Balancing Trade-Offs in Test-Suite Reduction

August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign
{awshi2, gyori, gliga, zaytsev2, marinov}@illinois.edu

ABSTRACT

Regression testing is an important activity but can get expensive for large test suites. Test-suite reduction speeds up regression testing by identifying and removing redundant tests based on a given set of requirements. Traditional research on test-suite reduction is rather diverse but most commonly shares three properties: (1) requirements are defined by a coverage criterion such as statement coverage; (2) the reduced test suite has to satisfy all the requirements as the original test suite; and (3) the quality of the reduced test suites is measured on the software version on which the reduction is performed. These properties make it hard for test engineers to decide how to use reduced test suites.

We address all three properties of traditional test-suite reduction: (1) we evaluate test-suite reduction with requirements defined by killed mutants; (2) we evaluate inadequate reduction that does not require reduced test suites to satisfy all the requirements; and (3) we propose evolution-aware metrics that evaluate the quality of the reduced test suites across multiple software versions. Our evaluations allow a more thorough exploration of trade-offs in test-suite reduction, and our evolution-aware metrics show how the quality of reduced test suites can change after the version where the reduction is performed. We compare the trade-offs among various reductions on 18 projects with a total of 261,235 tests over 3,590 commits and a cumulative history spanning 35 years of development. Our results help test engineers make a more informed decision about balancing size, coverage, and fault-detection loss of reduced test suites.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Experimentation, Measurement

Keywords: Test-suite reduction, software evolution

1. INTRODUCTION

Developers often build regression test suites that are automatically run as the software evolves. Modern software evolves fairly quickly, with changes pushed to repositories even several times per minute [7, 10, 12]. Meanwhile, regression test suites also grow, and even older reports mention real-world regression test suites that could take weeks to finish [33, 34]. As a result, regression testing is becoming more important but also more expensive.

Test-suite reduction [22–24,29,31,38,41,42] is an approach to make regression testing faster. The goal is to identify tests that can be removed from a test suite without substantially reducing its fault-detection capability. Most test-suite reduction techniques identify such tests based on satisfying redundant test *requirements*. Each test satisfies certain requirements, e.g., covers certain statements or branches for the statement or branch coverage criteria. A test t from a test suite $\mathcal{T}$ is *redundant* [38] if t satisfies only the requirements satisfied by the other $\mathcal{T} \setminus \{t\}$ tests from the test suite.

Researchers have proposed many algorithms to identify redundant tests [11, 13, 14, 22–24, 26, 29, 35, 36]. (Yoo and Harman present an extensive survey of regression testing, including test-suite reduction [38].) Removing redundant tests from an original test suite produces a *reduced test suite*. Prior research measured the quality of reduced test suites in comparison to the original test suites mostly by two metrics. The *reduction in test-suite size* is most commonly measured as the ratio of the number of tests removed from the original test suite to the number of tests in the original test suite. The *loss in fault-detection capability* is most commonly measured as the ratio of the number of faults missed by the reduced test suite to the number of faults detected by the original test suite; the faults are either automatically generated mutants or manually seeded faults [11,22,24,26,29,31,32,35,36,38,41]. Some prior work also used other metrics, e.g., test-execution cost or coverage of fault history, as discussed in Section 7.

Despite a variety of techniques, most traditional test-suite reduction research shares three properties. First, redundant tests are determined with respect to some *structural or data-flow coverage criteria*, e.g., statement coverage [11,13,22,41], block coverage [35, 36], branch coverage [24, 31], or def-use coverage [23, 24]. Second, test-suite reduction produces a reduced test suite that covers exactly the *same* requirements as the original test suite; we call this *adequate test-suite reduction*. Third, evaluation of the effectiveness of test-suite reduction was conducted only on the *same software version on which reduction was performed*.

These three properties of traditional test-suite reduction research limit the insight into the trade-offs of reduced test suites. Consider a test engineer who needs to decide whether

FSE'14 , November 16–22, 2014, Hong Kong, China

to use the original test suite or a reduced test suite. We can distinguish two scenarios. One scenario is to permanently remove redundant tests from the original test suite and keep only the reduced test suite. Such reduced test suites run faster, but the removed tests could detect faults that the remaining tests in the reduced test suite would not detect as software evolves. The other scenario is to keep both the reduced test suite and the original test suite, but run the reduced test suite more often, e.g., run it for all software changes during day and then run the original test suite only during night. In both scenarios, the test engineer needs to balance the reduction in the test-suite size with the risk that the fault-detection capability of the reduced test suite does not degrade substantially compared to the original test suite *as the project evolves*[1].

In this paper we address the three properties of traditional test-suite reduction. First, we evaluate *test-suite reduction based on killed mutants* rather than traditional coverage, i.e., the reduced test suite has to kill the same mutants killed by the original test suite; this reduction was proposed by Offutt et al. [30] almost 20 years ago but not widely evaluated. We also propose an even stronger reduction that requires the reduced test suite to preserve both statement coverage and killed mutants of the original test suite. Second, we evaluate *inadequate test-suite reduction* where the reduced test suite need not cover all the requirements as the original test suite [37]. Such inadequate reduction leads to higher reductions in test-suite size at the expense of higher losses in fault-detection capability. Third, we introduce *novel reduction metrics based on software evolution*. Our metrics evaluate the code coverage and fault-detection capability of reduced test suites on *multiple software versions*.

We make the following contributions:

Test-Suite Reduction Based on Killed Mutants We evaluate using killed mutants as requirements for test-suite reduction (potentially combined with the more traditional test-suite reduction based on statement coverage) to achieve lower loss in fault-detection capability (at the expense of lower reduction in test-suite size).

Inadequate Reduction We evaluate inadequate reduction that relaxes the constraint of covering all requirements when performing test-suite reduction to achieve higher reduction in test-suite size (at the expense of higher loss in fault-detection capability).

Evolution-aware Metrics We propose novel evolution-aware metrics to evaluate the impact of software evolution on reduced test suites.

Extensive Evaluation We explore the trade-offs of test-suite reduction based on killed mutants and inadequate reduction with both traditional and our proposed evolution-aware metrics on 18 projects with a total of 261,235 tests, over 3,590 commits, and spanning 35 years of cumulative evolution history. To the best of our knowledge, this is the largest dataset used for evaluating test-suite reduction.

The results show that traditional reduction based on statement coverage can reduce test-suite size on average 62.9% but loses up to 20.5% in killed mutants. In contrast, the reduction based on killed mutants achieves no loss in killed mutants with 11.9pp[2] increase in size over test suites reduced using statement coverage. The reduction based on both statement coverage and killed mutants has no losses in those requirements but has 2.7pp increase in test-suite size compared to the reduction based on killed mutants. Hence, a test engineer who plans to permanently remove redundant tests may prefer to use one of the reductions based on killed mutants instead of the traditional reduction based on covered statements.

Further, by allowing the reduced test suite to satisfy 95% instead of 100% of the original test suites' requirements, the median reduction in test-suite size increases by 17.14pp. We show how relaxing the coverage further results in even higher reductions in test-suite size. Hence, a test engineer who plans to use both the reduced test suite and the original test suite may prefer to use some inadequate reduction instead of the traditional adequate reduction.

Finally, we evaluate for all kinds of reductions how the killed mutants of reduced test suites change as software evolves. We find the numbers to remain fairly stable, dropping on median by at most 0.77pp even after many software versions. Hence, a test engineer deciding whether/how to use the reduced test suite can be relatively confident that making the decision based on the current software version is likely to reflect what happens in the future software versions.

