Comparing and Combining Test-Suite Reduction and Regression Test Selection

August Shi, Tifany Yung, Alex Gyori, Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{awshi2,yung4,gyori,marinov}@illinois.edu

ABSTRACT

Regression testing is widely used to check that changes made to software do not break existing functionality, but regression test suites grow, and running them fully can become costly. Researchers have proposed test-suite reduction and regression test selection as two approaches to reduce this cost by not running some of the tests from the test suite. However, previous research has not empirically evaluated how the two approaches compare to each other, and how well a combination of these approaches performs.

We present the first extensive study that compares test-suite reduction and regression test selection approaches individually, and also evaluates a combination of the two approaches. We also propose a new criterion to measure the quality of tests with respect to software changes. Our experiments on 4,793 commits from 17 open-source projects show that regression test selection runs on average fewer tests (by 40.15pp) than test-suite reduction. However, test-suite reduction can have a high loss in fault-detection capability with respect to the changes, whereas a (safe) regression test selection has no loss. The experiments also show that a combination of the two approaches runs even fewer tests (on average 5.34pp) than regression test selection, but these tests still have a loss in fault-detection capability with respect to the changes.

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

General Terms: Experimentation, Measurement

Keywords: Regression testing, Test-suite reduction, Regression test selection

1. INTRODUCTION

Regression testing is important but time consuming. Regression testing is widely used to check that changes to a software system do not break existing functionality [23, 33]. Modern software evolves fast, with changes pushed to a repository even several times per minute [6, 7, 9], triggering regression-test runs. Additionally, regression test suites grow and often become costly to run, with previous research reporting test suites taking even weeks to run [26]. Even Google, a corporation with a lot of computing resources, reported that running all tests is prohibitive, with a quadratic growth in the test execution over time [29].

Previous research has proposed two approaches to speed up regression testing: *test-suite reduction* and *regression test selection*. Both approaches focus on running only some tests from the full test suite; we call these tests *chosen-to-run tests*. Test-suite reduction [33] aims to (permanently) remove from the full test suite those tests that are *redundant* with respect to some testing requirements, commonly with respect to statement or branch coverage. The analysis is performed once on a *single revision* of software, and the reduced test suite is used in subsequent revisions. Although running the reduced test suite can miss some faults that the full test suite detects, it may still be tolerable in some contexts to get faster test results; the full test suite could be run more rarely [13, 18, 34]. Regression test selection [33] aims to select from the full test suite those tests that are relevant to the changes made to the software. The analysis is performed on *every revision* of software to determine how the changes between revisions affect the tests. Regression test selection is *safe* if it selects all tests that *may* be affected by the changes, guaranteeing that the outcome of the tests that are not selected will not be affected by the changes.

When test engineers wish to speed up regression testing, it is unclear which approach is better to use: test-suite reduction or regression test selection. To the best of our knowledge, there is no prior empirical comparison of test-suite reduction and regression test selection. Each approach has some pros and cons. Regression test selection, when safe, selects all tests that could detect faults due to the changes, while test-suite reduction can remove some tests that could detect such faults, because test-suite reduction is not evolution-aware (i.e., it removes tests that are deemed redundant for the current revision irrespective of future changes [27]). Test-suite reduction has a low (amortized) analysis cost because it is performed on one revision, while regression test selection incurs an additional analysis cost for every revision. However, test engineers care about the total testing time that includes not only the analysis time but also the time to execute the chosen-to-run tests; it is unclear a priori which approach has lower total time.

To compare test-suite reduction and regression test selection, we would ideally measure how much time each approach takes and how many real faults it finds in the con-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786878

text of software evolution. Because measuring total time is challenging, most research on regression testing measures the number of chosen-to-run tests as a proxy for time [33]. We also measure the number of chosen-to-run tests to determine whether test-suite reduction or regression test selection yields a smaller number. Because using real faults is challenging, most research uses mutants as a proxy [17,22,27,37]. We also use mutants, but we have an additional challenge to measure the quality of the chosen-to-run tests *with respect to the changes*.

We introduce a new criterion to measure the loss of fault-detection capability of test suites with respect to changes. We refer to the faults that are related to the changes as *change-related* faults. Intuitively, these faults can be (1) directly in the code parts that changed between two revisions or (2) *latent* faults that are in the unchanged code parts but exposed by the changes. Simply reusing a criterion for evaluating a test suite on an entire given program revision (i.e., measuring the fault-detection capability of the test suite on the code revision obtained after the changes) could be misleading because it can find faults unrelated to the changes. For example, consider two test suites, T_1 that broadly covers the entire code and T_2 that covers less code overall but focuses on one module M. If the faults are randomly seeded throughout the code, T_1 can be better as it detects a higher percentage of faults than T_2. However, if the change-related faults affect the module M on which T_2 focuses, then T_2 can be better than T_1 at detecting these change-related faults. We propose a new test criterion, called *Change-Related Requirements (CRR)*, that approximates the change-related faults a test suite needs to find. We use this criterion to measure the quality of a test suite and to compare test-suite reduction and regression test selection.

To further speed up testing, we can combine the two approaches: first perform test-suite reduction on the full test suite to create a smaller, reduced test suite, and then perform regression test selection on this reduced test suite after the changes, yielding an even smaller number of tests than selecting from the full test suite. Combining the two approaches provides the greatest reduction in the number of tests. We call this combination *selection of reduction*. We empirically evaluate how much it reduces the size of the test suite to run. Considering the quality of the tests selected by regression test selection from the reduced test suite, assuming safe selection, the change-related fault-detection capability of the tests selected from the reduced test suite is as good as the change-related fault-detection capability of the reduced test suite.

We perform an empirical evaluation on 17 open-source Java projects over a total of 4,793 revisions to answer the following three questions:

RQ1: Does test-suite reduction or regression test selection yield a smaller number of tests to run on average?

RQ2: What is the change-related loss in quality of the reduced test suite, based on our CRR criterion?

RQ3: How does selection of reduction compare with the other two approaches in terms of number of tests to run?

2. BACKGROUND

We review some background on test-suite reduction and regression test selection, and including how their effectiveness is traditionally evaluated.

2.1 Test-Suite Reduction

Test-suite reduction aims to remove redundant tests from a test suite, creating a reduced test suite that is a subset of the full test suite; ideally, a developer could run fewer tests yet still detect almost the same faults as if running the full test suite. Test-suite reduction techniques find redundant tests with respect to some testing requirements, commonly defined by code coverage criteria, such as statement or branch coverage. A test is redundant with respect to a test suite if the test satisfies only the requirements already satisfied by some other tests in the test suite. For a set of tests $\mathcal{T}$, we write $req(\mathcal{T})$ for the set of requirements satisfied by $\mathcal{T}$. Given a full test suite $\mathcal{O}$, test-suite reduction constructs a reduced test suite $\mathcal{R} \subseteq \mathcal{O}$ such that $req(\mathcal{O}) = req(\mathcal{R})$. After obtaining this reduced test suite $\mathcal{R}$, a developer would use it in lieu of the full test suite $\mathcal{O}$ for testing the software as it evolves.

