Regression Test-Suite Reduction: Looking into the Future

August Shi
Department of Computer Science
University of Illinois
USA
awshi2@illinois.edu

Alex Gyori
Department of Computer Science
University of Illinois
USA
gyori@illinois.edu

Abstract—Regression test suites tend to grow to the point where running them becomes expensive. Previous research has shown that developers/testers tend to introduce redundant tests, i.e., tests that cover the same line/branches or kill the same mutants. Test-suite reduction techniques have explored this problem by proposing the use of different coverage criteria to eliminate redundant tests. Previous research evaluated reduction techniques by measuring the reduction both in terms of number of tests and in terms of reduction of fault-detection capability of the reduced test suite. To the best of our knowledge, all previous research evaluated reduction techniques exclusively on a single version of a program – the same version which was used to perform the reduction itself. Because test-suite reduction is regarded as a technique that drops tests permanently, it would be useful to evaluate what the reduction in fault-detection capability is on future versions.

In this empirical study, we measure how the fault-detection capability of the reduced test suite varies over multiple versions. We propose metrics that consider software evolution. We evaluate on two projects used in previous research from the SIR repository and one widely used open-source project. Our findings show that software evolution negatively influences the fault-detection capability of the reduced test suite. We find that the fault-detection capability of the reduced test suite can drop to 50% of the original test suite. Coverage of the reduced test suite can also drop to 88% of the original test suite as software evolves.

I. INTRODUCTION

Regression testing involves retesting existing software after a change has been made in order to ensure that previously working functionality is not broken by the introduced changes [1]. Ideally, the testing process should be fast, in order to detect newly introduced faults as early as possible. Unfortunately, previous research has found real-world regression test suites that take between one and seven weeks to finish [2], [3]. This is partly due to the fact that the number of tests that have to be run after a change can grow extremely large.

Tests are usually written as software is developed either in a test-driven development fashion [4] or after the functionality has been added. New tests are also introduced whenever changes are made. Because tests for the same feature are written at different points in time and by different persons, redundant tests can be introduced. The extra redundant tests make the regression testing process unnecessarily expensive in terms of time.

Redundancy is measured with respect to a testing requirement. Previous research has focused on code coverage as a measure of redundancy [5]–[9]. The identification of such redundancy led to intense research in reducing the test-suite size by eliminating tests that are redundant with respect to the testing requirements. This led to techniques that remove tests without affecting the overall coverage achieved by the test suite. While meeting testing requirements is important this does not guarantee the absence of bugs. Moreover, removing tests can negatively impact the fault-detection capability of the test suite. The decrease is generally measured in terms of the reduced test suite’s capability to detect faults compared to the original test suite’s capability to detect faults. Previously, researchers have used either mutation score or manually seeded faults to measure capability of test suite to detect faults.

Previous research on test-suite reduction [6], [7], [9]–[12] used two metrics to evaluate the quality of the reduction techniques: reduction in test-suite size and fault-detection capability. The fault-detection capability is most commonly measured as the ratio of the number of faults detected by the reduced test suite to the number of faults detected by the original test suite. The results reported by prior techniques show relatively high reduction in test-suite size and varying loss of fault-detection capability. Rothermel et al. [11] found that the fault-detection capability of the reduced test suite is severely lowered whereas Wong et al. [10], [13] found that the fault detection capability does not substantially decrease.

However, these evaluations [5]–[13] were conducted on a single program version without considering the impact of software evolution on the fault-detection capability of the reduced test suite. Therefore it is unclear what is the effect of evolution on the fault-detection capability of the reduced test suite. The implicit assumption is that evolution does not influence the fault-detection capability of the reduced test suite. In this research we question this assumption and evaluate how well the reduced test suite behaves as software evolves.

This paper makes the following contributions:

RC1. Metrics: We propose a set of new metrics to measure the effectiveness of regression test-suite reduction techniques for evolving software. The metrics account for size, coverage, and fault-detection capability.

RC2. Empirical study: We evaluate how the effectiveness of reduced test suites changes as software evolves for two different reduction techniques using the newly proposed...
metrics. We evaluate on two SIR projects and one widely used and actively developed open-source project.

Section 2 introduces the techniques we use in this paper. Section 3 describes the metrics we propose. Section 4 describes the results of our empirical study. Section 5 provides some discussion on the results. Section 6 presents the threats to validity together with the actions we took to mitigate them. Section 7 presents related work, and Section 8 concludes.