2. BACKGROUND

We first describe the traditional test-suite reduction and commonly used reduction algorithms. We then revisit the three properties frequently shared in traditional test-suite reduction research: (1) requirements are defined by code coverage criteria, (2) reduction algorithms perform adequate test-suite reduction, and (3) evaluations of reduction algorithms use only one software version.

DEFINITION 1. *Traditionally [38], a test-suite reduction algorithm* Algo *takes two inputs: (1) a function ρ that returns the set of satisfied requirements for a given test suite and (2) the original test suite $\mathcal{O}$ to be reduced. It returns a reduced test suite $\mathcal{R} \subseteq \mathcal{O}$ that satisfies the same requirements as the original test suite:*

$$\mathsf{Algo}(\rho, \mathcal{O}) = \mathcal{R}, \text{ such that } \rho(\mathcal{O}) = \rho(\mathcal{R})$$

2.1 Requirements

Code coverage is widely used for measuring the quality of test suites. A coverage criterion defines a set of requirements and measures which requirements a given test suite satisfies. For example, statement coverage measures which statements are covered during a test-suite run.

Previous research on test-suite reduction has often used statement coverage to detect redundant tests [11, 13, 14, 22, 41], i.e., the first argument of Algo was the stmt function that returns the set of covered statements for a given test suite. Other criteria were also used for test-suite reduction, e.g., block coverage [35, 36], branch coverage [24, 31], and def-use coverage [23, 24]. In the remaining text, we use the term *technique* to refer to an algorithm instantiated with a function that returns the satisfied requirements.

[1]Note that test-suite reduction does *not* explicitly consider software changes from software evolution, unlike regression test selection that chooses tests based on software changes.

[2]Here "pp" stands for "percentage points" which are used to describe differences between values expressed in percentages to avoid the following example ambiguity: if a value is 51% and increases 11.9%, does it become 57.06% or 62.9%? The increase of 11.9pp means that it becomes 62.9%.

2.2 Reduction Algorithms

The most popular traditional reduction algorithms create reduced test suites that satisfy the same requirements as the original test suite, i.e., $\rho(\mathcal{O}) = \rho(\mathcal{R})$. We thus call them *adequate* test-suite reduction algorithms. Specifically, when a traditional test-suite reduction algorithm is instantiated with statement coverage, we call it *Statement Adequate Reduction* (SAR) technique.

The most widely used algorithm is Greedy [14], which iteratively selects the test that satisfies the most requirements not previously satisfied. More precisely, starting with an empty set of tests $\mathcal{R}$, and an empty set of satisfied requirements $\rho(\mathcal{R})$, the Greedy algorithm selects at each step a test $t \in \mathcal{O}$ that satisfies the highest number of requirements not in $\rho(\mathcal{R})$, and adds the selected test to $\mathcal{R}$. This process continues until $\rho(\mathcal{O}) = \rho(\mathcal{R})$. When two or more tests satisfy the same number of requirements, various strategies can be used to break ties to determine the next test to add to $\mathcal{R}$. We currently break ties randomly as commonly done in prior work [13, 14, 23, 31, 32, 41, 42].

The other commonly used algorithms for test-suite reduction are GE, GRE, HGS, and ILP; their details are available elsewhere, e.g., in the regression-testing survey [38].

2.3 Evaluating Reduction Algorithms

Studies [11, 22, 24, 26, 29, 31, 32, 35, 36, 41] that evaluated test-suite reduction algorithms used mainly two metrics—*reduction in test-suite size* and *loss in fault-detection capability*—to measure the quality of the reduced test suites on the *one software version* on which the reduction is performed. The size reduction was measured as the ratio of tests removed from the original test suite over the number of tests in the original test suite:

$$\mathsf{SizRed} = (|\mathcal{O}| - |\mathcal{R}|)/|\mathcal{O}| \times 100$$

The loss in fault-detection capability was measured as the ratio of the number of faults missed by the reduced test suite over the number of faults detected by the original test suite:

$$\mathsf{MutLoss} = (|\mathsf{mut}(\mathcal{O})| - |\mathsf{mut}(\mathcal{R})|)/|\mathsf{mut}(\mathcal{O})| \times 100$$

where mut is the function that returns the set of detected faults for the given test suite. Due to the challenges in collecting a large number of known real faults per project, researchers most commonly evaluated reduction algorithms using mutation testing or manually seeded faults.

Mutation testing [8, 15, 21, 22, 25, 39–41] systematically inserts syntactic changes, called *mutants*, in code and measures how many of these mutants are killed by a given test suite; a mutant is considered *killed* if at least one of the tests fails (and the same test passes in the non-mutated run). The quality of a test suite is measured as the ratio of the number of killed mutants over the total number of systematically inserted mutants; this ratio is called the *mutation score*. Previous research on test-suite reduction measured the effectiveness of reduction algorithms by comparing the mutation score of the reduced test suite to the mutation score of the original test suite [22, 41]. We also use mutation score to evaluate a test suite's fault-detection capability.

Interestingly, previous studies reported conflicting findings in terms of the loss of fault-detection capability: reduced test suites sometimes had low loss [35, 36, 41] and sometimes had high loss [22, 24, 29, 31, 32].

Additionally, Yoo and Harman [37] used multi-objective optimizations to consider multiple metrics at once (including execution cost, approximated by the number of instructions to obtain a machine-independent metric). While they applied these optimizations to perform test selection and prioritization, one could apply them also to test-suite reduction. However, in this study, we use only the size of the test suite as a proxy for execution cost.

Limited Evaluation: To summarize, we are concerned with three properties of test-suite reduction research. (1) Requirements based on code coverage may result in relatively small reduced test suites that do not preserve well the fault-detection capability of the original test suites. (2) Adequate test-suite reduction may result in relatively large reduced test suites, and if more size reduction is acceptable, the loss in the fault-detection capability may not be high. (3) *All* previous studies evaluated each reduced test suite only on a single software version on which the suite is reduced, which may not give enough insight into the quality of reduced test suites through evolution.

3. NEW EVALUATION METHODOLOGY

We next describe our evaluation that extends the traditional research on test-suite reduction. Specifically, we evaluate (1) requirements for test-suite reduction based on killed mutants, (2) inadequate reduction that achieves higher reductions in test-suite size by relaxing the requirements that a test-suite reduction algorithm has to satisfy, and (3) novel metrics to evaluate test-suite reduction algorithms by measuring the impact of software evolution on the reduced test suites produced by each algorithm.

3.1 Requirements

Considering the importance of preserving the high mutation score after test-suite reduction, we evaluate requirements defined by killed mutants and requirements defined by a combination of statement coverage and killed mutants.

3.1.1 Mutant Adequate Reduction

Mutant Adequate Reduction (MAR) technique reduces a test suite based on killed mutants [30]. Instead of finding a reduced test suite that covers the same statements as the original test suite (as done by SAR), MAR finds a reduced test suite that kills the same mutants (and thus achieves the same mutation score) as the original test suite. In other words, the first argument in the reduction algorithm, i.e., ρ in Algo (Definition 1), is the mut function that returns the set of mutants killed by a given test suite. All test-suite reduction algorithms we evaluate can then use killed mutants instead of covered statements as requirements.