Previous research on test-suite reduction [24, 25, 28, 30, 31, 33, 37] measured the effectiveness of a reduced test suite by two metrics, the reduction of size and the loss in fault-detection capability, both in comparison to the full test suite. For a reduced test suite $\mathcal{R}$ constructed from the full test suite $\mathcal{O}$, the size reduction is measured by $100 \times |\mathcal{R}|/|\mathcal{O}|$; a smaller value is better as it means fewer tests need to be run. The loss in fault-detection capability is measured using a proxy for the number of faults detected, often the number of mutants killed, but sometimes also the number of statements covered or other requirements satisfied [27]. With $req(\mathcal{T})$ denoting the set of requirements satisfied by the set of tests $\mathcal{T}$, the loss in fault-detection capability is measured by $100 \times |req(\mathcal{O}) \setminus req(\mathcal{R})|/|req(\mathcal{O})|$; a smaller value is better as the number of requirements satisfied by the full test suite but not satisfied by the reduced test suite is smaller relative to the set of all requirements satisfied. Note that different kinds of requirements should be used for the reduction and evaluation; otherwise, using the same kind would trivially produce a loss of 0.

2.2 Regression Test Selection

Unlike test-suite reduction techniques that operate on only one software revision, regression test selection techniques operate on two (or more [16]) software revisions. The aim of regression test selection is to run only the tests that may be affected by the changes made between the two revisions. Regression test selection techniques are often designed to be safe with respect to the changes, i.e., a test is not selected to be run only if the software changes cannot affect the outcome of the test [23].

Given a full test suite $\mathcal{O}_i$ at revision i, regression test selection selects to run a subset of these tests $\mathcal{S}_{i,\Delta} \subseteq \mathcal{O}_i$, where Δ represents all the other inputs that regression test selection takes, including most importantly the changes between revision i and the previous revision (typically revision $i-1$, except for branching version histories [16]) but also the coverage matrix from the previous revision. The effectiveness of regression test selection is measured by the ratio of number of tests selected to run over the total number of tests, $100 \times |\mathcal{S}_{i,\Delta}|/|\mathcal{O}_i|$; a smaller value is better.

Developers use regression test selection to speed up testing with a focus on detecting change-related faults. Commonly, a developer starts from a software revision where all tests in the test suite pass, and after making some changes, the developer wants regression test selection to select (all) the tests that could fail, because those tests find new faults. In the general case, however, a developer can start from a software revision where some tests in the test suite fail, and after making some changes, the developer can use regression test selection in two scenarios: (1) select all previously failing tests even if they are not affected by the change, reminding the developer that these failing tests are due to faults not fixed by the changes, or (2) select only the tests that are affected by the changes, and if a previously passing test now fails, it is due to a change-related fault. (Note that a change-related fault is not necessarily introduced in the changed code; instead, the change can be correct by itself but could lead the execution to some previously existing, latent fault that was not executed before.) Our CRR criterion (Section 3.2) captures the second scenario, where a developer is only concerned with the new faults related to the developer's own changes and ignores other, existing faults in the code (possibly introduced by other developers) that are unrelated to whatever change the developer made.

3. METHODOLOGY

This section describes our methodology for comparing the effectiveness of test-suite reduction, regression test selection, and their combination. We first describe how we evolve reduced test suites across multiple software revisions. We then define our CRR criterion that evaluates the change-related quality of a test suite for a change at a given revision. We finally describe our proposed combination of the two approaches that applies regression test selection after having performed test-suite reductionon an earlier revision.

3.1 Evolving Reduced Test Suite

Test-suite reduction constructs a reduced test suite on a *single* revision, so evaluating the effectiveness of a reduced test suite as software evolves requires evolving the reduced test suite into each subsequent revision. Ideally, we would evaluate test-suite reduction by applying it on some software revision and then consulting with developers on how the reduced test suite would actually evolve into the future. However, performing such an experiment with real developers is rather hard, so we instead simulate, from the actual project history, how the reduced test suite could have evolved had test-suite reduction been applied at some revision.

In general, we need a procedure that takes a reduced test suite from one revision and outputs the "evolved" test suite at a later revision. In our previous work [27], we considered the evolution of the reduced test suite into a future revision by (1) tracking tests based on name and (2) only considering the tests that existed across all revisions, which ignores the tests added (or removed) after performing reduction. This procedure captures how the tests from the reduced test suite behave in the future, but it ignores new (and removed) tests.

We modify our previous procedure to also consider the new (and removed) tests. New tests are often introduced to cover new functionality; if developers were to use test-suite reduction, we assume they would augment the reduced test suite with these new tests as their software evolves with new functionality. Removed tests, on the other hand, may cover functionality that has been removed, or may even be redundant tests manually identified, so we assume developers would remove these tests from the reduced test suite as well (if they still exist in the reduced test suite). In other words, the reduced test suite, used to replace the full test suite, should go through similar changes as the full test suite went through while the software evolved. We denote by $\mathcal{R}_i$ the reduced test suite constructed at revision i from the full test suite $\mathcal{O}_i$. To evolve $\mathcal{R}_i$ to a later revision i', we consider any new and removed tests between $\mathcal{O}_i$ and $\mathcal{O}_{i'}$, where the tests are identified by name. (A more precise approach could track tests not only by name but by the semantics, inferring likely renames [11].)

Definition 1. *The reduced test suite computed at revision i and evolved to subsequent revision i' is the evolved reduced test suite:* $\mathcal{E}_{i,i'} = (\mathcal{R}_i \cap \mathcal{O}_{i'}) \cup (\mathcal{O}_{i'} \setminus \mathcal{O}_i)$

3.2 Change-Related Requirements (CRR)

Our goal is to compare the quality of the subsets of tests chosen-to-run by test-suite reduction and regression test selection, which requires an appropriate metric. Because tests are run to detect faults, the ideal measure would be by the faults that the tests detect. In regression testing, developers focus on code changes and care most about *change-related faults* (either newly introduced faults or latent faults that are exposed by tests only after the change). The tests chosen-to-run by test-suite reduction and regression test selection should ideally detect all the change-related faults that the full test suite detects.

Ideally, we would like to measure only change-related faults and not all faults that a test suite detects. Given a set of tests $\mathcal{T} \subseteq \mathcal{O}_{i'}$ from revision i' that are chosen-to-run after a change Δ, let $faults(\mathcal{T})$ represent the set of faults detected by $\mathcal{T}$. Intuitively, change-related faults are the faults from $faults(\mathcal{T})$ that are not detected before the change Δ is applied. While the intuition can be clear, it is challenging to give an automatic procedure to precisely determine change-related faults, because it is challenging to (1) map each test failure to fault(s), and (2) map faults across software revisions (e.g., consider a case where a function with a fault is inlined in several callers, and a test fails in one of those callers). For this reason, we approximate the set of change-related faults with the set of faults detected only by the tests affected by the change Δ, as found using a safe regression test selection technique, and not detected by tests not affected, i.e., the set of change-related faults is $faults(\mathcal{S}_{i',\Delta} \cap \mathcal{T}) \setminus faults(\mathcal{T} \setminus \mathcal{S}_{i',\Delta})$, where $\mathcal{S}_{i',\Delta}$ is the set of tests selected from $\mathcal{O}_{i'}$. This formula can overapproximate change-related faults when it includes faults that are only detected by the selected tests but unrelated to Δ.