II. TECHNIQUES

In this section, we present the necessary background information on the techniques we use in this empirical study. We present the coverage, test-suite reduction, and mutation-testing techniques we use in this empirical study.

A. Coverage

Code coverage is a widely used metric for measuring the quality of a test suite. Various forms of coverage are used by research and industry for this purpose, e.g., statement coverage, branch coverage, predicate coverage. Previous research has primarily used statement coverage as a testing requirement to detect redundant tests [6], [7], [14], [15]. Statement coverage measures how many of the statements of the software are executed during a test-suite run. In this research, we also use statement coverage to reduce the test suite, as presented in [14], [15]. We also use statement coverage to evaluate how the test suite degrades over future versions. For evaluating the quality of the test suite other kinds of coverage could be used. We leave the usage of different coverage types for future work.

B. Reduction Techniques

In this section, we describe the two reduction techniques we evaluated. Finding a minimal subset of a test suite that satisfies the same testing requirement is an NP-complete problem [5]. All reduction techniques generalize based on some test requirements. We use statement coverage as a test requirement for our empirical study. Therefore, all of the reduction techniques create reduced test suites that achieve the same amount of statement coverage as the original test suite for the current version.

1) Greedy Technique: The greedy technique [14] selects tests from the original test suite based on the number of statements each test covers, constructing a reduced test suite. Given an original test suite \( T \) and the set of statements it covers \( S \), the greedy technique iteratively selects tests in \( T \) which cover the most uncovered statements from the set \( S \) until it achieves fix-point. Starting with an empty set of tests \( T_m \) representing the reduced test suite and an empty set of covered statements \( S_c \), representing statements covered by \( T_m \), the greedy algorithm selects at each step a test \( t \) in \( T \) which covers the most number of statements not in \( S_c \). The selected test is added to \( T_m \) and the statements it covers are added to \( S_c \). This process continues until \( |S_c| = |S| \), i.e., when the reduced test suite covers all the statements that the original test suite covers. The greedy algorithm does not achieve the minimum-sized test suite. Furthermore, different strategies can be used for tie-breaking when selecting the next test to add to \( T_m \). We currently randomly select one of the candidate tests and do not consider other possible reduced test suites.

2) ILP Technique: The ILP technique [15] encodes the coverage requirements and tests selected into an integer linear programming problem. The objective of the integer linear programming problem is to minimize the number of tests selected while meeting the constraint of covering all statements covered by the original test suite. The problem consists then of two equations:

Objective: \( \min \left( \sum_{i=1}^{n} x_i \right) \), \( x_i \in \{0, 1\} \)

Constraint: \( \bigwedge_{i=1}^{n} (\sum_{j=1}^{m} s_{i,j} \geq 1), s_{i,j} \in \{0, 1\} \)

where \( n \) represents the size of the test suite, \( x_i \) represents the selection status of test \( i \), \( x_i = 1 \) denotes the \( i^{th} \) test case being selected and \( x_i = 0 \) otherwise. \( s_{i,j} \) represents statement \( j \) being covered by test \( i \), \( m \) represents the number of statements covered by the original test suite, \( s_{i,j} = 1 \) when statement \( j \) is covered by test \( i \) and \( s_{i,j} = 0 \) otherwise.

With these definitions, the solver minimizes the number of tests to be selected while ensuring all \( m \) statements covered by the original test suite are covered by the selected tests.

C. Mutation Testing

Mutation testing is a widely used technique for evaluating the quality of a test suite [6], [7], [16]–[18]. The idea is to select a set of mutations to apply on a program and check if the tests are able to detect the mutant. Previous research has measured the effectiveness of reduction techniques by comparing the fault-detection capability of the reduced test suite to the fault-detection capability of the original [14], [15].

III. METRICS

We are the first to propose metrics that look into how the quality of the reduced test suite varies over several versions. Previous research only measured the fault-detection capability of the reduced test suite on the current version. Furthermore, we propose several other metrics besides fault-detection capability to better characterize the quality of the test suite over a range of versions. We argue these metrics give better insight into the effectiveness of test-suite reduction techniques than previously reported metrics. These metrics account for change in software, which is omnipresent in the current software development process. They give better insight on the influence of software evolution on reduction techniques. They can be used to fine-tune reduction, e.g., deciding when the reduced test suite needs to be recomputed.

In this section we describe the metrics we use, and how these metrics are relevant to measure the effects of evolution on reduced test suites. We also describe how we use these metrics as we carry out our empirical study.

