2011 22nd IEEE International Symposium on Software Reliability Engineering

An Empirical Study of JUnit Test-Suite Reduction

Lingming Zhang*, Darko Marinov†, Lu Zhang‡, Sarfraz Khurshid*
*Electrical and Computer Engineering, University of Texas at Austin
Email: zhanglm@utexas.edu, khurshid@ece.utexas.edu
†Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
Email: marinov@illinois.edu
‡Institute of Software, Peking University, Beijing, 100871, P. R. China
Email: zhanglu@sei.pku.edu.cn

***Abstract*—As test suites grow larger during software evolution, regression testing becomes expensive. To reduce the cost of regression testing, test-suite reduction aims to select a minimal subset of the original test suite that can still satisfy all the test requirements. While traditional test-suite reduction techniques were intensively studied on C programs with specially generated test suites, there are limited studies for test-suite reduction on programs with real-world test suites. In this paper, we investigate test-suite reduction techniques on Java programs with real-world JUnit test suites. We implemented four representative test-suite reduction techniques for JUnit test suites. We performed an empirical study on 19 versions of four real-world Java programs, ranging from 1.89 KLoC to 80.44 KLoC. Our study investigates both the benefits and the costs of test-suite reduction. The results show that the four traditional test-suite reduction techniques can effectively reduce these JUnit test suites without substantially reducing their fault-detection capability. Based on the results, we provide a guideline for achieving cost-effective JUnit test-suite reduction.**

I. Introduction

Large-scale software usually undergoes evolution to refactor existing code, fix bugs, or add new features. To validate software changes during software evolution, developers often use regression test suites. However, the accumulated regression test suites can become extremely large and time-consuming to run. For example, an industrial collaborator of Rothermel et al. [1] reported that running the entire test suite for one of their software products takes nearly seven weeks. Therefore, researchers have developed various techniques to reduce the cost of regression testing through test-suite reduction [2]–[4], test-case prioritization [1], [5], [6], and regression test selection [7]. Yoo and Harman [8] presented a detailed survey of regression testing techniques.

Test-suite reduction [2], [3], [9], [10] (also known as *test-suite minimization* [4]) aims to find a representative subset of the original test suite which can satisfy the same *test requirements* as the original test suite. For example, when testers use statement coverage as test requirements, test-suite reduction becomes the problem of finding a subset of the original test suite which covers the same statements as the original test suite. Formally, given an original test suite T, and a set of test requirements R, the problem of test-suite reduction is defined as finding a set of test cases $T_{reduced} \subseteq T$ such that $\forall r \in R(\exists t \in T, t \text{ satisfies } r \Rightarrow \exists t' \in T_{reduced}, t' \text{ satisfies } r)$.

Finding the minimal representative subset for a given test suite is equivalent to the problem of set covering and has been shown to be NP-complete [11]. Therefore, many algorithms were proposed to generate approximately minimal reduced test suites. Traditional test-suite reduction techniques include greedy techniques [10], heuristic-based techniques [2], [3], and techniques based on integer linear programming (ILP) [12], [13]. The key metrics for evaluating these techniques are (1) the size of the reduced test suites and (2) the fault-detection capability of the reduced test suites (because removing some test cases from the original test suite can lower its fault-detection capability).

Several empirical studies were conducted with the proposed test-suite reduction techniques to compare the sizes of reduced test suites. For example, Chen et al. [10] compared the sizes of reduced test suites via a simulation study, and Zhong et al. [9] studied both the sizes of reduced test suites and the time taken by various test-suite reduction techniques on a set of C programs. There are also empirical studies on the effect of test-suite reduction on lowering the fault-detection capability. However, the findings are non-conclusive: Wong et al. [14], [15] found that test-suite reduction does not substantially lower the fault-detection capability of test suites, whereas Rothermel et al. [16] found that test-suite reduction can severely lower the fault-detection capability.