3.1.2 Statement-Mutant Adequate Reduction

Statement-Mutant Adequate Reduction (SMAR) technique reduces a test suite based on *both* statement coverage and killed mutants. It creates one joint set of requirements of *all* statements covered and *all* mutants killed by the original test suite. Although SMAR by definition preserves both statement coverage and killed mutants, it is important to explore the reduction in test-suite size that SMAR provides.

3.2 Reduction Algorithms

Most traditional test-suite reduction algorithms aim to find a reduced test suite that satisfies *all* requirements sat-

	0	1	2	
0	$\rho^0(\mathcal{O}_0)$, $\rho^0(\mathcal{R}_0)$	$\rho^1(\mathcal{O}_0)$, $\rho^1(\mathcal{R}_0)$	$\rho^2(\mathcal{O}_0)$, $\rho^2(\mathcal{R}_0)$	$\cdots$
1	—	$\rho^1(\mathcal{O}_1)$, $\rho^1(\mathcal{R}_1)$	$\rho^2(\mathcal{O}_1)$, $\rho^2(\mathcal{R}_1)$	$\cdots$
2	—	—	$\rho^2(\mathcal{O}_2)$, $\rho^2(\mathcal{R}_2)$	$\cdots$
$\vdots$	—	—	—	$\cdots$

Figure 1: Visualized collected data used to calculate novel metrics, where $\rho = \{\mathsf{stmt}, \mathsf{mut}\}$

isfied by the original test suite, be the requirements all statements covered (SAR) or all killed mutants (MAR) or both (SMAR). However, if the goal is to greatly reduce the test-suite size, it may be acceptable to have some loss in both statement coverage and killed mutants [37].

We evaluate *inadequate reduction*: instead of producing a reduced test suite that satisfies 100% of the requirements of the original test suite, the reduced test suite can satisfy less than 100% of the requirements (e.g., 95% of the statements covered by the original test suite).

Definition 2. *An inadequate reduction algorithm takes three inputs: (1) a function ρ that returns the set of satisfied requirements for a given test suite, (2) the original test suite $\mathcal{O}$, and (3) the percentage $0 \leq l \leq 100$ of requirements that must be satisfied. It returns a reduced test suite $\mathcal{R} \subseteq \mathcal{O}$ that satisfies (at least) l% of the requirements satisfied by $\mathcal{O}$:*

$$\mathsf{InAlgo}(\rho, \mathcal{O}, l) = \mathcal{R}, \text{ such that } |\rho(\mathcal{R})| \geq l/100 \times |\rho(\mathcal{O})|$$

Note that InAlgo is a more general form of Algo, i.e., InAlgo with $l = 100$ is equivalent to Algo.

While InAlgo can be applied to any set of requirements, we particularly identify *Statement Inadequate Reduction* (SIR), which achieves less than 100% statement coverage of the original test suite (InAlgo is evaluated with the stmt function and $l \leq 100$), and *Mutant Inadequate Reduction* (MIR), which achieves less than 100% killed mutants of the original test suite (InAlgo is evaluated with the mut function and $l \leq 100$). Both SIR and MIR can be instantiated to any l between 0 and 100, which we call *inadequacy level*. We adjust the Greedy algorithm to select tests for the given inadequacy level. Note that the actual percentage of satisfied requirements may be higher than the given l if no set of tests can achieve the exact value.

3.3 Evaluating Reduction Algorithms

Previous research measured the size reduction (SizRed) and the loss in fault-detection capability (MutLoss) of each reduced test suite on only *one single version*. We propose an approach for evaluating reduction techniques by measuring how the quality of each reduced test suite varies over *multiple project versions*. Furthermore, we propose several other metrics, besides killed mutants, to better characterize the quality of the test suite over multiple versions. We believe that our metrics, by accounting for software evolution, give a better insight into the effectiveness of test-suite reduction techniques than the previously used metrics give.

As our evaluation introduces a temporal dimension (along software versions), we extend our previous notation to include this new dimension. We denote the original test suite at version i as $\mathcal{O}_i$ and the reduced test suite (obtained by a reduction technique on $\mathcal{O}_i$) at version i as $\mathcal{R}_i$. We use $\mathcal{T}_i$ to refer to any test suite at version i. The function $\rho^j(\mathcal{T}_i)$ returns the set of requirements satisfied at version j by the test suite $\mathcal{T}_i \cap \mathcal{O}_j$. As earlier, $\rho \in \{\mathsf{stmt}, \mathsf{mut}\}$, i.e., it either returns the set of covered statements or the set of killed mutants. Note that $\rho^j(\mathcal{T}_i)$ accounts for the tests that were removed between versions i and j ($\mathcal{T}_i \setminus \mathcal{O}_j$). In our experiments, the tests are considered removed if they are either deleted altogether or renamed/moved. $\rho^j(\mathcal{T}_i)$ ignores newly added tests (i.e., $\mathcal{O}_j \setminus \mathcal{T}_i$) because our goal is to evaluate how the *quality of the reduced test suite compares relative to the original test suite* as software evolves. In other words, if a test engineer needs to decide whether to use the original test suite or the reduced suite, what matters is the comparison among those two and not the fact that both of them can be further extended with more tests as software evolves. Finally, we introduce a function $\mathsf{dist}(i, j)$ that calculates the distance between versions i and j, i.e., $j - i$ if i and j are integer version numbers and $i \leq j$.

Figure 1 visualizes the computation of ρ values at each version. Columns and rows correspond to the sequence of project versions used to evaluate a reduction algorithm. For each row/version i, we evaluate $\rho^j(\mathcal{O}_i)$ and $\rho^j(\mathcal{R}_i)$ for each column/version j. Namely, we measure the number of statements covered and the number of mutants killed for the original test suite at version i (and reduced test suite at version i) on version j, with version j later than version i in history.

3.3.1 Impact of Evolution on Statement Coverage

We measure how the evolution impacts statement coverage of the reduced test suite with respect to the coverage of the original test suite. To measure this impact, we define *statement coverage loss* between two project versions.

Definition 3. *Statement coverage loss between versions i and j is:*

$$\mathsf{StmtLoss}_i^j = (|\mathsf{stmt}^j(\mathcal{O}_i)| - |\mathsf{stmt}^j(\mathcal{R}_i|))/|\mathsf{stmt}^j(\mathcal{O}_i)| \times 100$$

Namely, the statement coverage loss measures the ratio of statement coverage missed by the reduced test suite over the statement coverage achieved by the original test suite, at the same version or subsequent versions.

In the traditional evaluation approach, the statement coverage loss would always be 0, because the original test suite and the reduced test suite achieve the same statement coverage by construction on the current version i, and no other versions would be considered. In other words, the traditional approach only considers the diagonal elements from Figure 1. In contrast, our approach measures the statement coverage loss for all versions i and j such that $\mathsf{dist}(i, j) \geq 0$.

We measure the statement coverage loss for the following reasons. First, the statement coverage of the reduced test suite can vary as software evolves. It is important to quantify the variation and evaluate if the reduced test suite remains equally good (compared to the original test suite) as on version i on which the reduction was performed. Second, if test-suite reduction is performed using algorithms parameterized with our requirements, it is important to evaluate the impact of such reduction on statement coverage and contrast the loss in statement coverage with the loss in killed mutants, which we discuss next.

3.3.2 Impact of Evolution on Killed Mutants

We measure how the killed mutants vary between versions for the original and reduced test suites. Previous research

only evaluated the effectiveness of the reduced test suite on the version on which the reduction was performed (the elements on the diagonal in Figure 1). The effectiveness was often measured using MutLoss.