It is also challenging to build a dataset of real test suites that can detect real regression faults, because we need not only one test written to detect each regression fault after it had been already fixed [8, 20] but an entire test suite that could have detected the fault. For this reason, we measure, for each test, what test requirements—statements covered and killed mutants—it satisfies instead of measuring what real faults it detects.

We define a new criterion, which we call *Change-Related Requirements* (CRR), that specifies the change-related requirements that a test suite satisfies:

Definition 2. *The set of change-related requirements satisfied by $\mathcal{T}$ is:* $\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{T}) = req(\mathcal{S}_{i',\Delta} \cap \mathcal{T}) \setminus req(\mathcal{T} \setminus \mathcal{S}_{i',\Delta})$

Here, $req(\mathcal{T})$ is used in the broadest sense to denote any requirements, e.g., a set of statements covered or a set of (seeded or real) faults detected.

	R_1	R_2	R_3	R_4	R_5
t_1			✓	✓	
t_2				✓	✓
$[t_3]$	✓	✓			
$[t_4]$		✓	✓		✓

Figure 1: Matrix for requirements (R) satisfied by tests (t). Red color marks change-related requirements and tests selected by regression test selection. Brackets mark tests in the evolved reduced test suite.

As a simple example to demonstrate CRR, consider the matrix in Figure 1 that shows what requirements each test satisfies. In this example, only the requirement R_3 is change-related, $\mathcal{S}_{i',\Delta} = \{t_1, t_4\}$ and $\mathcal{E}_{i,i'} = \{t_3, t_4\}$. For test-suite reduction, the loss in quality of a reduced test suite is typically measured by $100 \times |req(\mathcal{O}) \setminus req(\mathcal{R})|/|req(\mathcal{O})|$, where $req(\mathcal{T})$ denotes all requirements that a test suite satisfies, including those that are not change-related. In this example, if we use all requirements that all four tests satisfy, both $\mathcal{S}_{i',\Delta}$ and $\mathcal{E}_{i,i'}$ would need to satisfy all five requirements, and we see that both $\mathcal{S}_{i',\Delta}$ and $\mathcal{E}_{i,i'}$ fail to satisfy some requirements. However, R_3 is the only change-related requirement, and CRR should consider only such requirements.

If regression test selection is safe, then the requirements satisfied by the tests chosen-to-run by regression test selection, $\mathcal{S}_{i',\Delta}$, satisfy all change-related requirements (and potentially some more requirements). We see in our example that $\mathcal{S}_{i',\Delta}$ does indeed satisfy R_3. We want to measure how many change-related requirements a reduced test suite satisfies. If we required it to satisfy all requirements satisfied by $\mathcal{S}_{i',\Delta}$, those would include R_2, R_4 and R_5 as well, which are not change-related, hence including these extra requirements could lead to inaccurate results. Indeed, $\mathcal{E}_{i,i'}$ does not satisfy R_4 despite satisfying the change-related requirement R_3, so using $req(\mathcal{S}_{i',\Delta})$ would inaccurately report a quality loss for $\mathcal{E}_{i,i'}$. However, using our proposed criterion, CRR, in this example, filters out the requirements not change-related by removing the requirements satisfied by the non-selected tests t_2 and t_3. We see that $\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{O}_{i'})$ does not include R_2, R_4, and R_5, but only includes R_3. Therefore, in this example, a set of tests only needs to satisfy R_3 to have no quality loss, which is the exact change-related requirement.

3.3 Quality of Reduced Test Suites

We use $\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{E}_{i,i'})$ as the set of requirements that an evolved reduced test suite $\mathcal{E}_{i,i'}$ (initially reduced at revision i and evolved to revision i') satisfies with respect to the changes Δ. We apply the traditional metric for measuring loss in quality by substituting CRR in lieu of requirements:

DEFINITION 3. *Given the full test suite $\mathcal{O}_{i'}$, the selected tests $\mathcal{S}_{i',\Delta}$, and the evolved reduced test suite $\mathcal{E}_{i,i'}$, the change-related loss in quality of $\mathcal{E}_{i,i'}$ is:*

$$\mathsf{CRRLoss}_{i,i',\Delta} = 100 \times \frac{|\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{O}_{i'}) \setminus \mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{E}_{i,i'})|}{|\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{O}_{i'})|}$$

In the example from Figure 1, we determine the change-related requirements for the full test suite to be just R_3, i.e., $\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{O}_{i'}) = \{R_3\}$. Because $\mathsf{CRR}_{\mathcal{S}_{i',\Delta}}(\mathcal{E}_{i,i'}) = \{R_3, R_5\}$ has all the change-related requirements, $\mathsf{CRRLoss}_{i,i',\Delta}$ is 0.

3.4 Selection of Reduction

We compare the effectiveness of test-suite reduction and regression test selection with each other because they both run a subset of tests from the full test suite. However, the two approaches are orthogonal: test-suite reduction removes redundant tests from a single revision, while regression test selection considers changes between two revisions to select tests to run. The two approaches can be combined. After test-suite reduction constructs $\mathcal{R}_i$ for revision i, $\mathcal{R}_i$ can be used as a replacement for $\mathcal{O}_i$, and its evolved form $\mathcal{E}_{i,i'}$ can be used for all future revisions i' in lieu of $\mathcal{O}_{i'}$. Regression test selection can then select tests, affected by the changes between revision i' and its predecessor, from $\mathcal{E}_{i,i'}$ instead of from $\mathcal{O}_{i'}$. Basically, to get selection of reduction, regression test selection would select tests from a reduced test suite. For a revision i', given $\mathcal{S}_{i',\Delta}$ and $\mathcal{E}_{i,i'}$, the tests selected from the evolved reduced test suite are $\mathcal{S}_{i',\Delta} \cap \mathcal{E}_{i,i'}$.

4. EVALUATION

We next describe our evaluation of test-suite reduction, regression test selection, and their combination on 17 open-source projects from GitHub [2]. We first summarize the projects used in our study and then describe how we set up our experiments to answer the following three research questions:

RQ1: Does test-suite reduction or regression test selection yield a smaller number of tests to run on average?

RQ2: What is the change-related loss in quality of the reduced test suite, based on our CRR criterion?

RQ3: How does selection of reduction compare with the other two approaches in terms of number of tests to run?

4.1 Projects

Figure 2 lists the 17 projects that we use in our evaluation. We cloned from GitHub Java projects that use Maven [3] to build and JUnit [19] to run tests. We use 15 projects from our prior study of test-suite reduction [27]; we do not use three projects from the prior study: one does not build any more, one has issues with running tests on many commits, and one does not work with the regression test selection tool Ekstazi [15]. We also include two new projects: Square Retrofit and Apache PDFBox. For each project, we tabulate the number of commits used in our evaluation; the starting SHA (the commit ID in Git); the minimum, median, and maximum number of lines of code among the commits used in the evaluation (measured using SLOCCount [5]); the minimum, median, and maximum number of tests in the full test suite among the commits used in the evaluation; and the time it takes to run the test suite (using `mvn test`) on the latest revision we evaluated on. We conducted the experiments on a 2.33GHz Intel Core2 Quad machine with 16GB of RAM, running Ubuntu 14.04.

