression ratio and various test-suite metrics

Subject	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	-0.92	78%	100%	95%	-0.84	39%	95%	90%	-0.87	2	100	60	-0.93	0.05	0.79	0.67	-0.03	0.42	1.00	0.65
print_tokens2	-1*	7%	8%	7%	0.65	3%	4%	3%	-0.88	2	100	59	-0.80	0.24	0.83	0.79	0.51	0.33	1.00	0.56
tot_info	-0.89	76%	97%	95%	-0.88	67%	92%	89%	-0.84	3	99	56	-0.86	0.51	0.88	0.80	0.91	0.19	1.00	0.49
schedule2	-0.4	98%	99%	99%	0.12*	81%	94%	90%	0.48	16	100	67	0.83	0.67	0.73	0.70	0.41	0.23	0.90	0.38
php	-0.65	0%	89%	22%	-0.75	0%	50%	13%	-0.88	1	100	50	-0.06	0.00	1.00	0.45	-0.7	0.00	1.00	0.3
libtiff	-0.7	9%	31%	20%	-0.73	6%	23%	15%	-0.77	1	71	19	-0.44	0.00	1.00	0.43	-0.47	0.00	1.00	0.82
grep	-0.92	36%	68%	51%	-0.85	22%	53%	36%	-0.62	4	93	18	-0.51	0.01	0.10	0.02	-0.81	0.73	1.00	0.95
findutils	-0.95	2%	33%	22%	-0.95	1%	22%	15%	-0.86	1	56	18	-0.78	0.00	0.54	0.18	-0.82	0.00	1.00	0.65
Average	-0.84	34%	70%	51%	-0.83	23%	56%	43%	-0.66	4	90	43	-0.64	0.13	0.84	0.55	-0.12	0.24	0.99	0.6

The topmost column shows the 5 test-suite metrics we investigate. The "r" column of each test-suite metric shows the correlation coefficient between the regression ratio and the corresponding metric in Pearson's r. Negative correlation coefficients, shaded in the table, imply that regressions are less observed as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites. All correlation coefficients shown in the table are statistically significant at the 0.05 level except those asterisked.

5.2 Correlation Coefficients about Regression Ratio

Our first research question involves correlation between test-suite metrics and the regression ratio of repairs.

Research Question 1: Is there a *negative correlation* between the *metrics* of a test-suite and the *regression ratio* of automatically generated repairs? In other words, are generated repairs *less likely* to cause regressions, as test-suite metrics increase?

Table 3 shows the correlations between the regression ratio and various metrics of test-suites, that is, statement coverage, branch coverage, test-suite size, mutation score, and capable-tests ratio. For each metric, the "r" column shows Pearson's product moment correlation coefficients (Pearson's r) (Pearson, 1895) rounded to two decimal places. The correlation coefficients of print_tokens and schedule are not available because in our experiments, repairs for these subjects either always caused regressions (in the case of schedule) or always did not cause regressions (in the case of print_tokens). All correlation coefficients shown in Table 3 are statistically significant at the 0.05 level except those asterisked. In Table 3, we also show the minimum/maximum and mean values of each metric (under the Min, Max, and Mean columns, respectively) of our randomly constructed test-suites.[10] For example, the test-suites of tcas has on average 95% statement coverage ranging between 78% and 100%. Note that these min/max/mean values are retrieved only from the test-suites that successfully guided repairs, excluding test-suites with which no repair is found within timeout; for these excluded test-suites, the regression ratio of repairs cannot be investigated.

Encouraging Results of Traditional Metrics. In Table 3, negative correlation coefficients are shown in shades. A negative correlation coefficient of a metric M implies that as the value of M increases, the regression ratio tends to decrease; in other words, the reliability of repairs tends to increase. In general, negative correlations are observed across all traditional metrics we investigated—statement coverage, branch coverage, test-suite size, and mutation score. In particular, statement coverage consistently shows negative correlations across all subjects. Similarly, the other traditional test-suite metrics also show negative correlations in the majority of subjects. Our results suggest that the traditional test-suite metrics can also be effectively used to control the regression rate of automatically generated repairs.

As the traditional test-suite metrics (statement coverage, branch coverage, test-suite size, and mutation score) increase, the regression rate of automatically generated repairs generally decreases, showing the promise of using the traditional test-suite metrics to control the regression ratio of automatically generated repairs.

Our finding implies that the efforts to improve test-suites for the purpose of testing—which is already practiced in the industry—can also benefit automated program repair. Note that our main finding is consistently observed across real-world large-scale software and controlled small-scale subjects (SIR subjects).

Discouraging Results of Capable-Tests Ratio. On the contrary to the traditional test-suite metrics, the results from capable-tests ratio are discouraging. The expected negative correlations are observed only in large real-world subjects (php, libtiff, grep, and findutils). In

[10] The minimum statement/branch coverage of php is 0 because some tests do not execute the marked source files.

Table 4: Average rankings of test-suite metrics

Metric	Statement Coverage	Branch Coverage	Test-Suite Size	Mutation Score	Capable-Tests Ratio
Avg. Ranking	1.75	2.5	2.75	4	3.875

all the small subjects except tcas, positive correlations are observed. Capable-tests ratio does not seem as useful as the traditional metrics in controlling the quality of generated repairs.

Next we compare correlation coefficients of different test-suite metrics to investigate our second research question.

Research Question 2: Which test-suite metric is most strongly correlated with the regression ratio of automatically generated repairs?