We introduce the *mutants killed loss*, which measures the impact of evolution on the effectiveness of the original and reduced test suites.

DEFINITION 4. *Mutants killed loss between two versions i and j is:*

$$\mathsf{MutLoss}_i^j = (|\mathsf{mut}^j(\mathcal{O}_i)| - |\mathsf{mut}^j(\mathcal{R}_i)|)/|\mathsf{mut}^j(\mathcal{O}_i)| \times 100$$

Namely, the mutants killed loss measures the ratio of mutants not killed by the reduced test suite over mutants killed by the original test suite, at the same version or subsequent versions. Note that by setting $i = j$, we get the formula commonly used in the traditional approaches for evaluating test-suite reduction techniques.

3.3.3 Relative Evolution Change

To further compare the changes in both statement coverage and killed mutants across different software versions between different projects, we introduce *Relative Evolution Change* (REC).

DEFINITION 5. *Given a test suite reduced at version i, the Relative Evolution Change at version j is:*

$$\mathsf{REC}_i^j = Loss_i^j - Loss_i^i$$

where $Loss \in \{\mathsf{StmtLoss}, \mathsf{MutLoss}\}$, i.e., REC can be calculated for either statement coverage loss or mutants killed loss. REC is defined to be the loss in evolution metrics some number of versions after reduction. The higher the REC, the more loss, and therefore the worse the reduced test suite performs (compared to the original test suite) as the software evolves across versions. The metric reported for every project is a relation between the reduced test suite and the original test suite, so the REC can be compared across projects. Therefore, we can define $\mathsf{REC}_d = \{\mathsf{REC}_i^j \mid$ for all i and j such that $d = \mathsf{dist}(i, j)\}$, i.e., REC is parameterized only by the distance from the initial reduction. This allows us to compare REC_d for various distances across various projects. Our goal is to evaluate the impact of distance on the reduced test suites.

Note that REC_d can have a positive value or a negative value. A positive value would indicate that changes had negative impact on the reduced test suite (more loss compared to original test suite in a later version). On the other hand, a negative value would indicate a positive impact on the reduced test suite, i.e., the changes cause the reduced test suite to be closer in quality to the original test suite. We define REC *stability* as the difference between min and max REC values across all versions considered.

4. EVALUATION

This section describes our evaluation of test-suite reduction on a set of 18 medium-sized, open-source projects from GitHub. We first describe the projects used in our study. We then describe the results obtained on a single software version, as often done in the traditional test-suite reduction. We next describe the results obtained by analyzing reduced test suites as software evolves as proposed in this paper; overall, our experiments consider the evolution of the projects over a total number of 3,590 commits. We finally describe the results for inadequate reduction.

We ran all our experiments on a 2.66GHz Intel Xeon X5650 machine with 16GB of RAM, running Scientific Linux 6.5 and Java OpenJDK 64-Bit Server version 1.7.0_51. The total machine time to run all our experiments is approximately eight days.

4.1 Implementation

We use the PIT mutation tool [5] to collect the statement coverage and killed mutants for each test suite. PIT uses 15 mutation operators, including these: replace numerical constant, negate conditional, replace arithmetic operator, and remove method calls. We chose PIT because it is somewhat robust and has been recently deployed in industry, but we still had to make two modifications to PIT to collect all the data required for our experiments. First, we extended PIT to collect the full coverage and kill matrices that record for each test in the test suite the statements it covers and the mutants it kills. Out-of-the-box, PIT stops running tests on a mutant as soon as one test kills the mutant. However, to evaluate test-suite reduction, we wanted to collect for each mutant the set of all the tests that kill the mutant. That way we can determine which mutants would be killed by various reduced test suites directly from the matrices, without rerunning PIT on those reduced test suites. We define a mutant to be killed if a test fails or errors when running on the mutated code; PIT ignores the cases where the test times out on the mutated code. Second, we modified PIT to ignore tests that fail on the original (not mutated) code due to PIT instrumentation. By default, PIT stops execution when a test fails on the original code. However, to collect data for more projects, we changed PIT to ignore such tests and to continue execution.

Following the traditional literature sources [11, 13, 14, 23, 41, 42], we implemented the Greedy, GE, GRE, HGS, and ILP reduction algorithms that can operate on the collected matrices. To implement the ILP algorithm, we used IBM's CPLEX Optimizer solver version 12.3 [2].

For baseline comparison, we also implemented random reduction that, given a test suite and a desired size, selects a random subset of the given size from the given test suite. We create random test suites of the same size as the reduced test suites created by the other reduction algorithms, giving a random test suite corresponding to each reduced test suite.

4.2 Projects

As subjects of our evaluation, we select Java projects from GitHub that are built through Maven [4]. We focus on Maven projects to simplify the automation of running mutation testing using PIT, as PIT integrates well with Maven. Initially, we downloaded the 2,000 most popular Java projects on GitHub and selected the projects that satisfy these four conditions: (1) the project uses Maven (876 out of 2,000); (2) the software history of the project has more than 100 commits (655 out of 876); (3) PIT can successfully run tests on the latest version of the project[3] (44 out of 655), i.e., without crashing or reporting an internal error; and (4) after running PIT over several commits from the project and eliminating runs for which PIT failed, the project should have at least four data points (18 out of 44).

[3]In Git, the version we used is HEAD of the master branch.

1. Project	2. Ver	3. LOC Min	4. LOC Med	5. LOC Max	6. Tests Min	7. Tests Med	8. Tests Max	9. Mutants Min	10. Mutants Med	11. Mutants Max	12. Mut Score (%) Min	13. Mut Score (%) Med	14. Mut Score (%) Max
Commons-Lang	10	61,353	62,224	64,038	2,291	2,365	2,480	25,415	25,620	26,538	80.0	80.1	80.3
AssertJ-Core	8	55,443	63,829	67,282	4,055	4,864	5,269	7,691	8,988	10,004	82.1	85.3	87.4
Square Wire	4	6,410	10,327	12,019	19	49	61	3,415	4,736	5,625	77.1	78.8	79.1
Jasmine	6	1,640	3,912	4,348	7	82	118	377	1,005	1,210	43.5	59.9	61.4
Dropwizard	6	11,477	11,849	12,536	270	291	318	5,041	5,489	5,592	64.5	66.4	73.1
GS-Collections	10	147,899	168,088	205,637	12,627	13,519	16,612	44,417	49,347	57,600	83.9	84.1	84.6
Scribe-Java	8	2,497	3,803	5,957	38	59	99	579	974	1,622	34.7	49.5	59.4
Cloudera ML	4	5,142	6,417	8,366	14	16	39	2,723	3,283	4,641	35.6	37.7	41.4
Caelum V|raptor	5	31,176	31,964	32,997	985	1,081	1,124	9,338	9,667	9,733	72.5	75.1	75.3
Cloudfoundry	6	14,957	19,317	25,648	103	141	477	4,155	5,737	9,852	45.8	51.1	52.9
LA4J	9	8,094	10,613	12,555	172	437	625	7,062	8,408	9,086	42.7	44.7	58.1
JodaTime	7	81,324	82,052	83,610	3,834	3,924	4,033	27,451	27,750	28,382	72.2	72.4	74.3
MessagePack	10	23,724	34,381	41,542	428	1,117	1,148	6,243	6,856	11,452	37.2	53.7	59.4
SLF4J	10	10,356	11,292	12,023	76	85	106	2,486	2,618	2,992	27.2	28.5	28.9
JOPT-Simple	5	6,636	7,863	8,569	394	621	657	1,561	1,608	1,796	92.7	94.2	94.6
Java-APNS	8	1,362	2,822	3,839	15	62	87	337	734	1,122	30.3	40.9	57.4
Caelum Stella	9	19,100	27,229	36,002	387	524	660	4,341	4,653	5,515	66.7	68.4	74.5
SQL-Parser	6	13,632	15,532	17,577	48	109	243	9,615	11,437	12,778	23.3	28.6	33.7
Mean over ver	-		36,838			1,994			11,366			60.7%	