4.2 Experimental Setup

For each project, we perform regression test selection for the range of commits specified in Figure 2. We use the state-of-the-art regression test selection tool Ekstazi [1, 14] to find selected tests for the changes between each commit and its predecessor. Ekstazi selects the tests based on *file-level dependencies*, i.e., if a file changed between two commits, Ekstazi selects every test that depends on that file. Moreover, Ekstazi selects at the level of test classes, i.e., if any one test

ID	Project	Commits	Start SHA	LOC			Tests			
				Min	Median	Max	Min	Median	Max	Time
P1	Commons-Lang	509	9ad1a4df	65621	67798	69952	2312	2491	2609	42.546s
P2	Caelum Stella	309	39e50b7f	21374	29418	38153	659	737	801	19.582s
P3	Caelum V|raptor	165	443cf0ed	33972	33972	34743	1052	1113	1147	21.554s
P4	Cloudfoundry UAA	209	76cb4b8c	18059	22794	39051	312	387	858	30.927s
P5	Dropwizard	270	6bf66144	20395	20690	20690	274	313	429	30.149s
P6	Scribe-Java	214	0222f08f	5498	5652	5652	51	79	94	4.182s
P7	SQL-Parser	251	a1ddf59b	46655	46910	46923	49	200	431	9.812s
P8	JodaTime	256	07002501	91448	91482	91482	3866	4000	4136	13.106s
P9	AssertJ-Core	348	df1adedd	55467	64091	76468	4534	5281	6139	23.259s
P10	MessagePack	340	39c7fa7f	6129	6129	6131	8	1499	1532	4.009s
P11	JOPT-Simple	153	e4c251d6	9078	9655	9658	562	691	786	8.255s
P12	SLF4J	335	cfd6bdba	14146	14146	14146	63	114	141	11.864s
P13	Jasmine	180	8d9121ab	11523	11523	11523	34	126	147	34.401s
P14	Square Wire	181	3cad6d0c	14366	14549	14549	23	62	106	11.188s
P15	LA4J	437	7bd10910	7807	8911	8911	245	611	1646	6.741s
P16	Square Retrofit	335	4f3798e6	4641	8096	9923	101	183	250	30.206s
P17	Apache PDFBox	301	7ffff962	101922	110937	115718	626	647	1014	112.464s

Figure 2: Statistics of projects used in our experiments

from a test class depends on a changed file, Ekstazi selects the entire test class with all of its tests.

To measure test quality, we use covered statements and killed mutants as the requirements that the tests must satisfy. We use the mutation testing tool PIT [4] to map each test from the full test suite to the statements covered and killed mutants. Due to the high cost of mutation testing, we do not map this for every commit but only every 30 commits, starting from the starting commit specified in Figure 2. At every commit where we record the satisfied requirements, we also perform test-suite reduction to generate a reduced test suite at that commit. We constructed the reduced test suite by applying the Greedy algorithm [10] on the statements covered. From each reduced test suite, we then create an evolved reduced test suite in each subsequent commit; to perform this evolution, we use the procedure described in Section 3.1.

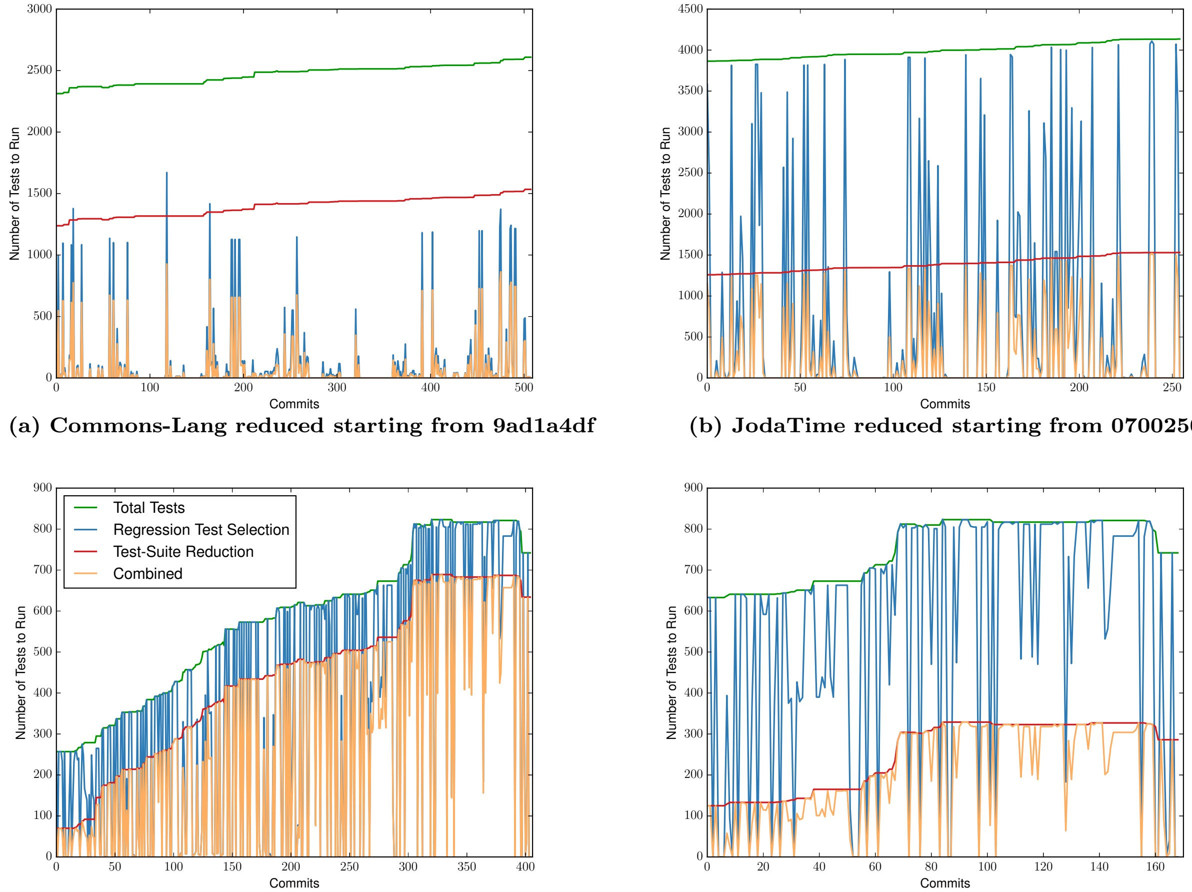
4.3 RQ1: Comparing Number of Tests to Run

To answer RQ1 (and RQ3), we measure how many tests would be chosen-to-run by test-suite reduction and regression test selection (and their combination) across the range of commits for each project. To visualize how the number of tests changes from commit to commit, we show the number as a line plot. Figure 3 shows these plots for three projects: Commons-Lang, JodaTime, and LA4J. In each plot, the lines show the number of tests chosen-to-run by the different approaches at each commit. The green ● line (at the top) represents the total number of tests for each commit, the red ● line (in the middle) represents the number of tests in the evolved reduced test suite (where test-suite reduction is applied at the starting SHA specified in the caption), the blue ● line (with bigger "zig-zags") represents the number of tests selected by regression test selection, and the orange ● line (with smaller "zig-zags") represents the number of tests selected by regression test selection from the evolved reduced test suite.