To investigate this research question, we rank test-suite metrics in each subject in order of the correlation coefficients. The metric whose correlation coefficient is the smallest is ranked first in each subject. For example, in tcas, mutation score is ranked first, statement coverage is ranked second, test-suite size is ranked third, and so on. Table 4 shows the average ranking of each test-suite metric. Statement coverage has the highest average ranking. Indeed, statement coverage is ranked first in 5 subjects (print_tokens2, tot_info, schedule2, grep, and findutils) out of total 8 subjects, and ranked second (tcas) in one subject. Also, only statement coverage consistently shows a negative correlation across all subjects.

In our experiments, statement coverage is, on average, more strongly correlated with regression ratio than other metrics we investigate. Our results suggest that to reduce the regression ratio, increasing statement coverage is more promising than improving the other test-suite metrics.

Implication and Limitation of Correlation. It should be noted that the highest correlation of statement coverage does not necessarily imply that a 100% statement coverage-adequate test-suite is most effective in controlling the reliability of repairs. A correlation between A and B only shows how B tends to change as A changes, or vice versa. In fact, covering a buggy statement may not be enough to reveal the bug, and this is why more sophisticated coverage such as branch coverage is more commonly advocated in software testing. Our finding only implies that return on investment tends to be higher in statement coverage than in other metrics. In other words, a practical implication of our finding is that if the currently available test-suite is neither statement coverage-adequate nor branch coverage-adequate, improving the statement coverage of the test-suite would improve the reliability of repairs more effectively than improving branch coverage.

In Table 4, mutation score has the lowest average ranking. Mutation score is ranked even lower than another mutation-based metric, capable-tests ratio, although the average ranking gap between these two metrics is marginal. Mutation score is ranked last in 5 subjects (schedule2, php, libtiff, grep, and findutils) out of total 8 subjects. A possible reason for the low ranking of mutation score is that the mutants used in mutation testing are sampled *evenly from all possible mutants*, whereas in automated program repair, repair edits are performed *only on suspicious program locations* (the suspicious locations are identified through the fault localization step of automated program repair). When the mutant sampling rate is 100% as in the case of our small subjects, a mutant M can be sampled at a non-suspicious program location L in which a program repair tool does not generate a repair candidate. Even if M is killed, the increase of the mutation score has no direct bearing on improving the reliability of a repair in this case, because no repair candidate is generated at L. In other words, M is not likely to represent unreliable repairs. Meanwhile, when the mutant sampling rate is low as in the case of our large subjects, the chance that a mutant is sampled at suspicious program locations is also low. Another possible reason for the low ranking of mutation score is the discrepancy between mutation operators and repair operators. Given that the mutation scores can change depending on which mutation operators are used (Yao et al, 2014), using selective mutation operators—instead of all mutation operators—may change the correlation coefficients.

Comparison with Test-Suite Size. Notice that mutation score and capable-tests ratio show lower average ranking than test-suite size, whereas statement coverage and branch coverage show higher average rankings than test-suite size. This result suggests that increasing statement coverage or branch coverage is likely to reduce regression ratio more effectively than blindly adding arbitrary tests into the test suite. To further investigate whether (a) improving coverage indeed influences the reduction of regression ratio in the statistical sense or (b) regression ratio reduces merely because the test-suite size increases, we perform ANCOVA (analysis of covariance). When performing ANCOVA with statement coverage and test-suite size, the p-value of the statement coverage is less than 0.05 in all subjects except in php where the p-value is 0.057. This indicates that the influence of statement coverage on regression ratio is, in general, statistically significant. Meanwhile, the interaction effect between statement coverage and test-suite size is not as significant. The p-value of the interaction effect is statistically insignificant (> 0.05) in tot_info, php, libtiff, grep, and findutils. The result on branch coverage is similar. The p-value of the branch coverage is less than 0.05 in all subjects except in print_tokens2 where the p-value is 0.075. The p-value of the interaction effect between branch coverage and test-suite size is statistically insignificant (> 0.05) again in tot_info, php, libtiff, grep, and findutils.

5.3 Correlation Coefficients about Repairability

Next we report our results on repairability. Our research question regarding repairability is:

Research Question 3: Is there a *negative* correlation between the *metrics* of a test-suite and the *repairability* of automated program repair? In other words, should repairability be sacrificed in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

Table 5 shows the correlations between the repairability and various metrics of test-suites. Pearson's correlation coefficients are shown in the table with negative coefficients being highlighted.[11] The overall correlation patterns are different between the small SIR subjects and the large real-world subjects. In the small subjects, positive correlations are observed more often than negative correlations across traditional test-suite metrics (statement/branch coverage, test-suite size, and mutation score); test-suite size and mutation score

[11] Min/Max/Mean values of the table are different from those of Table 3, because there we consider only test-suites from which repairs are generated, whereas in Table 5, we consider all test-suites.