A. Evolution-Aware Metrics

The main goal of our empirical study is to investigate how software evolution affects the effectiveness of test suites that are reduced using existing test-suite reduction techniques. To
do so, we propose metrics that measure the effectiveness of test suites over several versions.

In this section, we use the following notation to refer to the set of tests representing the original test suite and the reduced test suite:

- $T_i$ represents the set of tests from the original test suite of version $i$.
- $R_i$ represents the set of tests in the reduced test suite computed using version $i$.
- $T^i_j$ and $R^i_j$ represent $T_i$ and $R_i$ with tests from version $i$ evolved to version $j$, respectively. This test evolution accounts for test renames, test code being moved or split.

Since we are evaluating on multiple program versions, we commonly use $i$ and $j$ to refer to program versions, with $i \leq j$. In this context, $i \leq j$ represents the fact that version $i$ was committed before or at the same time as version $j$.

**Size Variation**

We measure how the size of reduced test suites varies over future versions with respect to the entire test suite. The size of the reduced test suite can change over time due to addition or removal of tests. Tests from the reduced test suite can be removed, tests not from the reduced test suite can be removed, tests can be added, or tests can be changed. We are more concerned about the removal of tests from the reduced test suite. Tests removed from the reduced test suite can have a more damaging effect as the reduced test suite has less redundancy than the original one. Moreover, the fact that tests kept in the reduced test suite are removed in further versions can indicate that developers found them unimportant. Previous research has measured the reduction in test-suite size on a single version by using $|R_i| / |T_i|$. We measure how the reduced test-suite size varies over multiple versions, to account for the impact of evolution on test-suite size.

**Definition 1:** Test suite size reduction between two versions $i$ and $j$ is:

$$\text{SizeReduction}_{i,j} = \frac{|R^i_j \cap T^i_j|}{|T^i_j|} \quad \text{where } i \leq j$$

By computing $|R^i_j \cap T^i_j|$ and $|T^i_j|$ we measure the size of the test suites by counting only the tests that were kept in later versions. Their ratio gives an insight into how tests evolve and how the reduction rate is influenced.

**Coverage Variation**

We measure how the coverage of the reduced test suite with respect to the coverage of the original test suite varies over versions. The original test suite and the reduced one achieve the same coverage by construction on the current version. This fact is confirmed by the traditional metric that compares coverage of a subset of tests with respect to the entire test suite: $\frac{\text{Cov}_i(R_i)}{\text{Cov}_i(T_i)}$, where $\text{Cov}_i(X)$ represents the number of elements, e.g., statements, covered by test suite $X$ on version $i$. Elements can be statements, branches, predicates and others depending on the coverage criteria used. We report results using statement coverage.

While this previous metric performs well for individual versions, it is unable to account for the effects of software evolution on coverage, because the coverage of the reduced test suite can vary as code is evolving. Therefore, we measure how the reduced test suite’s coverage varies over versions. We compare the coverage achieved by the reduced test suite with coverage achieved by the original one. We use two metrics: one includes the coverage of the newly added tests, while the other does not.

**Definition 2:** Coverage reduction between two versions $i$ and $j$ is:

$$\text{CovReduction}_{i,j} = \frac{\text{Cov}_i(R^i_j \cap T^i_j)}{\text{Cov}_i(T^i_j)} \quad \text{where } i \leq j$$

Coverage reduction does not consider the tests that were added, and only measures how the coverage of the reduced test suite changes over versions.

In contrast, augmented coverage reduction accounts also for the coverage of the newly added tests and gives a more global comparison between the scenario in which reduction is performed and the scenario in which it is not.

**Definition 3:** Augmented coverage reduction between two versions $i$ and $j$ is:

$$\text{AugmCovReduction}_{i,j} = \frac{\text{Cov}_i((R^i_j \cap T^i_j) \cup (T^j_i \setminus T^i_j))}{\text{Cov}_i(T^i_j)} \quad \text{where } i \leq j$$

Note that $T_j = (T^i_j \cap T^i_j) \cup (T^j_i \setminus T^i_j)$

**Fault-Detection Capability Variation**

We measure how the fault-detection capability varies between versions for the reduced test suite. Previous research only evaluated the effectiveness of the reduced test suite on the version the reduction was performed. The effectiveness was measured using $\frac{\text{Mut}_i(R_i)}{\text{Mut}_i(T_i)}$, where $\text{Mut}_i(X)$ represents the number of mutants killed by test suite $X$ on version $i$.