In this paper we present a new study on test-suite reduction, which greatly extends four aspects compared to the existing studies. (1) The subject programs in our study are larger than those used in previous studies; our programs are in Java not in C as in most existing reearch [2], [4], [9], [15], [17], and coding styles in Java and C can differ. (2) The nature of tests is substantially different: we focus on tests for small units of code from a larger system and hence some tests may have mutually disjoint execution paths, whereas previous studies looked at small programs where many tests execute the same `main` method albeit on different inputs. (3) Test generation is substantially different: our focus is on real test suites that are accumulated during a system's evolution, whereas previous studies mainly focused on synthetic suites that were generated in the laboratory. (4) The cost-benefit analysis in our study is substantially more extensive: some previous studies considered both benefits (reduction of test suite size) and costs (lowering of fault-detection capability) of test-suite reduction [4], [14],

1071-9458/11 $26.00 © 2011 IEEE
DOI 10.1109/ISSRE.2011.26

[15] but did not consider the influence of reduction techniques and test granularities; similarly, there are studies considering the influence of reduction techniques [3], [9], [10], but they only considered benefits and not costs of test-suite reduction. In addition, no previous study evaluated the benefits/costs of different JUnit test *granularities*, where *granulatiry* refers to grouping JUnit test cases based on their test methods or their test classes.

More specifically, we evaluate both the benefits and costs of four existing, traditional test-suite reduction techniques on 19 versions of four real-world Java programs, totaling over 700 KLoC, with accumulated JUnit test suites. We implemented four techniques—one greedy technique [10], two heuristic-based techniques [2], [3], and one ILP-based technique [13]—and applied them on reducing test suites written in JUnit. In addition to the choice of technique, our study involves two more independent variables: test-case granularity and test-coverage level.

The results show that these traditional test-suite reduction techniques can effectively reduce the size of these test suites without substantially lowering the fault-detection capability, which partially confirms Wong et al.'s study [14], [15] but differs from Rothermel et al.'s study [4]. The results also show that, for the Java programs and JUnit test suites considered, the choice of test-reduction technique does not impact much the sizes of reduced test suites or their fault-detection capabilities, while the choice of test-case granularity and the choice of test-coverage level significantly impact both costs and benefits of test-suite reduction. Based on the results, we provide a practical guideline for cost-effective reduction of real-world JUnit test suites.

II. Preliminaries

In this section, we introduce the concepts and notations used throughout this paper. Given a test suite T and a set of test requirements R for the program under test, we denote the *satisfiability* relation between test cases and test requirements as $S : T \times R = \{\langle t, r\rangle | t \in T, r \in R, t \text{ satisfies } r\}$. Then, for any test case $t \in T$, we use $S[t] = \{r | \langle t, r\rangle \in S\}$ to denote the set of test requirements satisfied by t. Furthermore, for any subset of test cases $T' \subseteq T$, we denote the set of test requirements satisfied by any test cases in T' as $S[T'] = \bigcup_{t \in T'}(S[t])$. Similarly, for any test requirement $r \in R$, we use $S^{-1}[r] = \{t | \langle t, r\rangle \in S\}$ to denote the set of test cases satisfying r. For any subset of test requirements $R' \subseteq R$, we use $S^{-1}[R'] = \bigcup_{r \in R'}(S^{-1}[r])$ to denote the set of test cases satisfying any test requirements in R'.

As briefly mentioned in Section I, the definition of test-suite reduction is finding a subset of the original test suite that can still satisfy all the test requirements satisfied by the original suite. Using our notation, we can define test-suite reduction simply as finding a representative subset $T' \subseteq T$ such that $S[T'] = S[T]$. We can further define the minimal representative subset as a test suite $T_m \subseteq T$ such that $S[T_m] = S[T] \wedge (\forall T' \subseteq T,\ S[T'] = S[T] \Rightarrow |T_m| \leq |T'|)$, where $|T|$ denotes the size of test suite T. Although the problem of finding a minimal representative subset has been recognized as NP-complete, researchers have utilized the characteristics of different test cases to develop algorithms that find approximately minimal representative subsets. There are mainly two kinds of test cases that have been shown to be useful in guiding test-suite reduction:

Essential Test Cases. A test case t is called *essential* if and only if t satisfies some test requirements exclusively, i.e., t must appear in every minimal representative subset [3]. In this paper, we use the symbol $E(T)$ to denote the set of all essential test cases in T. Because essential test cases must appear in every minimal representative subset, these test cases should be selected as early as possible. This insight was adopted by various test-suite reduction techniques [2], [3].