Figure 2: Statistics of projects used in our experiments

		Rand_{SAR}	Rand_{MAR}	Rand_{SMAR}	SAR	MAR	SMAR	SIR_{MAR}	MIR_{SAR}
SizRed	Min	29.2	10.5	10.5	29.2	10.5	10.5	31.6	36.8
	Med	62.9	51.0	48.3	62.9	51.0	48.3	74.2	73.2
	Max	92.9	91.5	91.0	92.9	91.5	91.0	96.7	95.8
StmtLoss	Min	1.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Med	24.2	19.9	18.3	0.0	2.1	0.0	2.0	6.4
	Max	55.2	49.5	34.5	0.0	14.9	0.0	14.1	26.5
MutLoss	Min	2.4	1.0	1.3	0.2	0.0	0.0	0.2	0.2
	Med	27.5	23.2	21.5	3.5	0.0	0.0	10.8	3.5
	Max	55.7	45.6	45.7	20.5	0.0	0.0	27.2	20.5

Figure 3: SizRed, StmtLoss, and MutLoss for different requirements using Greedy algorithm

In the end, we obtained 18 projects, and we refer to their commits for which we collected data as *versions*.

Overall, we collected data for a total of 131 versions from all 18 projects. We select each version by counting backwards 30 commits at a time from the HEAD commit of each project[4] up to 10 versions per project. (A project may have fewer than 10 versions if either it did not use Maven in the past or PIT starts failing at an earlier commit.) If the project does not build or work with PIT at some commit, we go back one commit at a time until we obtain a version that works. Note that we regenerate new mutants for every software version. We do not sample mutants but rather use all mutation operators available in PIT. After collecting data for all versions for a project, we automatically remove from each version the tests that were manually renamed/moved in subsequent versions.

Figure 2 shows, for each project, the number of versions used in our experiments along with some statistics about them. Columns 3-5 show the min, median, and max lines of code for the project across the versions used in the experiments, obtained using SLOCCount [6]. Columns 6-8 show min, median, and max number of tests across the versions used in the experiments. Columns 9-11 show min, median, and max number of mutants generated by PIT. The last three columns show min, median, and max mutation score, as reported by PIT. The last row in Figure 2 shows mean values across all 131 versions.

4.3 Single Version Evaluation

In this section, we report the quality of the reduced test suites by SizRed, StmtLoss, and MutLoss for a single software version at a time, i.e., diagonal elements in Figure 1. Figure 3 summarizes the results. The values are calculated across all 18 projects. SAR, MAR, and SMAR columns show the values obtained by the Greedy algorithm using covered statements, killed mutants, or both as requirements, respectively. (The values for other algorithms differ only marginally; we further discuss difference between algorithms in Section 5.) We compare the Greedy algorithm instances for various requirements among each other and with our baseline random reduction. We consider three randomly reduced test suites with the same sizes as the sizes of test suites reduced through the Greedy algorithm with the various requirements. We denote the algorithm on which the size is based in the subscript (e.g., Rand_{MAR} is random selection with the same size as Greedy with MAR requirements).

Random Reduction The Rand columns in Figure 3 show that random reduction performs worse than Greedy, regardless of the requirements. With the substantial loss in statement coverage and killed mutants of up to 55.2% and 55.7%, respectively, we use random reduction only for research comparison and do not suggest its use in practice.

Adequate Reduction As anticipated, SAR and SMAR achieve the same statement coverage as the original test suite, while MAR and SMAR achieve the same killed mutants as the original test suite. Statement coverage for MAR can be less than the original, ranging from 0.0% to 14.9% (median of 2.1%) loss in statement coverage. For SAR, the original loss in killed mutants ranges from 0.2% to 20.5%

[4]We consider only commits on the master branch to properly track the main (linear) software evolution.

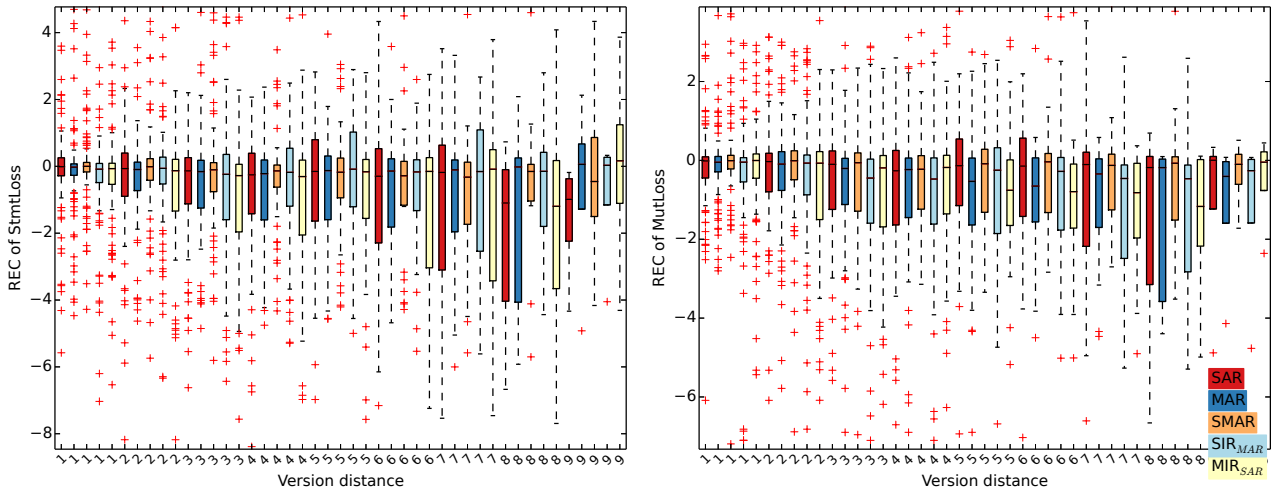

Figure 4: REC for StmtLoss (left) and MutLoss (right) for suites reduced randomly at the same size of their corresponding Greedy reduced test suites