We use these three projects because they show interesting varying behaviors. For Commons-Lang, regression test selection is relatively small, and for most commits, the number of tests selected by regression test selection is smaller than the number of tests in the evolved reduced test suite. However, for JodaTime, we see many commits throughout the history where regression test selection would have selected many more tests than the evolved reduced test suite. Commons-Lang and JodaTime are both rather mature, and the growth of the test suite seems relatively small and stable, i.e., the number of tests in the full test suite does not change substantially across the commits.

In contrast, for LA4J, the test suite changes greatly as the software evolves. Because of that, we show two plots for LA4J, which also highlight how we apply test-suite reduction at various points in our evaluation. In Figure 3c, we apply test-suite reduction on the starting SHA and evolve the reduced test suite across the entire range of commits used in the evaluation. In Figure 3d, we apply test-suite reduction on a later SHA and evolve it until the end of the range. In both cases we see that regression test selection frequently selects more tests than the evolved reduced test suite. We also see that reapplying test-suite reduction on a later commit reduces the number of tests chosen-to-run by test-suite reduction. For example, from the starting point shown in Figure 3d (the 241st commit from the starting SHA we evaluate on), the evolved reduced test suite in Figure 3c had grown to 486 tests, whereas rerunning test-suite reduction at that point creates a smaller reduced test suite that now has 115 tests.

We do not show detailed plots for the other projects and starting points, because they are similar in shape to the four plots shown and would take a lot of space; we instead summarize their results. For each project and each starting point, we collect for each commit the ratio of the number of tests chosen-to-run by each approach over the number of tests in the full test suite in that commit. Figure 4a shows the distribution of these ratios as violin plots. For each project, we show three violin plots, one for each approach: the red ● (leftmost) for the evolved reduced test suite, the blue ● (middle) for the tests selected by regression test selection, and the orange ● (rightmost) for the tests selected by regression test selection on the evolved reduced test suite. Each violin plot shows the minimum, median (horizontal line), mean (black dot), and maximum values,



(a) Commons-Lang reduced starting from 9ad1a4df
(b) JodaTime reduced starting from 07002501
(c) LA4J reduced starting from 7bd10910
(d) LA4J reduced starting from c8e61571

Figure 3: Tests chosen-to-run across multiple commits for select projects

and the width of the plot depends on the number of points for each value.

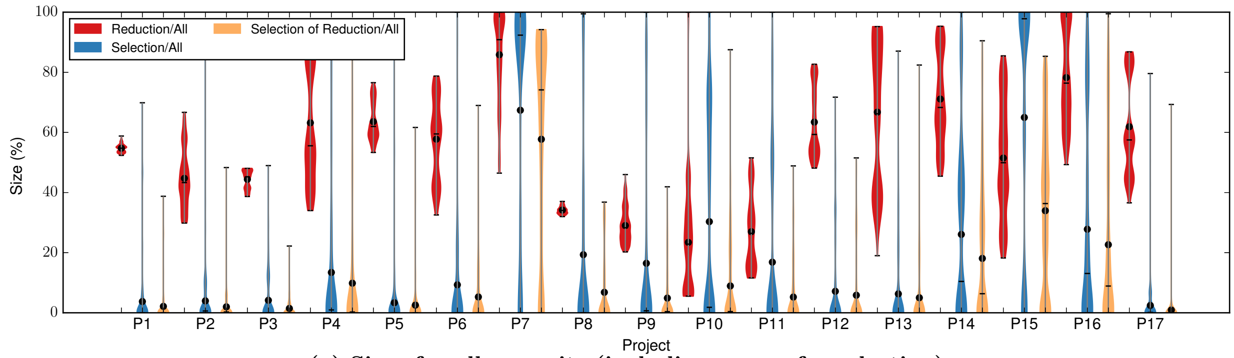
We find that, across the range of commits we evaluated for each project, *regression test selection would have, on average, run fewer tests than test-suite reduction*. (As expected, the number of tests selected by combining the two approaches is even smaller than the number of tests chosen-to-run by either approach.) In particular, the median ratio of tests for evolved reduced test suite is higher than the median ratio of tests for regression test selection for all projects but three (LA4J, MessagePack, and SQL-Parser). In other words, the difference in the median ratio of tests for evolved reduced test suite and the median ratio of tests for regression test selection is positive for all projects but three. The mean of these differences shows that the evolved reduced test suite size is 40.15pp[1] higher than the number of tests selected by regression test selection. However, we note that the number of tests selected by regression test selection across the commits has a much wider distribution than test-suite reduction, i.e., one cannot easily predict the number of tests when using regression test selection.

We also note that regression test selection relatively often selects no tests (because no relevant change was made between commits that would affect any of the tests). The average values for regression test selection could be skewed by these zero cases, and in theory, a developer may realize that no relevant change is made and could manually choose to not run any tests even from a reduced test suite. To explore this further, we plot in Figure 4b the number of tests selected by regression test selection only for the commits when a non-zero number of tests is selected. We find that there are now four projects (JOPT-Simple in addition to LA4J, MessagePack, and SQL-Parser) where regression test selection on average selects more tests than the evolved reduced test suite. Also, the median evolved reduced test suite size is now on average only 22.61pp higher than the median number of tests selected by regression test selection across all projects.

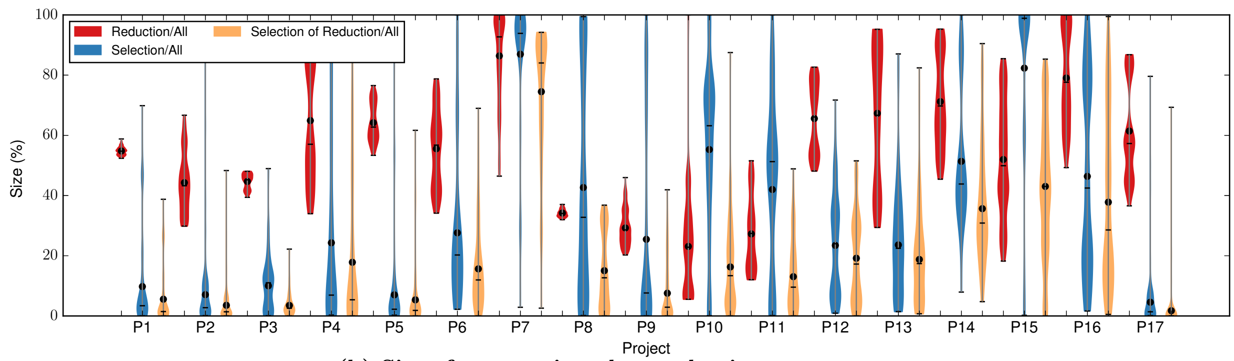
Finally, utilizing the Wilcoxon signed rank test to compare the distributions of the numbers of tests chosen-to-run by test-suite reduction and regression test selection yields p-values significant at an α-level of 0.001, indicating that there is a high probability that the two distributions differ.