Table 5: GENPROG experiments: correlations between repairability (repair success rate) and various test-suite metrics

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	*r*	Min	Max	Mean	*r*	Min	Max	Mean	*r*	Min	Max	Mean	*r*	Min	Max	Mean	*r*	Min	Max	Mean
tcas	0.55	45%	100%	94%	0.88	14%	95%	85%	0.92	2	100	54	0.97	0.00	0.83	0.65	-0.03	0.00	1.00	0.68
print_tokens	0.13	53%	97%	90%	0.24*	39%	93%	82%	0.64	1	100	49	0.78*	0.35	0.85	0.80	0.72	0.37	1.00	0.62
print_tokens2	-0.82	7%	8%	7%	-0.4*	3%	4%	3%	0.87	1	100	52	0.26	0.02	0.84	0.79	0.38	0.02	1.00	0.60
tot_info	0.71	65%	98%	94%	0.87	60%	93%	88%	0.87	1	99	52	0.92	0.42	0.90	0.82	-0.19	0.17	1.00	0.50
schedule	0.33	30%	99%	96%	0.43	14%	95%	86%	0.93	1	100	49	0.71	0.00	0.89	0.81	0.42	0.00	1.00	0.47
schedule2	0.31	66%	99%	99%	0.71	49%	96%	88%	0.99	1	100	54	0.65	0.28	0.77	0.69	0.26	0.20	1.00	0.45
php	0.73	0%	89%	22%	0.76	0%	50%	13%	0*	1	100	50	-0.08	0.00	1.00	0.44	0.81	0.00	1.00	0.28
libtiff	-0.31	6%	31%	23%	-0.19	4%	23%	17%	-0.93	1	77	29	-0.33	0.00	1.00	0.47	0.83	0.00	1.00	0.81
grep	-0.99	36%	73%	60%	-0.99	22%	58%	45%	-0.91	4	100	43	-0.46	0.00	0.77	0.02	-0.69	0.00	1.00	0.58
findutils	-0.94	2%	36%	28%	-0.94	1%	24%	19%	-0.99	1	81	40	-0.66	0.00	0.61	0.20	-0.22	0.00	1.00	0.70
Average	-0.03	31%	73%	61%	0.19	20%	67%	55%	0.27	1	95	47	0.22	0.08	0.84	0.54	0.23	0.08	1.00	0.57

The topmost column shows the 5 test-suite metrics we investigated from statement coverage to capable-tests ratio. The "r" column of each test-suite metric shows the correlation coefficient between the repairability and the corresponding metric in Pearson's r. All shown coefficients are statistically significant at the 0.05 level except those asterisked. Negative correlation coefficients, shaded in the table, imply that less repairs are obtained as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites.

Table 6: Mean and max time of successful repairs with a one-hour timeout

Subject	Repair Time	
	Mean	Max
tcas	2.7 m	14.1 m
print_tokens	6.9 m	48.3 m
print_tokens2	0.6 m	29 m
tot_info	1.4 m	4.9 m
schedule	3 m	24 m
schedule2	1 m	13 m

Subject	Repair Time	
	Mean	Max
php	7.2 m	56 m
libtiff	14.1 m	57.6 m
grep	24 m	59.6 m
findutils	11.6 m	59.5 m

are positively correlated with repairability across all small subjects, and statement/branch coverage is also positively correlated in the majority of the small subjects. This implies that as test-suite metrics increase, it is more likely for a repair to be generated automatically in the small subjects. Meanwhile, the opposite pattern is observed in the large subjects. There, negative correlations are observed across the same traditional test-suite metrics as the preceding; mutation score is negatively correlated with repairability across all large subjects, and the remaining traditional test-suite metrics (statement/branch coverage and test-suite size) are also negatively correlated except for in php.

Our experimental data suggest a possibility that a high-score test-suite helps find a repair in small programs. One possible explanation is that the use of a higher quality test-suite for statistical fault localization tends to localize faulty program locations more precisely as reported in previous studies (Artzi et al, 2010; Baudry et al, 2006), and knowing where to fix is a big advantage when fixing a program. However, it can also be more difficult to satisfy more tests, which makes a negative impact on repairability. Depending on the situations in which repair takes place, test-suite may impact on repairability positively or negatively. We conjecture that our inconclusive result on repairability may be due to interaction effects. For example, repairability may be affected significantly by failing-tests ratio (the proportion of failing tests in a test-suite), and the interaction between failing-tests ratio and test-suite metrics may cause the observed inconclusive result. We leave the investigation of this conjecture as future work.

Our experimental results are inconclusive about the correlation between test-suites and repairability. However, we note that increasing test-suite metric does not always decrease repairability. Im some subjects, positive correlations were observed between test-suite metrics and repairability, indicating that as the test-suite metrics increase, repairability tends to increase.

5.4 Correlation Coefficients about Repair Time

Our last research question involves repair time.

Research Question 4: Is there a *negative* correlation between the *metrics* of a test-suite and repair time? In other words, should more time be spent in an attempt to obtain a higher-quality repair via a higher-quality test-suite?

Table 7: GENPROG experiments: correlations between repair time and various test-suite metrics