We argue that such evaluation is insufficient due to software evolution, and we measure how the fault-detection capability varies between versions. Fault-detection capability reduction measures the impact of evolution on the effectiveness of the reduced test suite.

**Definition 4:** Fault-detection capability reduction between two versions $i$ and $j$ is:

$$\text{FDReduction}_{i,j} = \frac{\text{Mut}_i(R^i_j \cap T^i_j)}{\text{Mut}_i(T^i_j)} \quad \text{where } i \leq j$$

Augmented fault-detection capability reduction also accounts for the newly introduced tests and provides for a more global comparison between the scenarios in which reduction is used and the scenario in which it is not used.

**Definition 5:** Augmented fault-detection capability reduction between two versions $i$ and $j$ is:

$$\text{AugmFDReduction}_{i,j} = \frac{\text{Mut}_i((R^i_j \cap T^i_j) \cup (T^j_i \setminus T^i_j))}{\text{Mut}_i(T^i_j)} \quad \text{where } i \leq j$$

**B. Experimental Setup**

We select jTopas and xStream from the SIR repository [19] as subjects for our evaluation. We choose these projects because they have multiple versions and they were used in previous research on regression testing [6], [20]. We also choose another larger program not from the SIR repository to get a more realistic development environment. We select JFreeChart, a sufficiently large project with a large number

1http://www.jfree.org/jfreechart/
of versions and a well-maintained test suite. Details about the
sizes of the projects’ test suites, sizes of the reduced test suites,
and number of mutants killed by original test suites can be
found in Table I.

To gather preliminary data, we evaluate the effects of reduc-
tion on the effectiveness of test suites for xStream, by using
statement coverage and mutants-killed data from a previous
study [20]. Using the mutants-killed data, which consists of
mutants mapped to tests that kill it, we record the number of
mutants killed by both the original test suites and the reduced
test suites. To calculate \( FDReduction \) we measure the fault-
detection capabilities across versions for the reduced test suite
and the original test suite. We also record coverage ratios in
the same manner.

To gather more data, we used JaCoCo [21] to collect
statement coverage and Javalanche [22] for mutation testing.
We created coverage matrices which record for each test in
the test suite the statements it covers. We created mutants-killed
matrices which record for each test the mutants it kills. Based
on this data we ran our own implementations of Greedy and
ILP reduction algorithms. To implement the ILP technique we
used IBM’s CPLEX Optimizer solver [23].

For data collection we used four versions of both JFreeC-
chart and jTopas. For JFreeChart we selected versions that
are approximately one year apart from each other ending with
the latest version at the time of collection. We list the actual
version numbers in Table I so anybody can replicate our exper-
iments. For jTopas we used the four versions provided in the
SIR repository. All our data and collection scripts are publicly
available so that anybody can replicate our experiments.

IV. EXPERIMENTAL RESULTS

In this section we describe the results we gathered
by measuring the metrics proposed in Section III-A
on three different projects. The tables that follow, e.g.,
Figure 1, contain matrices reporting \( SizeReduction, \)
\( CovReduction, \) \( AugmCovReduction, \) \( FDReduction, \)
\( AugmFDReduction. \) The cell for row \( i \) and column \( j \)
corresponds to the respective value of the metric for versions
\( i \) and \( j \), respectively.

A. Size Reduction

In Table (a) from Figures 1–6, we present the variation of
size for the reduced test suite. The reduction in test-suite size
for the projects we evaluated is rather high, ranging from
46.25% down to 21.05% of the original test-suite size. The
collected data shows that the ratio of the size of the reduced
test suite to the original test suite does not vary much over
evolution. While there is some variation, which indicates tests
being removed, there is not a clear trend towards any direction.
Tests are removed from either the reduced test suite or the
complement of the reduced test suite.

For the SIR projects the size of the test suite increases
greatly across versions; for both jTopas and xStream the size
of the test suite doubles from the first to the last version (see
Table I). This is due to the fact that randomly-generated tests
are added in those projects. In contrast, the size of the test
suite for JFreeChart stays relatively constant over versions.
This indicates that in the projects we selected, tests were rarely
removed. In the SIR projects, tests were only added. This could
indicate that projects do not remove tests from their test suites.