RQ1: *In sum, for both scenarios (with and without zero cases), regression test selection selects to run, on average, fewer tests than the evolved reduced test suite.*

[1]The "pp" ("percentage point") unit shows the difference between two percentages by subtracting one from the other.



(a) Sizes for all commits (including zeroes for selection)



(b) Sizes for commits where selection was not zero

Figure 4: Violin plots showing distribution of sizes for reduction, selection, and selection of reduction

This is a surprising result, because regression test selection aims to be safe and selects all tests that could be affected by the changes (and potentially selects many more tests given the coarse, imprecise level of files at which Ekstazi operates), whereas test-suite reduction is unsafe and could miss some test that is affected by the changes.

4.4 RQ2: Comparing Quality of Tests

To evaluate how much test-suite reduction could miss, and to answer RQ2, we measure the quality of the chosen-to-run tests using two kinds of requirements—statement coverage and killed mutants—to compute CRR (Definition 2) and the quality loss (Definition 3). We evaluate the quality loss at each point where we collected the mapping from the tests to the satisfied requirements, by default every 30 commits. For cases where regression test selection did not select any tests (e.g., between commits 29 and 30), we went back one commit at a time (to 28, 27, and so on) and added in all tests selected by regression test selection due to the changes in earlier commits until the selected tests satisfy non-zero requirements. Going back simulates having had a bigger change (i.e., in the Git terminology, as if the developers squashed several commits), because the tests selected by regression test selection reflect the changes between a wider range of commits.

Before we summarize the results for quality loss across all of the projects and various commit points, we show more detailed results for one project. Figure 5 shows, for LA4J, how the change-related statement coverage and killed mutants of evolved reduced test suites compare to the corresponding metrics for the test suites selected by regression test selection, as measured using CRR. Each column shows, for the specified commit (every 30 commits from the beginning), the percentage of change-related statements covered (Figure 5a) or killed mutants (Figure 5b) that are missed by the evolved reduced test suite for that commit. Each row represents the evaluation of an evolved reduced test suite starting from a different commit: the first row is for the evolved reduced test suite computed at the starting SHA, and each subsequent row starts from a later commit (typically 30 commits later). We note that for LA4J, for the most part, there is a very small loss in change-related killed mutants of the evolved reduced test suite, although there are several points where the loss in killed mutants is higher, going up to 9.68%.

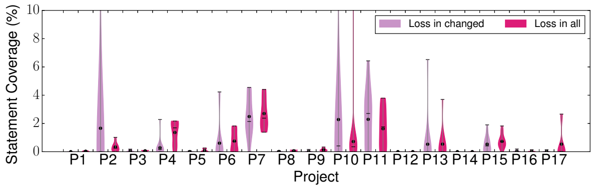
	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
V_0	1.90	1.15	1.11	0.00	1.73	0.00	1.44	1.45
V_1		0.00	0.00	0.00	0.49	0.00	0.41	0.41
V_2			0.37	0.00	0.82	0.00	0.72	0.79
V_3				0.00	0.49	0.00	0.41	0.41
V_4					0.91	0.00	0.76	0.76
V_5						0.00	0.21	0.41
V_6							0.89	0.90
V_7								0.21

(a) Loss in change-related statement coverage (%)

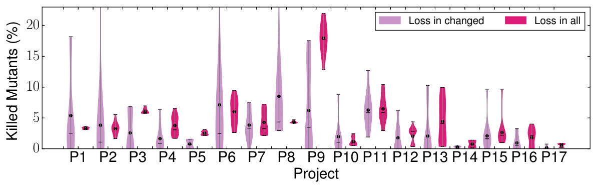
	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
V_0	3.43	2.53	2.43	0.00	2.77	0.00	2.25	2.25
V_1		1.00	1.02	0.00	1.41	0.00	1.18	1.17
V_2			1.81	0.00	2.11	0.00	1.64	1.61
V_3				0.00	1.80	0.00	1.38	1.37
V_4					3.21	0.00	2.21	2.21
V_5						0.00	2.73	2.97
V_6							9.28	9.48
V_7								9.68

(b) Loss in change-related killed mutants (%)

Figure 5: Change-related loss in quality of evolved reduced test suite, with reduced test suite constructed at V_i (i is row) and evaluated at V_j (j is column) for LA4J; distance between row/column is about 30 commits.



(a) Statement coverage



(b) Killed mutants

Figure 6: Violin plots showing distribution of loss in quality due to reduction for different projects

We do not show detailed tables for the other projects and starting points, but we instead summarize their results. Figures 6a and 6b show, for each project, a lavender ● (leftmost) violin plot representing the distribution of the loss in CRR, of both statement coverage and killed mutants, respectively. To contrast CRR, we also show for each project a magenta ● (rightmost) violin plot for the traditional metric of loss for the evolved reduced test suite, where the loss is measured with respect to *all* requirements satisfied by the full test suite: $100 \times |req(\mathcal{O}) \setminus req(\mathcal{R})|/|req(\mathcal{O})|$.

We see that the change-related loss in statement coverage typically has low absolute values, with the median for a project being at most 2.74% (for JOPT-Simple). The values for the traditional metric are also fairly similar, but recall that change-related loss is the metric that matters for evolving code. In contrast, the loss in killed mutants has higher absolute values across all projects, with the median value of loss being as high as 5.93% (for JOPT-Simple) These higher values mean that, after applying test-suite reduction, the resulting evolved reduced test suite at each commit could miss detecting many change-related faults. This is the peril of using test-suite reduction. Compared to the full test suite, test-suite reduction makes the test-suite smaller but can miss some faults that the full test suite can find. This was well understood from previous studies [24,25,28,30,31,33,37] that considered all faults, but we focus on change-related faults and compare with regression test selection.

> ***RQ2:*** *Test-suite reduction can miss on median up to 2.74% change-related statements and 5.93% change-related mutants that regression test selection finds, while (safe) regression test selection cannot miss any change-related fault that the full test suite can find.*

4.5 RQ3: Evaluating Selection of Reduction

Despite the peril of test-suite reduction, test engineers may still use it to speed up regression testing; one can then apply regression test selection on the reduced test suite to speed up regression testing even more. To answer RQ3, we evaluate how effective the combined selection of reduction is compared to individual test-suite reduction and regression test selection. As a comparison metric, we use the number of tests that selection of reduction selects to run at each commit for a project. Note that we do not need to empirically evaluate the loss in change-related fault-detection capability of selection of reduction: because selection of reduction selects from the reduced test suite all the tests that are affected by the changes, assuming a safe regression test selection, the change-related loss of selection of reduction is the same as the change-related loss of test-suite reduction.

Figure 4 shows the violin plots for the number of tests selected by selection of reduction, making it easy to compare the numbers for the combined approach to the two individual approaches. The distribution across all projects shows that, as expected, selection of reduction always selects fewer tests to run than any of the other two approaches. However, as selection of reduction performs selection on top of the reduced test suite, it is not known a priori how the ratio of tests selected by regression test selection from the full test suite compares to the ratio of tests selected by selection of reduction from the reduced test suite. Figure 7a shows the distributions of these two ratios as violin plots for each project. As regression test selection often selects no tests, Figure 7b plots only the non-zero numbers of tests selected by both regression test selection and selection of reduction.