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	0.92	78%	100%	95%	0.77	39%	95%	90%	0.99	2	100	60	0.85	0.05	0.79	0.67	-0.88	0.42	1.00	0.65
print_tokens	-0.96	74%	95%	88%	-0.94	62%	92%	79%	-0.98	9	100	50	-0.79	0.76	0.86	0.82	0.78	0.46	0.94	0.62
print_tokens2	-1*	7%	8%	7%	0.04*	3%	4%	3%	-0.57	2	100	59	-0.37	0.24	0.83	0.79	0.75	0.33	1.00	0.56
tot_info	0.68	76%	97%	95%	0.73	67%	92%	89%	0.95	3	99	56	0.9	0.51	0.88	0.80	-0.84	0.19	1.00	0.49
schedule	-0.66*	95%	99%	97%	-0.75	83%	93%	89%	-0.45*	8	100	64	-0.89	0.74	0.86	0.84	-0.48	0.21	1.00	0.40
schedule2	0.84*	98%	99%	99%	0.92	81%	94%	90%	0.93	16	100	67	$0.94^\dagger$	0.67	0.73	0.70	0.24	0.23	0.90	0.38
php	-0.36	0%	89%	22%	-0.29	0%	50%	13%	-0.74	1	100	50	0.53	0.00	1.00	0.44	-0.78	0.00	1.00	0.30
libtiff	0.98	9%	31%	20%	0.99	6%	23%	15%	0.87	1	71	19	0.75	0.00	1.00	0.43	0.61	0.00	1.00	0.82
grep	0.85	36%	68%	51%	0.95	22%	53%	36%	0.76	4	93	18	$0.65^\dagger$	0.00	0.10	0.01	0.66	0.00	1.00	0.58
findutils	0.86	2%	33%	22%	0.87	1%	22%	15%	0.86	1	56	18	0.6	0.00	0.54	0.18	0.67	0.00	1.00	0.65
Average	0.42	39%	73%	56%	0.36	40%	68%	57%	0.34	4	91	44	0.2	0.29	0.84	0.62	0.07	0.18	0.98	0.54

The topmost column shows the 5 test-suite metrics we investigated from statement coverage to capable-tests ratio. The “r” column of each test-suite metric shows the correlation coefficient between the repairability and the corresponding metric in Pearson’s r. All shown coefficients are statistically significant at the 0.05 level except those asterisked. Negative correlation coefficients, shaded in the table, imply that less time tends to be taken to obtain a repair, as the metric increases. In the “Average” row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites.

Table 6 shows the mean and max time taken to generate repairs in each subject. Repair time of small subjects (shown in the left-hand side table) is generally smaller than the repair time of large subjects (shown in the right-hand side table). Table 7 shows the correlation between repair time and test-suite metrics. Similar to the case of repairability, no conclusive pattern is observed.

Our experimental results are inconlusive about the correlation between test-suites and repair time. However, we note that increasing test-suite metric does not always increase repair time. In some subjects, negative correlations were observed between test-suite metrics and repair time, indicating that as the test-suite metrics increase, repair time tends to decrease.

5.5 Generalizing the Results

To mitigate external threats to our results, we perform the following. First, we replace Pearson's correlation coefficients shown earlier with Kendall's rank correlation coefficients, and see if similar results are observed (Section 5.5.1). Second, we replace GenProg, an automated program repair tool used in our experiments, with another program repair tool, SEMFIX (Nguyen et al, 2013), and see if similar results are observed (Section 5.5.2).

5.5.1 Different Correlation Coefficient: Kendall Rank Correlation Coefficient

To investigate how our results are affected by the use of different kinds of correlation coefficients, Table 8 shows the correlation coefficients between regression ratio and test-suites in Kendall's rank correlation coefficients. We use Kendall's τ_b to handle tied ranks (Kendall, 1945). Despite the changes of correlation coefficients, the overall pattern remains similar. As in our previous analysis, we find that:

– Negative correlations are generally observed across all traditional metrics.
– Statement coverage is, on average, most strongly correlated with regression ratio. The average ranking of test-suite metrics is ordered as follows: statement coverage (2.25) $\leq$ test-suite size (2.25) $\leq$ branch coverage (3) $\leq$ mutation score (3.25) $\leq$ capable-tests ratio (4.25), where the numbers in parentheses show the average ranking of the corresponding metrics.
– Coverage-based metrics generally show stronger correlation with regression ratio than mutation-based metrics.
– In modularized real-world software, capable-tests ratio is shown to be negatively correlated with regression ratio.

5.5.2 Different Repair Algorithm: SEMFIX

Apart from the main experiments performed with GENPROG, we conducted supplementary experiments with another repair tool, SEMFIX. Note that SEMFIX takes a fundamentally different repair approach from GENPROG; while GENPROG repeats to run each repair candidate until all available tests pass, SEMFIX first constructs repair constraints that should be satisfied by a repair and synthesizes a repair satisfying the repair constraint using a theorem prover. Meanwhile, the differences in fault localization employed in these two tools are not

Table 8: Correlations between the regression ratio and various test-suite metrics (Kendall's τ_b)

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean	τ_b	Min	Max	Mean
tcas	-0.89	78%	100%	95%	-0.64	39%	95%	90%	-0.82	2	100	60	-0.96	0.05	0.79	0.67	-0.12	0.42	1.00	0.65
print_tokens2	-1*	7%	8%	7%	0.98	3%	4%	3%	-0.97	2	100	59	-0.99	0.24	0.83	0.79	0.25	0.33	1.00	0.56
tot_info	-0.86	76%	97%	95%	-0.92	67%	92%	89%	-0.99	3	99	56	-0.82	0.51	0.88	0.80	0.49	0.19	1.00	0.49
schedule2	-0.71	98%	99%	99%	-0.08*	81%	94%	90%	0.51	16	100	67	0.82	0.67	0.73	0.70	0.47	0.23	0.90	0.38
php	-0.44	0%	89%	22%	-0.36	0%	50%	13%	-0.83	1	100	50	0.27	0.00	1.00	0.45	-0.74	0.00	1.00	0.3
libtiff	-0.95	9%	31%	20%	-0.94	6%	23%	15%	-0.99	1	71	19	-0.58	0.00	1.00	0.43	-0.45	0.00	1.00	0.82
grep	-0.98	36%	68%	51%	-0.99	22%	53%	36%	-0.97	4	93	18	-0.93	0.01	0.10	0.02	-0.71	0.73	1.00	0.95
findutils	-0.87	2%	33%	22%	-0.88	1%	22%	15%	-0.89	1	56	18	-0.9	0.00	0.54	0.18	-0.73	0.00	1.00	0.65
Average	-0.83	34%	70%	51%	-0.79	23%	56%	43%	-0.74	4	90	43	-0.66	0.13	0.84	0.55	-0.19	0.24	0.99	0.6