B. Coverage Reduction

In Table (b) from Figures 1–6, we present the variation of
coverage reduction over versions for each subject and
reduction technique we use. We notice that the diagonal of
each of these tables has 100% coverage; this is achieved by
the way reduction is performed. The goal of the reduction

<table>
<thead>
<tr>
<th>Name</th>
<th>#Tests</th>
<th>#Sel G</th>
<th>#Sel ILP</th>
<th>Mutant</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart v2.340</td>
<td>2272</td>
<td>1011</td>
<td>994</td>
<td>21584</td>
<td>28934</td>
</tr>
<tr>
<td>JFreeChart v2.522</td>
<td>2351</td>
<td>1024</td>
<td>1008</td>
<td>21748</td>
<td>29006</td>
</tr>
<tr>
<td>JFreeChart v2.938</td>
<td>2155</td>
<td>987</td>
<td>975</td>
<td>20790</td>
<td>28532</td>
</tr>
<tr>
<td>JFreeChart r3.070</td>
<td>2197</td>
<td>992</td>
<td>979</td>
<td>21670</td>
<td>26705</td>
</tr>
<tr>
<td>JTopas v1.01</td>
<td>35</td>
<td>21</td>
<td>20</td>
<td>1066</td>
<td>1751</td>
</tr>
<tr>
<td>JTopas v1.02</td>
<td>35</td>
<td>21</td>
<td>20</td>
<td>1066</td>
<td>1751</td>
</tr>
<tr>
<td>JTopas v1.03</td>
<td>35</td>
<td>21</td>
<td>20</td>
<td>1066</td>
<td>1751</td>
</tr>
<tr>
<td>JTopas v1.04</td>
<td>35</td>
<td>21</td>
<td>20</td>
<td>1066</td>
<td>1751</td>
</tr>
<tr>
<td>xStream 1.20</td>
<td>637</td>
<td>210</td>
<td>210</td>
<td>5645</td>
<td>3088</td>
</tr>
<tr>
<td>xStream 1.21</td>
<td>698</td>
<td>219</td>
<td>217</td>
<td>6141</td>
<td>5273</td>
</tr>
<tr>
<td>xStream 1.22</td>
<td>765</td>
<td>236</td>
<td>234</td>
<td>6590</td>
<td>5741</td>
</tr>
<tr>
<td>xStream 1.30</td>
<td>885</td>
<td>264</td>
<td>269</td>
<td>7011</td>
<td>6326</td>
</tr>
<tr>
<td>xStream 1.31</td>
<td>924</td>
<td>271</td>
<td>265</td>
<td>8567</td>
<td>6824</td>
</tr>
<tr>
<td>xStream 1.41</td>
<td>1200</td>
<td>302</td>
<td>298</td>
<td>10022</td>
<td>7659</td>
</tr>
</tbody>
</table>
C. Augmented Coverage Reduction

In Table (c) from Figures 1–6, we present the variation of augmented coverage reduction over versions for each subject and reduction technique we use. The trends for this metric are similar to the ones for coverage reduction presented in Section IV-B. The reduction in augmented coverage reduction can drop down to 89.90% of the original test suite in the case of jTopas when using ILP (see Figure 6 Table (c)). We note that in the case of augmented coverage reduction the interpretation of the column is different. The variation in each column now represents how the reduced test suites perform with respect to each other on version \( n \). For augmented coverage, because we are adding new tests, the decrease in coverage is smaller. We find less of a downwards trend, but nevertheless the downwards trend exists.

D. Fault Detection Capability Reduction

In Table (d) from Figures 1–6, we present the variation of fault detection reduction over versions for each subject and reduction technique we use. The reduction in fault-detection capability of the reduced test suite can range from 95.57% down to 80.23% of the original test suite.

The \( n^{th} \) diagonal, i.e., cells \( < i, i + n > \) represent fault detection achieved by the reduced test suite from version \( i \) after \( n \) versions compared to the original test suite for version \( i \). We observe that as \( n \) increases, the fault-detection capability reduction ratio decreases for two of our subjects. For xStream the results are inconclusive, with no clear trend.

In Table (d) from Figures 1–6, each \( i^{th} \) line illustrates how fault-detection capability reduction varies for a test suite reduced on version \( i \) across all subsequent versions. For two of our subjects, the fault detection achieved by the reduced test suite with respect to the original one generally decreases as we get further away from the version we computed the reduced test suite. Again, xStream does not present a clear trend as can be observed in Table (d) from Figures 3, 4.