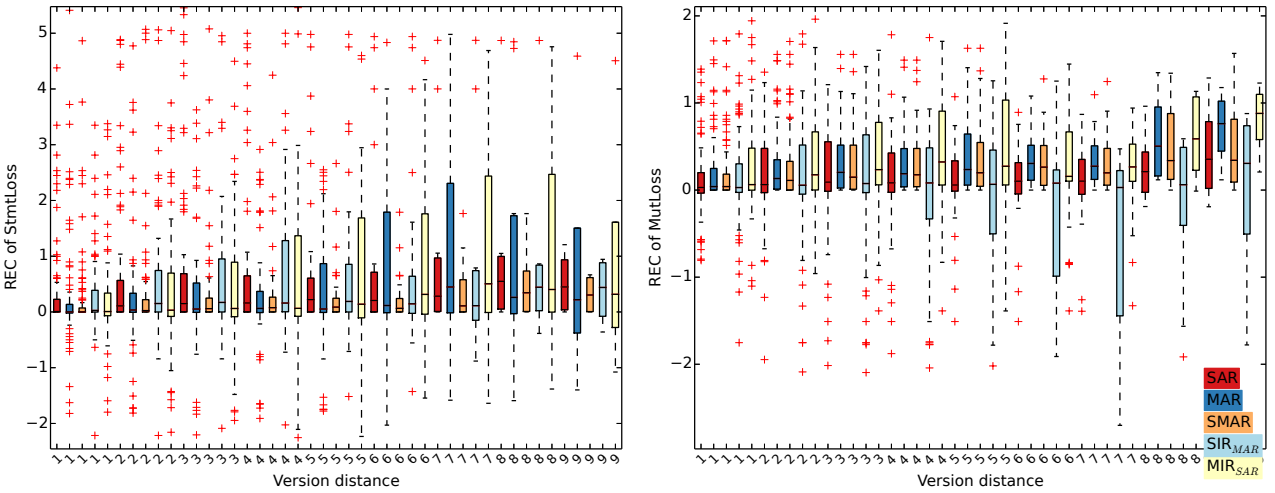

Figure 5: REC for StmtLoss (left) and MutLoss (right) for suites reduced using Greedy

(median of 3.5%). Based on these metrics, SMAR achieves the best for both requirements, having no initial loss in statement coverage or killed mutants. The size of the test suites shows that SAR gives the largest reductions in test-suite size, ranging from 29.2% to 92.9% (median of 62.9%). SMAR gives the smallest reductions in test-suite size, ranging from 10.5% to 91.0% (median of 48.3%). MAR reductions in test-suite size fall between the other two, ranging from 10.5% to 91.5% (median of 51.0%). Considering that the most widely used traditional metric to evaluate the quality of reduced test suites is killed mutants, we believe that using MAR is preferred over SAR; MAR achieves high killed mutants, with good size reduction. However, considering that SMAR has only a median of 2.7pp loss in reduction of test-suite size compared to MAR, while SMAR achieves maximum in both requirements, we believe SMAR could be the best approach to test-suite reduction if sufficient resources are available to run the reduced test suites.

In summary, the loss in killed mutants for the traditional SAR does not rise above 20.5%. In comparison with some prior studies that found significant losses (around 40%-50% in most cases) in fault-detection capability on the version of reduction [22, 24, 29, 31, 32], our results show that test-suite reduction has less significant losses on killed mutants, similar as reported by some other studies [35, 36, 41].

4.4 Evolution-Aware Evaluation

Following the methodology described in Section 3, we evaluate how different test-suite reduction techniques perform over 131 versions of all projects. Similar to the previous section, we compare the Greedy algorithm, instantiated for various requirements, with our baseline random reduction. We compare the REC trends for all these techniques by comparing the stability of REC values. To show the distribution of RECs, we use boxplots. Each boxplot (figures 4 and 5) shows a distribution of RECs for all projects for one of the distance values; the i-th boxplot shows distribution of all REC$_i$. We use red ●, blue ●, and orange ● to illustrate how the number of killed mutants of the reduced test suite varies as software evolves for SAR, MAR, and SMAR respectively.

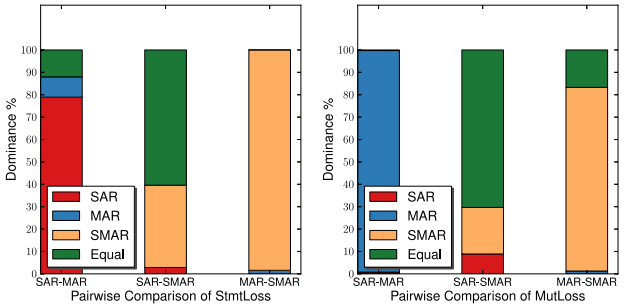

Figure 6: Pairwise value comparison for statement coverage (left) and mutants (right) over versions for Greedy reduced test suites

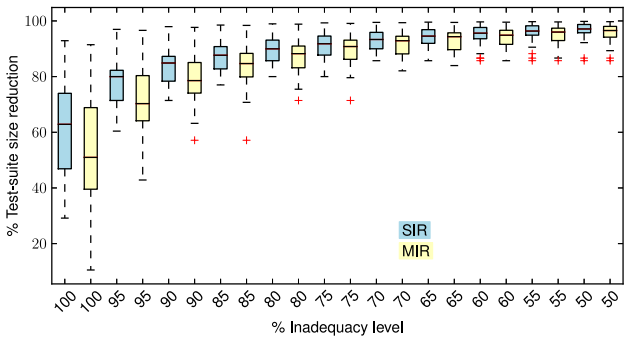

Figure 7: Reduction in size for inadequate test suites

(The light blue ○ and yellow ○ boxplots are for MIR and SIR as described later.)

Random Reduction Figure 4 shows the REC trends for the randomly reduced test suites. The left and right plots in Figure 4 show trends for statement loss RECs and mutant loss RECs, respectively. We cut the boxplots to only show 10% over the whiskers for better visualization. First, we can observe that random reduction variants (i.e., various targeted size) achieve similar stability. Second, REC of statement coverage is less stable than REC of killed mutants for all three sizes considered. While statement coverage can be more significantly affected (-8pp to 4pp), REC of killed mutants changes less as software evolves (-6pp to 2pp). This surprising result—higher stability—illustrates that the killed mutants of reduced test suites are not influenced even after a large number of commits.

Adequate Reduction Figure 5 shows the REC trends for SAR, MAR, and SMAR (that use the Greedy algorithm); the left and right plots in Figure 5 show trends for statement loss and mutant loss RECs, respectively. We noticed, similar to random reduction, that REC for statement loss (ranging from -3pp to 6pp) is less stable than REC for mutant loss (-3pp to 2pp). For all requirements used for reduction, the overall trend is a slight increase in loss over versions. Surprisingly, although SAR starts at a lower level of killed mutants, it is on average slightly more stable than MAR and SMAR. For all of SAR, MAR, or SMAR, the loss in killed mutants is at most 15.76pp from the initial loss in killed mutants due to reduction, regardless of which type of reduction is performed.

Figure 6 additionally summarizes the comparison of all techniques across multiple versions of all projects. For each evolution point (i.e., each element in the matrix in Figure 1) we do a pairwise comparison of SAR, MAR, and SMAR and measure how many times each is better (dominant), equal, or worse than the other. Similar to our previous conclusions, we conclude that in terms of killed mutants, MAR and SMAR are the best techniques to use.

4.5 Inadequate Reduction

Finally, we compare inadequate reduction (Section 3.3) with random and adequate reduction. We perform two inadequate reductions: (1) SIR_{MAR} is coverage-based reduction targeting the same statement coverage as achieved by MAR and (2) MIR_{SAR} is mutant-based reduction targeting the same percentage of killed mutants as achieved by SAR. Figure 3 includes values for SIR_{MAR} and MIR_{SAR} at a single software version (last two columns). Further, figures 4 and 5 include stability for inadequate reduction. We use light blue ○ and yellow ○ for SIR and MIR, respectively. Although the test suites produced by inadequate reduction are similarly stable as test suites produced by other reduction algorithms (figures 4 and 5), these test suites lead to higher initial loss in killed mutants and statement coverage (Figure 3).