The topmost column shows the 5 test-suite metrics we investigate. The "τ_b" column of each test-suite metric shows the correlation coefficient between the regression ratio and the corresponding metric in Kendall's τ_b. Negative correlation coefficients, shaded in the table, imply that regressions are less observed as the metric increases. In the "Average" row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites. All correlation coefficients shown in the table are statistically significant at the 0.05 level except those asterisked.

Table 9: SemFix experiments: correlations between the regression ratio and various test-suite metrics

	Statement Coverage				Branch Coverage				Test-Suite Size				Mutation Score				Capable-Tests Ratio			
Subject	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean	r	Min	Max	Mean
tcas	-0.9	78%	100%	95%	-0.62	39%	95%	90%	-0.79	2	100	59	-0.73	0.05	0.79	0.67	-0.01	0.42	1.00	0.646
print_tokens2	-1*	7%	8%	7%	0.46*	3%	4%	3%	-0.69	4	100	60	-0.97	0.49	0.83	0.80	0.66	0.36	1.00	0.554
schedule2	-0.35*	98%	99%	99%	0.64*	82%	93%	90%	-0.3*	16	100	68	0.86	0.67	0.73	0.70	-0.53	0.23	0.69	0.374
libtiff	-0.82	6%	31%	23%	-0.81	4%	23%	17%	-0.87	1	74	27	-0.37	0.00	1.00	0.45	-0.83	0.00	1.00	0.824
Average	-0.77	47%	59%	56%	-0.08	32%	54%	50%	-0.66	5.75	93.5	53	-0.30	0.30	0.84	0.66	-0.18	0.25	0.92	0.6

The topmost column shows the 5 test-suite metrics we investigate. The “r” column of each test-suite metric shows the correlation coefficient between the regression ratio and the corresponding metric in Pearson’s r. Negative correlation coefficients, shaded in the table, imply that regressions are less observed as the metric increases. In the “Average” row, the coefficients of each metric is averaged across all subjects. For each metric, we also show the minimum/maximum/mean values of our test-suites. All correlation coefficients shown in the table are statistically significant at the 0.05 level except those asterisked.

Table 10: Average rankings of test-suite metrics (SemFix)

Metric	Statement Coverage	Branch Coverage	Test-Suite Size	Mutation Score	Capable-Tests Ratio
Avg. Ranking	1.5	4	2.75	3.75	3

as significant. Statistical fault localization (Jones et al, 2002; Liblit et al, 2003) is employed in both tools. Still, the concrete fault localization techniques employed in these two tools are not exactly identical with each other.

Note that the objective of this additional SEMFIX experiments is to see if our findings obtained from the GENPROG experiments also hold when a different repair algorithm is used. We emphasize that comparing the performance of GENPROG and SEMFIX is not the purpose of this study. In fact, comparing correlation coefficients between the two tools does not determine the winner. One tool may show a stronger correlation with the test-suite quality than the other tool, while it still generates regression-causing repairs more frequently.

Table 9 shows the correlations between the regression ratio observed in the SEMFIX experiments and various test-suite metrics. We used the same test-suites as used in our GENPROG experiments. In general, the results are similar to those obtained from the GENPROG experiments. As in the GENPROG experiment, negative correlations between regression ratio and the traditional test-suite metrics (statement coverage, branch coverage, test-suite size, and mutation score) are observed in the majority of cases. In particular, statement coverage and test-suite size are negatively correlated with regression ratio in all subjects. Mutation score shows negative correlations except in schedule2, similar to the GENPROG experiment. While branch coverage shows positive correlations in two subjects (print_tokens2 and schedule2), these results are not statistically significant ($p > 0.05$).

Similar to the GENPROG experiment, we compute the average ranking of each test-suite metric and show the result in Table 10. Recall that in each subject, the metric whose correlation coefficient is the smallest is ranked first. Statement coverage again is ranked highest as in our GENPROG experiment. The rest of the metrics are, on average, ranked lower than test-suite size.

Our experimental results from SEMFIX generally coincide with our finding from the GENPROG experiment, despite the differences in repair algorithms and fault localization techniques. The traditional test-suite metrics are, overall, negatively correlated with regression ratio, similar to our GENPROG experimental results. In particular, statement coverage is again shown to be most strongly correlated with regression ratio.

6 Threats to Validity

External: Subjects, Test Universes, Mutants, and Repair Tools. Our findings may not generalize to other subjects, although our subjects consist of various software projects of different sizes, extracted from diverse sources (SIR, GenProg, and CoREBench) that contain seeded bugs (SIR), actual bugs (GenProg), and actual regression bugs (CoREBench). Similarly, our test universes may not be representative of the whole test case population, which is theoretically infinite. In general, the larger a test universe is, the more likely regressions are observed when testing a repaired program. To mitigate this threat, we selected subjects that have a large number of test cases.