In Table (d) from Figures 1–6, each \( i^{th} \) column illustrates how the reduced test suite computed on every previous version performs on version \( i \) with respect to fault-detection capability. The data shows that the reduced test suites computed for versions closer to \( i \) have better fault-detection capability for JFreeChart and jTopas as can be seen in Table (d) from Figures 1, 2, 5, 6. For xStream the trend is inverse, i.e., reduced test suite computed on versions closer to version \( i \) perform better.
worse than test suites computed further away. We are still to investigate the causes of these observations.

The data shows that there is some variation between versions in terms of fault-detection capability. In the xStream project there is not any clear trend, the fault-detection capability fluctuates across newer versions. However, in JFreeChart and jTopas we observe a decreasing trend which indicates that the reduced test suite’s fault-detection capability drops on newer versions of software. The analysis by column shows that recomputing the reduced test suite is valuable.

E. Augmented Fault-Detection Capability Reduction

In Table (e) from Figures 1–6, we present the variation of augmented fault-detection reduction over versions for each subject and reduction technique we use. The reduction in augmented fault-detection capability of the reduced test suite can range from 97.98% down to 82.81% of the original test suite. We do not notice any clear trend diagonally, by column, or by line in the subjects we evaluate when we consider the newly added tests.

V. Discussion

We observe a large increase in test-suite size for the projects from the SIR repository. This influences our interpretation of the trends; more specifically we tend to put more weight on results and trends derived from JFreeChart, which is more indicative of real-world software.

We notice that the ILP technique always selects less tests than the Greedy technique, but not by a large margin. We also observe that Greedy outperforms ILP in terms of both coverage and fault-detection capability constantly. Greedy was also faster to compute the reduced test suite. Given that the
trends for coverage and fault-detection capability are similar between the two across versions, Greedy is preferable to ILP if software evolution is a concern.

There is a downwards trend in coverage for both augmented and non-augmented test suites. The decrease in coverage could be due to newly added features. We are mitigating this by augmenting the reduced test suite with newly added tests. The fact that there is still a loss in coverage of the augmented reduced test suite indicates that the eliminated tests would provide additional coverage in newer versions. The reduction techniques are unable to consider how the initial redundancy in tests could end up not being redundant in future versions.

While overall our mutants measurements do not show a clear trend, we notice a more clear trend in JFreeChart; we believe this trend is more indicative of real software evolution. We acknowledge that we need to evaluate on other real-world software to validate this claim. We consider that this discrepancy in conjunction with the artificial nature of the projects in SIR corroborates that these projects might not be appropriate for evaluation of test-suite reduction techniques and other regression testing techniques.

We plan to further evaluate the fault-detection capability of more reduction techniques over a number of versions of large, widely used, and actively developed projects, e.g., Apache Ant. Also, we plan to empirically provide insight into what software changes have the greatest impact on the fault-detection capability of test suites.

VI. THREATS TO VALIDITY

In this section we describe the possible threats to the validity of our empirical study. We also describe the steps we took to mitigate these threats.

A. External

Could our choice of subjects introduce errors in our conclusions? We evaluate reduction techniques on projects from the SIR repository [19], [24]. These projects are widely used in research on regression testing [6], [20]. However, they are not necessarily representative of mature, actively developed software projects. Tests are randomly generated, therefore they are not indicative of manually constructed test suites. To mitigate this concern, we chose another subject, JFreeChart, which is not from the SIR repository. This project is actively developed, widely used, and its evolution is indicative of real-world projects. While we made efforts for our results to be...
generalizable, they might not generalize beyond the scope of the projects on which we evaluated.

**Could the tools we used in our experiments introduce bias in our results?** We used JaCoCo [21], a widely used coverage collection tool in our experiments. We manually inspected the results to ensure their correctness. We used Javalanche [22] to perform mutation testing. We manually inspected the results to ensure correctness. While we made efforts to ensure no bias was introduced due to bugs in our toolset, we acknowledge that the results might be subject to bias due to bugs. Also, the results might not be generalizable to other tools that perform similar functionality, i.e., coverage collection and mutation testing.

**B. Internal**

**Are there any inherent threats in how we conducted the experiments?** We implemented ourselves the Greedy and ILP algorithms described in Section II-B. We tested and peer-reviewed our implementations to ensure correctness. Furthermore, the techniques are fairly simple, therefore correctness is easy to check.