Utilizing the Wilcoxon signed rank test to compare the ratios of tests selected from the full test suite to the ratio of tests selected from the reduced test suite yields p-values significant at the level of 0.001 for all projects except Cloudfoundry UAA (with a p-value of 0.091) and Square Retrofit (with a p-value of 0.293). With the exception of Square Retrofit, there is a high probability that the distributions of the ratio of tests selected by regression test selection from the full test suite and from the reduced test suite differ.

Note, however, that the mean (taken over all projects) pairwise difference between the ratios of the two distributions is only 0.72pp. Therefore, although the Wilcoxon statistical test establishes a high probability that the distributions of these ratios differ for each project (except Square Retrofit), their medians do not differ greatly.

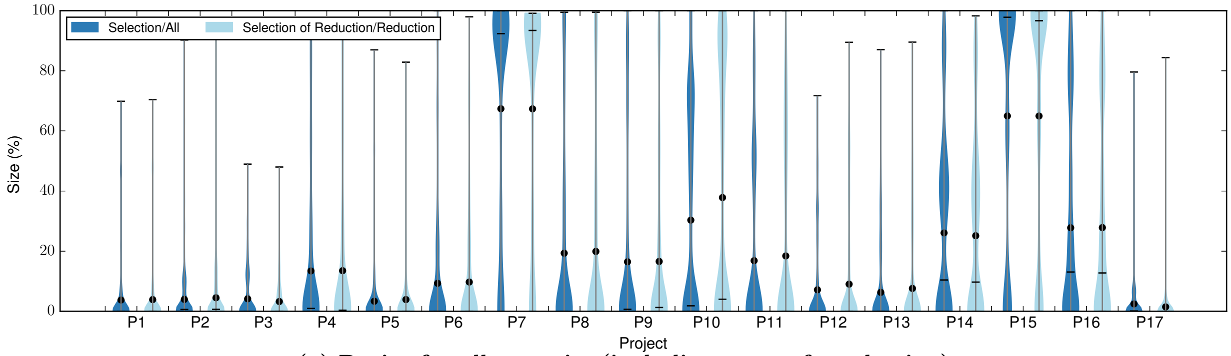
> ***RQ3:*** *Selection of reduction selects fewer tests than either test-suite reduction or regression test selection, and the ratio of tests that selection of reduction selects from the evolved reduced test suite is about the same as the ratio of tests that regression test selection selects from the full test suite.*

4.6 Threats to Validity

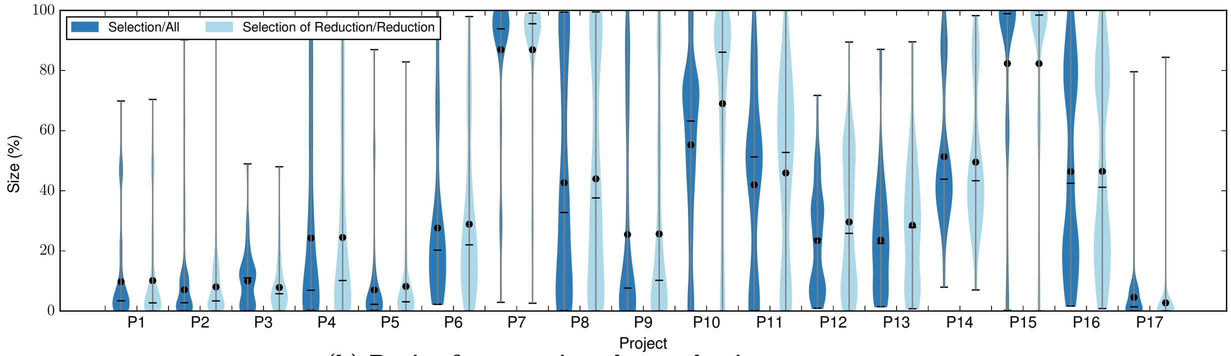
External: Our conclusions may not generalize beyond the projects and revisions we evaluated. To mitigate this issue, we chose actively developed projects from GitHub, varying in size, number of tests, length of history, and application domain. Many of these projects were also previously used to study the effects of software evolution on test-suite reduction [27].

Internal: We automated the process of recording what tests are selected by the Ekstazi tool at each commit in each project used in our evaluation. We also implemented the Greedy algorithm to perform test-suite reduction. We increased the confidence in our code through many small experiments and peer code review.

Construct: We define a new metric, CRR, to measure change-related quality of a reduced test suite. To determine the change-related requirements to be satisfied, we ef-



(a) Ratios for all commits (including zeroes for selection)



(b) Ratios for commits where selection was not zero

Figure 7: Violin plots showing selection/all and selection of reduction/reduction

fectively assume that regression test selection is not only safe but also relatively precise. We need regression test selection to be safe to get all of the change-related requirements, and to be precise to help filter out the requirements not affected by change but accidentally satisfied by the tests selected by regression test selection. If regression test selection selects tests that are not affected by the changes (in the extreme selecting all tests), then we cannot filter out the accidentally satisfied requirements.

The regression test selection tool we use, Ekstazi, tracks dependencies at the level of files (including classes for code) and is fairly safe (with respect to code changes). While Ekstazi is not as precise as tools that track finer-grained dependencies than files [35], Ekstazi is safer than those tools, and safety is more important to properly determine which requirements are change-related. Also, Ekstazi is publicly available [1]. Ekstazi collects dependencies at the level of test classes not test methods: if a single test (method) is affected by a change, its entire test class is selected to be run. In contrast, our test-suite reduction is at the level of individual tests, which allows removing each test based on the requirements it satisfies. Hence, there is a discrepancy in the level of the chosen-to-run tests for the two approaches. Using Ekstazi, our results for regression test selection may select more tests than actually necessary, so our comparison of test-suite sizes for test-suite reduction and regression test selection overestimates the sizes for regression test selection. Therefore, one of our key findings, that the size of selected test suites is, on average, smaller than the size of reduced test suite, could be even stronger.

We evolved a reduced test suite across many commits using a particular procedure (Section 3.1) to simulate how a test engineer who used a reduced test suite in lieu of the full test suite may have evolved the reduced test suite in future commits. This simulation allows a comparison to regression test selection, which directly considers changes, including those that modify the test suite, adding or removing tests. Specifically, our procedure for evolving the reduced test suite considers all changes made to the full test suite and adds all new tests. However, had the developer actually performed test-suite reduction, the reduced test suite could have evolved differently than the way we simulated it. Furthermore, we track tests between revisions by name only, so if a test is renamed, that test would be considered a newly added test. Not handling renames can affect the results if a test that was originally considered redundant (and therefore excluded from the reduced test suite) was renamed in a subsequent revision: that test would be considered a new test and "re-added" to the reduced test suite, possibly introducing the same redundancy back in.