Similarly, the use of 1-hour timeout also threatens the external validity of our experimental results. Results may vary depending on which timeout is used in the experiments. The external validity of our mutants are also similarly threatened, because in large subjects, we randomly sampled 1–3% of mutants to be able to deal with the large size of the mutant population (for SIR subjects whose sizes are smaller, we used the whole mutant population). The relatively weak correlation between mutation score and regression ratio as compared to other test-suite metrics may be due to the difference between mutation testing and automated program repair. Repair candidates generated from an automated program repair tool are not necessarily identical with or similar to mutants generated from a mutation testing tool. Also, an automated program repair tool modifies only suspicious program locations, whereas mutation testing does not consider the suspiciousness of program locations when sampling mutants. Our results obtained with randomly sampled mutants is, despite its limitations, still interesting from practical point of view, because mutant sampling is a common approach taken in mutation testing to deal with a large number of mutants practically. One way to mitigate the threats posed by sampled mutants is to change the sampling rate and see if similar results are maintained. We leave this investigation as future work.

Lastly, a repair tool affects the experimental results, and different results may be obtained when a different repair tool is used. To mitigate this threat, we also conducted an additional experiment with SEMFIX, and observed that the overall results are similar to the results from our main experiment. Note that SEMFIX uses a fundamentally different repair approach from GENPROG used for our main experiment.

Internal: Correctness of Tools. Our findings are based on the raw data generated by various tools, i.e, GCOV, GENPROG, SEMFIX, and PROTEUM, where the latter three tools are research prototypes. We also modified PROTEUM because the original PROTEUM cannot handle any of our non-SIR subjects. In order not to exacerbate this threat, our modification to PROTEUM is minimally restricted to its parser.

7 Related Work

7.1 Empirical Studies on Automated Program Repair and Test Suites

Automatically generated repairs cause regressions essentially because the space of plausible repairs (repairs that pass all tests of a given test-suite) is larger than the space of correct repairs (Long and Rinard, 2016a). Obviously, not all repairs that pass a given test-suite are correct repairs. After all, automatically generated repairs can be overfit to the test-suite available to a repair system (Smith et al, 2015). To prevent more efficiently a repair system from generating incorrect repairs, a stronger test-suite is necessary (Long and Rinard, 2016a). However, Long and Rinard do not investigate *how* to improve a test-suite. Meanwhile, Smith et al. report a positive correlation between test-suite *size*[12] and the reliability of repairs (Smith et al, 2015). Their experiments are conducted with small student programs ($\leq$ 23 LoC), and other test-suite metrics such as statement/branch coverage and mutation score are not investigated. More recently, Kong et al. investigate 9 SIR subjects and report a similar correlation between test-suite size and the reliability of repairs (Kong et al, 2015).

[12] The "coverage" referred to in (Smith et al, 2015) essentially means how many tests of a given test-universe are covered.

Another small-scale empirical study was performed by Assiri and Bieman with 4 SIR subjects ($\leq$ 565 LOC) (Assiri and Bieman, 2014) . One of their main findings is that branch coverage-adequate test-suites tend to be more effective in controlling regression errors than statement coverage-adequate test-suites. As discussed in Section 5.2, our result that statement coverage is most strongly correlated with regression ratio should not be confused with comparing coverage-adequate test suites. Our result implies that when a test-suite is neither statement coverage-adequate nor branch coverage-adequate, improving the statement coverage of the test-suite is more likely to improve the reliability of repairs than improving branch coverage. Meanwhile, if there are already a statement coverage-adequate test-suite and a branch coverage-adequate test-suite, using the branch coverage-adequate test-suite appear to be better (Assiri and Bieman, 2014). However, this result is obtained from 4 small SIR subjects, and whether this result can be extended to large real-world software is not shown. In fact, test-suites of real-world software are usually not coverage-adequate, which is also the case for our real-world subjects.

We for the first time perform a correlation study between various test-suite metrics and the reliability of generated repairs. Large real-world software and their test-suites are investigated in our correlation study unlike previous correlation studies.

Apart from the aforementioned studies on test-suite quality, the quality of an individual test can also affect the quality of automatically generated repair. For instance, if a test oracle is weak, a generated repair may not fix the bug manifested by the test (Qi et al, 2015). To improve the quality of generated repairs, both the quality of the individual test and the quality of the test-suite should be improved.

The quality of a test-suite have long been the subject of research in software testing. There is a large body of research on identifying/generating/maintaining effective and efficient test-suites, in particular in terms of bug finding (Andrews et al, 2006; Cadar et al, 2008; Cadar and Engler, 2005; Godefroid et al, 2005; Miller and Spooner, 1976; Shoenauer and Xanthakis, 1993). More recently, other applications of test-suites other than bug finding such as fault localization have attracted the attention of researchers, and thereafter appropriate attributes of these unconventional applications have been studied (Baudry et al, 2006; Artzi et al, 2010).

While improving the quality of the test-suite provides a *proactive* measure to improve the repair quality, regression test generation techniques can provide a *reactive* measure (Böhme et al, 2013a,b; Person et al, 2011; Santelices et al, 2008). A generated repair induces program changes. These changes can be stress-tested automatically with regression test generation techniques that are specifically directed towards these repair-related changes. In future, we envision automated repair techniques that integrate regression test generation techniques to detect regression-introducing repairs and simultaneously improve the quality of the test-suite.