**C. Construct**

**Are the metrics we use indicative of the actual quality of reduction techniques?** We measure fault-detection capability of test suites by using mutation testing. While mutants are not a perfect match with real software faults, they were used by previous research to measure fault-detection capability [6], [18], [20], [25]. Furthermore, research shows that a test suite’s ability to kill mutants is correlated with its likeliness to reveal actual software faults [16].

**VII. Related Work**

Previous research has investigated several aspects of test-suite reduction. The main focus was twofold: developing new test-suite reduction techniques and empirical studies on the effectiveness of existing test-suite reduction techniques.

There are several different techniques for reducing test-suite size. Harrold et al. [26] used the heuristic of selecting essential tests as early as possible into the reduced test suite. Chen et al. [14] proposed the Greedy algorithm of selecting tests based on coverage, which we use in our study. Chen et al. [27] proposed another heuristic that combines the characteristics of essential tests and 1-to-1 redundant tests along with the Greedy algorithm to select tests into the reduced test suite. Black et al. [15] use ILP models to reduce test suites, and we use their technique as part of our own evaluation. Previous research evaluated the effectiveness of these techniques in terms of test-suite size reduction and fault-detection capability on single versions of software. We evaluated the effectiveness of the Greedy and ILP techniques in terms of test-suite size reduction, coverage reduction, and loss of fault-detection capability across multiple versions software. We plan to conduct similar studies on the other techniques to quantify which techniques perform better with regards to software evolution.

Wong et al. [10], [13] researched the effect of reducing the size of test suites on their fault-detection capability. Like our evaluation, they measured test-suite size reduction and loss in fault-detection capability after reduction. Unlike our evaluation, they used block coverage as a testing requirement to guide reduction whereas we use statement coverage. They also measured fault-detection capability by manually injecting faults into programs while we use mutation testing. Finally, they evaluated reduction techniques on tests for a single software version while we considered reduction of test-suite size, coverage reduction, and loss in fault-detection capability across multiple versions.

Zhong et al. [8] focus their research on measuring the improvement of running time for the reduced test suite and the scalability of the evaluated techniques. Their measurements also focus only on a single software version. In contrast, we focus on fault-detection capability and on multiple versions. The metrics we propose could be extended for running time measurements across several versions in order to get better insight into the effectiveness of test-suite reduction techniques with respect to the running time of the reduced test suite. We leave this for future work.

Chen et al. [14] perform a simulation study on several reduction techniques. Their aim was to provide guidelines on choosing the best technique depending on the application. They find that the degree of test requirement overlap in tests influences the effectiveness of the techniques. In our empirical study we also use multiple reduction techniques and evaluate their effectiveness, but we focus on their effectiveness over several versions. We plan to investigate more reduction techniques in order to determine which technique works best with respect to software evolution.

Zhang et al. [6] conducted an empirical study on the effectiveness of four different test-suite reduction techniques in terms of reduction of test-suite size and loss in fault-detection capability. Of the four reduction techniques they use, our study uses two of those techniques, Greedy and ILP. We also measure reduction of test-suite size and loss in fault-detection capability. They also evaluated the techniques on multiple versions for three projects. However, their evaluation focused on reduction applied to each version individually without considering the versions in relation to each other. Essentially, each project version was considered a separate subject for reduction. In contrast, our evaluation considers reduction of test-suite size, coverage reduction, and loss in fault-detection capability across multiple versions and how these metrics compare between versions.

While there is extensive research that investigates the effectiveness of the reduced test-suites in terms of running time and fault detection capability, no previous work has focused on the effects of evolution on the effectiveness of the minimized test suite. Our empirical study aims to extensively investigate how software evolution influences the reduced test suite.
VIII. CONCLUSION

This paper presents a set of metrics to evaluate the effectiveness of test-suite reduction techniques with regards to software evolution. We use these metrics to evaluate two reduction techniques: Greedy and ILP. We find that there is a downwards trend of the fault-detection capability of the reduced test suite and statement coverage. This trend is more apparent in real-world open-source projects, in contrast to our SIR subjects. This indicates that regression testing research should account for evolution and use real-world subjects in order to evaluate the proposed techniques.

ACKNOWLEDGMENT

We thank Darko Marinov and Milos Gligoric for advice and support in performing this research and for the fun moments spent debugging research tools, Andrey Zaytsev for helping with data-gathering scripts, Tao Xie for being critical with our project – this helped us improve it, and Lingming Zhang for providing preliminary data from his previous research.

REFERENCES