To further explore the trade-offs of inadequate reduction, we evaluated how it targets a range of values for sizes of reduced test suites. Figure 7 shows how the size of a reduced test suite changes based on the amount of loss in requirements we set for inadequate reduction. The horizontal axis shows the inadequacy level for the requirements we set for the reduction algorithm, and the vertical axis shows the reduction in the test-suite size (compared to the original test suite). The boxplots are drawn from data gathered from reducing on all projects and versions. We find that for adequate reduction, the size of the reduced test suite is (median of) 62.9% of the original test-suite size (first boxplot); by dropping down to 95% of the requirements, the reduction in size of the reduced test suite can increase 17.14pp (third boxplot). Further decreases in the inadequacy level reduces the size of the reduced test suite even more, eventually reducing 34.25pp from the median reduction in test-suite size of the adequately reduced test suite when only 50% of the requirements are satisfied. At an extreme of 10%, the median of reduction in test-suite size goes up to 99.77%, which is 36.90pp more reduction in size than the adequately reduced test suite. These trends are seen in both statement coverage and killed mutants requirements.

Additional Results: Due to space constraints, we only report detailed results for Greedy algorithm. Additional results for other algorithms along with details about the projects and versions that we used can be found at

`http://mir.cs.illinois.edu/evolred`

5. DISCUSSION

Reduction Algorithms While the previous section reported and visualized values obtained by running Greedy algorithm, we also evaluated other algorithms mentioned in Section 2 (i.e., GE, GRE, HGS, and ILP). We found that they produce very similar results to Greedy. Across all algorithms, we found the difference in size reduction to be at most 5.26pp for all types of requirements. Furthermore, the difference in StmtLoss for algorithms that use MAR is at

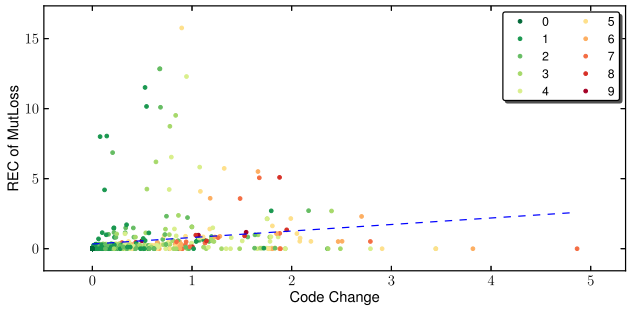

Figure 8: Correlation between code change and REC of mutation score loss for Greedy MAR

most 4.15pp, and the difference in MutLoss for algorithms that use SAR is at most 7.15pp. For stability, we found that the median REC_i, at every distance i, varied by at most 0.33pp for MutLoss and 0.67pp for StmtLoss across all algorithms.

SIR and MIR We instantiate and evaluate SIR and MIR with inadequacy levels based on the statement coverage and killed mutants results of MAR and SAR, respectively. We choose to set the inadequacy to these levels to compare how the inadequately reduced test suites compare to the adequately reduced test suites, i.e., comparing the killed mutants of a test suite with the same statement coverage as a MAR test suite that has no loss in killed mutants.

Code Change vs. Loss We quantify how the *amount of code change* correlates with loss in requirements satisfied by the reduced test suite.

DEFINITION 6. *We quantify code change between two versions i and j by using normalized change:*

$$CC_i^j = (Removed_i^j + Added_i^j)/SLOC_i, \text{ where } i \leq j$$

where $Removed_i^j$ and $Added_i^j$ represent the number of removed and added lines, respectively, between versions i and j. We obtained these values by using Git's diff command. $SLOC_i$ represents the total number of source lines of code at version i.

Figure 8 plots the correlation between the amount of code change and the REC for killed mutants. The x-axis shows the CC, and the y-axis shows the REC for MutLoss. We use different colors for each distance, e.g., for all (CC_i^j, REC_i^j) for which $\mathsf{dist}(i,j)$ is 1 (i.e., subsequent versions), we use green. We also measure the correlation between the amount of change and REC for StmtLoss and MutLoss. We computed coefficient of determination (R^2), Spearman's and Pearson's coefficients for all algorithms. The data shows a weak correlation between change and RECs. Across all algorithms and reduction criterion, R^2 varies between 0.013 and 0.509, Spearman's coefficient varies between 0.09 and 0.83, and Pearson's coefficient varies between 0.03 and 0.71.

Statement Loss for MAR We find that MAR may not achieve the same statement coverage as the original test suite. This can happen for several reasons. First, there might be statements that cannot be mutated by any mutation operator. Second, although reached, some mutants might not be killed by the original test suite (e.g., a mutant is equivalent or there is no test oracle that fails). In either of the two cases, it can happen that some statements are not covered by a test suite produced by MAR, although the original test suite covers them.

6. THREATS TO VALIDITY

External (projects and tools): Our results may not generalize to projects beyond the scope of those used in our evaluation. To mitigate this threat, we considered *actively developed* projects with *real evolution* from GitHub. Although we excluded some projects due to the fact that PIT could not run their tests, we used 18 projects that vary in size, number of developers, number of tests, and mutation score (23.28%–94.60%). Several of the projects used in our study are larger and include more tests than any project used in previous studies on test-suite reduction [11,13,14,22,24,29,31,32,35–37,41,42].

We used PIT [5] to collect statement coverage and perform mutation testing. The results might not be generalizable to other tools that perform similar functionality, e.g., Javalanche [3] or Major [27]. We used PIT because it scales to large projects, and Major was not available at the time of our study. In addition, PIT has been used in research [20] and in industry (most notably in the Apache Lucene project [1]).

Internal (correctness of our implementation): To the best of our knowledge, no implementations of the test-suite reduction algorithms mentioned in Section 2.2 are publicly available. Therefore, we implemented those algorithms ourselves, following the usual literature [11,13,14,23,41,42]. To ensure the correctness of our implementation, we inspected the results of several small runs, wrote unit-tests, and peer-reviewed our code.

Construct (metrics and versions): We measure the fault-detection capability of test suites using mutation score. Although mutants are not real software faults, previous research has shown that a test suite's ability to kill mutants can highly correlate with its ability to reveal actual software faults [8,9,17,28,41]. Another threat is that we do not identify and remove equivalent mutants generated by PIT. Note however that the users of a mutation-testing tool would likely not identify equivalent mutants either and would base their evaluation of the test suites on all generated mutants.

Measuring statement coverage loss for SAR and mutants killed loss for MAR may bias the results. To mitigate this threat, we measure reduction in size, statement coverage loss, and mutants killed loss regardless of the reduction technique used. In addition we measure these values at multiple versions (that are far apart). We assume mutants approximate real faults on a single version [28]. Therefore, as we regenerate mutants on future versions, we expect the stability of the real-faults detected loss to mirror the stability of the mutants killed loss.

Previous research on test-suite reduction has used different kinds of code coverage to perform reduction. We compare with the traditional approach to test-suite reduction using only statement coverage. However, statement coverage is the most commonly used criterion in practice, and it is widely used in research on test-suite reduction [11,13,14,41].

We define a version for a project to be 30 commits, which covers a significant portion of software evolution history for the 18 projects we used in our evaluation. Regardless of the difference in scale of changes for versions between different projects, our conclusions remained the same.

7. RELATED WORK

Elbaum et al. [18] investigated the effect of software evolution on coverage. While we also measure how coverage is impacted by software evolution, we do this for reduced test suites. Our findings (on reduced test suites) seemingly contradict theirs (on original test suites): we find that coverage of reduced test suites, *relative* to the original test suite, is not greatly impacted by the changes; note that this can hold even if the *absolute* values of both reduced and original test suites are greatly impacted by the changes.