7.2 Automated Program Repair Approaches

Automated program repair approaches can be broadly classified into a search-based approach and a constraint-based approach. The search-based repair approach, also often called the generate-and-validate approach, navigates a set of repair candidates explicitly through a search algorithm such as genetic programming, whereas the constraint-based approach constructs repair constraints that should be satisfied by a repair and symbolically searches for a repair satisfying the repair constraint using a theorem prover (typically, an SMT solver). GenProg (Weimer et al, 2013; Le Goues et al, 2012b) and SemFix (Nguyen et al, 2013), used in our experiments, are first repair systems employing the search-based and constraint-based repair approach, respectively.

More recent search-based repair systems include RSRepair, SPR, Prophet, SearchRepair, and a system of Debroy and Wong. The system of Debroy and Wong (2014) uses mutation operators to generate a repair. RSRepair (Qi et al, 2014) employs a random search algorithm to search for a repair. SPR (Long and Rinard, 2015) employs an efficient staged repair search algorithm (instead of directly editing a program, SPR first checks whether a certain class of edits defined by an edit schema contains a repair, and generates an edit only when a repair can be instantiated from the edit schema). Prophet (Long and Rinard, 2016b) employs a machine learning algorithm to prioritize a patch candidate similar to human patches. The idea of using human patches was first suggested in PAR (Kim et al, 2013) where patch templates commonly observed in human patches are manually extracted and used to generate patches. SearchRepair (Ke et al, 2015) searches a large database of code fragments for ones that are semantically similar to the defective code fragment and uses the found fragments to generate a repair for the defect. A related idea is to extract patterns of bad patches generated from automated repair systems (e.g., deleting an error-handling code to pass a test) to filter out similarly bad repairs (Tan et al, 2016). We also mention that there is a repair system fixing specifically regression errors, using repair templates tailored for fixing regression errors (Tan and Roychoudhury, 2015).

Meanwhile, more recent constraint-based repair systems include DirectFix, Angelix, and Nopol. DirectFix (Mechtaev et al, 2015) generates minimal repairs (repairs that change the original program minimally) by reducing the program repair problem into a MaxSAT problem. It is shown that minimal repairs generated by DirectFix cause regression less frequently than SemFix-generated repairs. Angelix (Mechtaev et al, 2016) transfers DirectFix techniques to large-scale real-world software such as a PHP interpreter and Heartbleed-containing OpenSSL. While SemFix, DirectFix and Angelix repair buggy C programs, Nopol (Xuan et al, 2017) is a constraint-based repair system for Java programs. Despite the advancement of these recent repair tools such as SPR and Angelix, they still inherit the same problem of their predecessors such as GenProg and SemFix; that is, they also run a risk of generating repairs causing regressions, due to the incompleteness of a test-suite. Investigating whether our results are extended to these more recent tools is left as future work.

While most automated program repair approaches are test-driven (a test-suite is used as a specification), AutoFix (Pei et al, 2014) uses program contracts such as pre/post-conditions as a specification. Similarly, there are other specification-driven program repair techniques (He and Gupta, 2004; Jobstmann et al, 2005; Gopinath et al, 2011; Könighofer and Bloem, 2011; Samimi et al, 2010; Elkarablieh and Khurshid, 2008). Apart from the preceding source-level repair techniques, there are also runtime error repairing techniques that recover from corrupted program states (e.g., null-derefernce) while the program is being executed (Perkins et al, 2009; Long et al, 2014).

8 Conclusion

Many automated program repair tools use a test-suite as the specification of the software under repair. Thus, automated program repair tools may end up generating a repair that fails new tests that were not available at the time of repair, causing regressions. Indeed, our experimental results show that regression in generated repairs is widespread. Our study is the largest to date to show how severe the regression problem of automatically generated repairs is. To address this problem, we in this paper investigate the possibility of using test-suite metrics proposed for software testing to control the regression ratio of automatically generated repairs. Overall, the results of our study are positive. Traditional test-suite metrics are generally negatively correlated with the regression ratio of repairs, implying that traditional test-suite metrics can also be used for automated program repair. In particular, statement coverage is shown to be most strongly correlated among the metrics we investigate, implying that to reduce the regression ratio, increasing statement coverage is generally more promising than improving branch coverage or mutation score.

Acknowledgements This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. The first author thanks Innopolis University for its support.

References

Andrews JH, Briand LC, Labiche Y, Namin AS (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering 32(8):608–624

Artzi S, Dolby J, Tip F, Pistoia M (2010) Directed test generation for effective fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pp 49–60

Assiri FY, Bieman JM (2014) An assessment of the quality of automated program operator repair. In: Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, ICSE '14, pp 273–282

Baudry B, Fleurey F, Le Traon Y (2006) Improving test suites for efficient fault localization. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp 82–91

Böhme M, Roychoudhury A (2014) CoREBench: Studying complexity of regression errors. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA '14, pp 105–115

Böhme M, Oliveira BCdS, Roychoudhury A (2013a) Partition-based regression verification. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp 302–311

Böhme M, Oliveira BCdS, Roychoudhury A (2013b) Regression tests to expose change interaction errors. In: Proceedings of the 2013 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '13, pp 334–344

Cadar C, Engler D (2005) Execution generated test cases: How to make systems code crash itself. In: Proceedings of the 12th International Conference on Model Checking Software, SPIN '05, pp 2–23