4.7 Discussion

Change-Related Requirements: Automatically evaluating the fault-detection capability of test suites *with respect to software changes* is challenging. We define CRR (Definition 2) by contrasting the requirements satisfied by tests selected by regression test selection to the requirements satisfied by tests not selected. While this definition is not ideal, it is better than some alternatives. For example, we could have considered only the requirements directly in the changed code, e.g., finding changed lines between two revisions and inserting mutants only on those lines. However, that would favor regression test selection over test-suite reduction and miss some requirements that are affected by the changed code but not directly in the changed code. If a fault is in some unchanged part of code, it is possible that a change elsewhere in the code could trigger the detection of this fault. For instance, consider a fault that is control dependent on some branch condition, and a code change that affects the branch condition. If we ignore whether a test suite can find this fault, we could incorrectly label two test suites as equally good even when one finds this fault and the other does not find it. Future work should seek a better research methodology to evaluate quality of test suites with respect to changes.

Reduction of Selection: For research evaluation purposes, one could consider performing test-suite reduction on test suites selected by regression test selection, and this combined approach—called *reduction of selection*—could yield different results than selection of reduction. However, reduction of selection is impractical for speeding up regression testing, because performing (precise) reduction requires first running all the tests on the current revision to construct a requirements matrix used to determine redundant tests. Thus, one could only determine which tests not to run after actually running those tests. An alternative could be to use a stale requirements matrix computed on some previous revision to reduce the selected test suite, but this requirements matrix would be imprecise for the current revision and would become more imprecise as some tests would not be run due to the reduction. Moreover, new requirements could have been introduced due to a change, and reduction on a stale matrix, being unaware of these new requirements, cannot guarantee that the test suite reduced from the selected test suite would satisfy all the requirements, which goes against the purpose of (precise) test-suite reduction. Future work could evaluate such imprecise test-suite reduction and compare it with unsafe regression test selection.

5. RELATED WORK

There is a large body of work on test-suite reduction and regression test selection, but to the best of our knowledge, there was no prior work on comparing and combining those two approaches for code. Korel et al. [21] propose a technique that combines test-suite reduction and regression test selection, which (1) is similar to reduction of selection, but differs from our selection of reduction, and (2) operates on models, specifically, extended finite state machines (EFSMs), unlike our combination that operates on real code and tests. Given two EFSMs, their technique first computes a set of elementary changes, then uses a static dependence analysis to find which test covers what changes, and finally selects to run only tests that are non-redundant with respect to the changes. In contrast, our evaluation does not assume any static analysis and measures the change-related loss in quality for the reduced test suite.

There are also studies of the effects of software evolution on test information. Elbaum et al. [12] studied the effects on test coverage and found that coverage information changed greatly even for small software changes. We recently studied the effects of software evolution on test-suite reduction [27]. We conducted experiments on 18 open-source projects and measured the quality of the constructed reduced test suite across multiple commits. We found that the quality of the reduced test suite does not drop much as software evolves, comparing the reduced test suite to the full test suite using requirements of the entire software. The current study focuses on measuring the test quality using change-related requirements, and we find that the reduced test suite can suffer some greater loss in quality.

There is prior work on combining various regression testing approaches. For example, Zhang et al. [36] combined test-suite prioritization and test-suite reduction for mutation testing. We are the first to combine test-suite reduction and regression test selection.

There is also other work on speeding up regression testing, e.g., using history of test runs to perform unsafe test selection on large industrial codebases. Most recently, Herzig et al. [18] proposed a technique that selects tests based on historical data about test results. Their technique balances the cost of running a test many times in certain development branches versus running the test just once before release at the risk of it revealing a fault that requires much more time to debug. Their simulation of the technique on Microsoft products found a reduction of 50% of test executions. Elbaum et al. [13] combine unsafe test selection in the pre-commit stage with test prioritization in the post-commit stage, and evaluate their technique at Google. Yoo et al. [34] propose a related technique using multi-objective sampling [32] based on coverage, cost, and fault history, and also evaluate their technique at Google. We evaluate our approaches on small open-source projects but expect that some of the key findings would carry over to large codebases.

6. CONCLUSIONS

This paper is the first to empirically compare and combine test-suite reduction and regression test selection, two approaches that can speed up regression testing but were evaluated only separately. We propose a new criterion, Change-Related Requirements, to evaluate the quality of running a set of tests with respect to the changes made in software. We also use test-suite size to evaluate chosen-to-run tests. Our results on 17 open-source projects show three interesting conclusions. First, regression test selection on average selects *fewer* tests than test-suite reduction. Second, test-suite reduction can lose change-related fault-detection capability (of up to 5.93% for killed mutants), while (safe) regression test selection has no loss. Third, our proposed selection of reduction approach provides the greatest speed-up, though with the same loss in change-related fault-detection capability as the reduced test suite, and has about the same selection ratio from the reduced test suite as regression test selection has from the full test suite.

In summary, our results show that if only one approach must be chosen, either test-suite reduction or regression test selection, to speed up regression testing, the test engineer should choose regression test selection, as it selects fewer tests and preserves change-related fault-detection capability, where the faults are most likely to appear. If there is a need to speed up testing even further, then combining both test-suite reduction and regression test selection is a worthwhile approach that provides even greater savings in the number of tests as long as one is willing to tolerate the possible loss in fault-detection capability for some changes in software.

7. ACKNOWLEDGMENTS

We would like to thank Milos Gligoric for his help with the Ekstazi tool and the anonymous reviewers for feedback on a previous draft of this paper. This research was partially supported by the NSF Grant Nos. CCF-1012759, CCF-1421503, CCF-1434590, and CCF-1439957. Alex Gyori was partially supported by the Saburo Muroga Endowed Fellowship.

8. REFERENCES

[1] Ekstazi. http://www.ekstazi.org/.
[2] GitHub. https://github.com/.
[3] Maven. http://maven.apache.org/.
[4] PIT mutation testing. http://pitest.org/.
[5] SLOCCount. http://www.dwheeler.com/sloccount/.
[6] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? A characterization of open source software repositories. In *ICPC*, 2008.
[7] 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*, 2009.
[8] M. Böhme and A. Roychoudhury. CoREBench: Studying complexity of regression errors. In *ISSTA*, 2014.
[9] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *ICSE*, 2014.
[10] T. Y. Chen and M. F. Lau. A simulation study on some heuristics for test suite reduction. *IST*, 40(13), 1998.
[11] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.
[12] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *ICSM*, 2001.
[13] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, 2014.
[14] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE*, 2015.
[15] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, 2015.
[16] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov. Regression test selection for distributed software histories. In *CAV*, 2014.
[17] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *ICSE*, 2012.
[18] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *ICSE*, 2015.
[19] JUnit home page. https://github.com/kentbeck/junit/wiki.
[20] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*, 2014.
[21] B. Korel, L. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *ICSM*, 2002.
[22] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *ICTCS*, 1995.
[23] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2), 1997.
[24] 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*, 1998.
[25] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *STVR*, 12(4), 2002.
[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, 1999.
[27] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, 2014.
[28] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *PASTE*, 2005.
[29] Testing at the speed and scale of Google, Jun 2011. http://goo.gl/2B5cyl.
[30] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE*, 1995.
[31] 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*, 1997.
[32] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *ISSTA*, 2007.
[33] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2), 2012.
[34] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In *ESEC/FSE*, 2011.
[35] L. Zhang, M. Kim, and S. Khurshid. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In *SIGSOFT FSE*, 2012.
[36] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*, 2013.
[37] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of JUnit test-suite reduction. In *ISSRE*, 2011.