Previous studies on test-suite reduction find conflicting results on the fault-detection capability of reduced test suites. Wong et al. [35, 36] reported insignificant losses in fault-detection capability when performing test-suite reduction on small programs. Zhang et al. [41] also found small losses in test-suite reduction evaluated on Java programs from the Software Infrastructure Repository [16]. In contrast, Rothermel et al. [31, 32] found significant losses in fault-detection capability in their evaluation on the same programs that Wong et al. used. Our results show that test-suite reduction does not significantly impact killed mutants for the projects we studied. We find the loss in killed mutants does not vary due to software evolution either. Our evaluation was conducted on a large number of actively developed real Java projects downloaded from GitHub as opposed to curated projects from the Software Infrastructure Repository. Our choice of using mutants and projects could explain some differences between our results and the results of previous research.

Previous research also looked into improving the fault-detection capability of reduced test suites by adding in extra tests beyond those necessary to preserve code coverage requirements. Jeffrey and Gupta [24] suggested adding in extra tests which cover a secondary set of requirements, e.g., reducing first based on requirements defined by block coverage but then adding in tests that satisfy a set of requirements defined by def-use coverage. Lin and Huang [29] proposed adding tests based on a secondary set of requirements only when breaking ties in traditional reduction algorithms. Our SMAR technique is similar to these previously proposed algorithms in that we use a secondary set of requirements to gain higher quality reduced test suites. However, we use killed mutants as another set of requirements as opposed to some other type of structural code coverage.

Previous research also considered fault-detection capability to some degree when performing test-suite reduction. Black et al. [11] proposed a bi-criteria ILP solution to test-suite reduction. While the ILP formulation they proposed would find a minimal set of tests that cover all statements, it could be configured to take seeded faults into account. Their ILP algorithm can be used to add tests known to detect at least one fault. At the extreme, all tests which are known to detect faults would be added to the reduced test suite. Hao et al. [22] proposed a reduction algorithm that seeks to reduce the loss in killed mutants. They collect statistics about killed mutants on the individual statement level of various programs and apply those statistics to perform reduction using an ILP formulation. We differ from both these algorithms in that we directly reduce based on killed mutants as requirements: we remove redundant tests based on mutants unlike the bi-criteria ILP algorithm which keeps redundant tests, and we directly apply killed mutants instead of using statistics about them.

Yoo and Harman [37] proposed and evaluated Pareto-optimal multi-objective algorithms for regression testing that explored the Pareto frontier where a specified requirement could be set to an inadequate level. While they mainly focused on test-suite selection and prioritization, their work can be applied to inadequate reduction as well. Moreover, they considered multiple metrics (code coverage, execution cost, and fault history) while we use only statement coverage and killed mutants as our requirements. More generally, inadequate reduction is similar to test prioritization or unsafe test selection that execute only a subset of tests from the original test suite. For example, Elbaum et al. [19] evaluated a number of prioritization techniques, including two that use fault-exposing-potential based on mutants. Our evaluations based on SAR and SMAR are larger than any previous evaluation, and we are the first to consider evolution effects on inadequate reduction.

Offutt et al. [30] described a test-suite reduction algorithm called *ping-pong* which relied on mutation testing. The algorithm orders the tests in the test suite based on heuristics and runs the tests in those orders until all mutants are killed. Their work uses mutants as requirements for test-suite reduction, but the ping-pong algorithm does not compute all the mutants killed by each test; rather, it computes only the additional (previously non-killed) mutants that are killed by each test when run. In contrast, SAR and SMAR use the full test-to-mutants-killed matrix to compute the reduced test suites. Although computing the full matrix is more expensive, the reduced test suites that SAR and SMAR compute cannot be larger than the ones computed by ping-pong.

8. CONCLUSIONS

We evaluated several trade-offs in test-suite reduction algorithms that balance the goals of high reductions in size and small losses in code coverage and fault-detection capability. Specifically, we evaluated adequate test-suite reduction with requirements defined by killed mutants, evaluated inadequate reduction that conducts reduction without satisfying all the requirements as the original test suite, and introduced novel evaluation metrics based on software evolution. We performed an extensive evaluation on 18 projects with 261,235 tests, 3,590 commits, and representing 35 years of total development time. We find that the reduction based on killed mutants is more attractive because it produces reduced test suites with the same, stable killed mutants as the original test suite, but it does have slightly larger reduced test suites compared to the reduction based on statement coverage. In addition, inadequate reduction can achieve significantly higher reductions in test-suite size. Finally, we find test-suite reduction algorithms to be robust to software evolution, regardless of requirements used.

9. ACKNOWLEDGMENTS

We thank Caius Brindescu, Mihai Codoban, Jessy Li, Marius Minea, and the anonymous reviewers for feedback on a draft of this paper; Tao Xie for the insightful comments; students from CS598TX at the University of Illinois for constructive discussions; and Lingming Zhang for providing some initial data. This research was partially supported by the NSF Grant Nos. CNS-0958199, CCF-1012759, and CCF-1439957. August Shi was supported by the Ray Ozzie Computer Science Fellowship.

10. REFERENCES

[1] Apache Lucene. https://lucene.apache.org/.

[2] IBM CPLEX Optimizer. http://www.ibm.com/software/commerce/optimization/cplex-optimizer/.

[3] Javalanche. http://javalanche.org/.

[4] Maven. http://maven.apache.org/.

[5] PIT mutation testing. http://pitest.org/.

[6] SLOCCount. http://www.dwheeler.com/sloccount/.

[7] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? A characterization of open source software repositories. In *ICPC*, pages 182–191, 2008.

[8] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

[9] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *TSE*, 32(8):608–624, 2006.

[10] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining Git. In *MSR*, pages 1–10, 2009.

[11] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE*, pages 106–115, 2004.

[12] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *ICSE*, pages 322–333, 2014.

[13] T. Chen and M. Lau. A new heuristic for test suite reduction. *IST*, 40(5-6):347–354, 1998.

[14] T. Y. Chen and M. F. Lau. A simulation study on some heuristics for test suite reduction. *IST*, 40(13):777–787, 1998.

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

[16] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[17] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE*, 32(9):733–752, 2006.

[18] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *ICSM*, pages 170–179, 2001.

[19] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *TSE*, 28(2):159–182, 2002.

[20] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation for developers. In *ICSE*, pages 72–82, 2014.

[21] R. G. Hamlet. Testing programs with the aid of a compiler. *TSE*, 3(4):279–290, 1977.

[22] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *ICSE*, pages 738–748, 2012.

[23] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *TOSEM*, 2(3):270–285, 1993.

[24] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *TSE*, 33(2):108–123, 2007.

[25] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[26] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, pages 92–102, 2001.

[27] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*, pages 315–326, 2014.

[28] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, 2014. to appear.

[29] J.-W. Lin and C.-Y. Huang. Analysis of test suite reduction with enhanced tie-breaking techniques. *IST*, 51(4):679–690, 2009.

[30] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *ICTCS*, pages 111–123, 1995.

[31] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998.

[32] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *STVR*, 12(4):219–249, 2002.

[33] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–189, 1999.

[34] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002.

[35] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE*, pages 41–50, 1995.

[36] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *COMPSAC*, pages 522–529, 1997.

[37] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *ISSTA*, pages 140–150, 2007.

[38] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[39] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ICSE*, pages 435–444, 2010.

[40] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*, pages 235–245, 2013.

[41] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *ISSRE*, pages 170–179, 2011.

[42] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *IST*, 50(6):534–546, 2008.