Cadar C, Dunbar D, Engler D (2008) KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI' 08, pp 209–224

Dallmeier V, Zeller A, Meyer B (2009) Generating fixes from object behavior anomalies. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pp 550–554

Debroy V, Wong WE (2010) Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation, ICST '10, pp 65–74

Debroy V, Wong WE (2014) Combining mutation and fault localization for automated program debugging. Journal of Systems and Software 90:45–60

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4):405–435

Elkarablieh B, Khurshid S (2008) Juzi: A tool for repairing complex data structures. In: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pp 855–858

Godefroid P, Klarlund N, Sen K (2005) DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pp 213–223

Gopinath D, Malik MZ, Khurshid S (2011) Specification-based program repair using SAT. In: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software, TACAS '11/ETAPS '11, pp 173–188

He H, Gupta N (2004) Automated debugging using path-based weakest preconditions. In: Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering, FASE '04, pp 267–280

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

Jobstmann B, Griesmayer A, Bloem R (2005) Program repair as a game. In: Proceedings of the 17th International Conference on Computer Aided Verification, CAV '05, pp 226–238

Jones JA, Harrold MJ, Stasko JT (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp 467–477

Ke Y, Stolee KT, Le Goues C, Brun Y (2015) Repairing programs with semantic code search (t). In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE '15, pp 295–306

Kendall MG (1945) The treatment of ties in ranking problems. Biometrika 33(3):239–251

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp 802–811

Kong X, Zhang L, Wong WE, Li B (2015) Experience report: How do techniques, programs, and tests impact automated program repair? In: Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE '15, pp 194–204

Könighofer R, Bloem R (2011) Automated error localization and correction for imperative programs. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, pp 91–100

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012a) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp 3–13

Le Goues C, Nguyen T, Forrest S, Weimer W (2012b) GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38(1):54–72

Le Goues C, Forrest S, Weimer W (2013) Current challenges in automatic software repair. Software Quality Journal 21(3):421–443

Liblit B, Aiken A, Zheng AX, Jordan MI (2003) Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, PLDI '03, pp 141–154

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '15, pp 166–178

Long F, Rinard M (2016a) An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp 702–713

Long F, Rinard M (2016b) Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, pp 298–312

Long F, Sidiroglou-Douskos S, Rinard M (2014) Automatic runtime error repair and containment via recovery shepherding. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp 227–238

Maldonado JC, Delamaro ME, Fabbri SCPF, Simão AdS, Sugeta T, Vincenzi AMR, Masiero PC (2001) Proteum: A family of tools to support specification and program testing based on mutation. In: Wong WE (ed) Mutation Testing for the New Century, Kluwer Academic Publishers, Norwell, MA, USA, pp 113–116

Mechtaev S, Yi J, Roychoudhury A (2015) DirectFix: Looking for simple program repairs. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, ICSE '15, pp 448–458

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp 691–701

Miller W, Spooner DL (1976) Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3):223–226

Namin AS, Andrews JH (2009) The influence of size and coverage on test suite effectiveness. In: Proceedings of the 8th International Symposium on Software Testing and Analysis, ISSTA '09, pp 57–68

Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) SemFix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp 772–781

Pearson K (1895) Note on regression and inheritance in the case of two parents. Proceedings of the Royal Society of London 58:240–242

Pei Y, Furia C, Nordio M, Wei Y, Meyer B, Zeller A (2014) Automated fixing of programs with contracts. IEEE Transactions on Software Engineering 40(5):427–449

Perkins JH, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G, Wong WF, Zibin Y, Ernst MD, Rinard M (2009) Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, pp 87–102

Person S, Yang G, Rungta N, Khurshid S (2011) Directed incremental symbolic execution. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pp 504–515

Qi Y, Mao X, Lei Y (2013) Efficient automated program repair through fault-recorded testing prioritization. In: Proceedings of the 2013 IEEE International Conference on Software

Maintenance, ICSM ’13, pp 180–189

Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, ICSE ’14, pp 254–265

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pp 24–36

Samimi H, Aung ED, Millstein T (2010) Falling back on executable specifications. In: Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP’10, pp 552–576

Samimi H, Schäfer M, Artzi S, Millstein T, Tip F, Hendren L (2012) Automated repair of HTML generation errors in PHP applications using string constraint solving. In: Proceedings of the 34th International Conference on Software Engineering, ICSE ’12, pp 277–287

Santelices R, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ (2008) Test-suite augmentation for evolving software. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE ’08, pp 218–227

Shoenauer M, Xanthakis S (1993) Constrained GA optimization. In: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA ’93, pp 573–580

Smith EK, Barr ET, Le Goues C, Brun Y (2015) Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 2015 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE ’15, pp 532–543

Tan SH, Roychoudhury A (2015) relifix: Automated repair of software regressions. In: Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE ’15, pp 471–482

Tan SH, Yoshida H, Prasad MR, Roychoudhury A (2016) Anti-patterns in search-based program repair. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE’16, pp 727–738

Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: Models and first results. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE ’13, pp 356–366

White DR, Arcuri A, Clark JA (2011) Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation 15(4):515–538

Xuan J, Martinez M, Demarco F, Clement M, Marcote SRL, Durieux T, Berre DL, Monperrus M (2017) Nopol: Automatic repair of conditional statement bugs in Java programs. IEEE Transactions on Software Engineering 43(1):34–55

Yao X, Harman M, Jia Y (2014) A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering, ICSE ’14, pp 919–930