1-to-1 Redundant Test Cases. A test case t is called *1-to-1 redundant* if and only if there exists another test case $t' \neq t$ that can satisfy all the requirements satisfied by t, i.e., $S[t] \subseteq S[t']$ [3]. Whenever a 1-to-1 redundant test case t would have appeared in a representative subset, we can use t' to replace t, i.e., any representative subset of $T - \{t\}$ is also representative subset of T. Therefore, 1-to-1 redundant test cases need not appear in any minimal representative subsets and should be eliminated as early as possible. This insight was also adopted by traditional test-suite reduction techniques [3], [18].

III. Studied Techniques

In this section, we briefly introduce the four traditional test-suite reduction techniques investigated in this paper. For the original test suite T and test requirements R, we use m and n, respectively, to denote their sizes, i.e., $m = |T|$ and $n = |R|$. In addition, we denote the maximum number of test cases satisfying one test requirements as k, and the maximum number of test requirements that are satisfied by one test case as l. Formally, $k = \max_{r \in R}(|S^{-1}[r]|)$ and $l = \max_{t \in T}(|S[t]|)$.

A. The Greedy Technique

Given the original test suite T and test requirements R, the greedy technique iteratively selects a test case that satisfies the maximum number of unsatisfied test requirements until all the test requirements satisfied by T have been satisfied. More precisely, the greedy technique initializes the resulting suite $T_{reduced}$ as $\emptyset$. During each iteration, the greedy technique first selects a test case t such that $\forall t' \in (T - T_{reduced}), |S[t']| \leq |S[t]|$, then the technique puts t into $T_{reduced}$ and removes $S[t]$ from the requirement set satisfied by each unselected test cases. Finally, the greedy technique terminates and returns $T_{reduced}$ when $S[T_{reduced}] = S[T]$. Because each iteration selects a test case that satisfies at least one unsatisfied test requirement, the maximum number of iterations is $\min(m, n)$. During each iteration, the time complexity for selecting the test case with maximum number of satisfied requirements is at most $O(m)$, and the time complexity for updating the satisfied requirements of not selected test cases is at most $O(ml)$. Therefore, the total time complexity for the greedy technique is $O(ml\min(m, n))$.

B. Harrold et al.'s Heuristic

This heuristic was first proposed by Harrold et al. [2], and its basic idea is to select more essential test cases earlier. More precisely, this heuristic first groups all the test requirements into R_1, R_2, ..., R_k, where R_i represents the set of test requirements that are satisfied by exactly i test cases, i.e., $R_i = \{r|\ i = |S^{-1}[r]|\}$, and k is the maximum number of test cases that can satisfy one requirement. As R_1 represents all the requirements that can only be satisfied by one test case, all the test cases satisfying requirements in R_1 are essential test cases, i.e., $E(T) = S^{-1}[R_1]$. Therefore, this heuristic first selects all the test cases satisfying R_1, i.e., $E(T)$, and marks all requirements in R_1 as satisfied. The situation for R_i $(i > 1)$ is more complicated: the heuristic continuously selects the test cases that satisfy the maximum number of not-yet-satisfied requirements in R_i; if two or more test cases are tied, the heuristic continues to compare the number of not-yet-satisfied requirements satisfied by them in R_{i+1}, and the procedure continues until $i = k$. If the procedure still fails to select a winner, the heuristic randomly selects from the candidates. When all the requirements in R_i have been satisfied, the heuristic will continue to R_{i+1}, following the same procedure as for R_i. This heuristic terminates when all the requirements in R have been satisfied. The time complexity of this heuristic is $O((m+n)nk)$ [10].

C. The GRE Heuristic

The GRE heuristic was first proposed by Chen and Lau [3], [10]. This heuristic utilizes both the characteristics of essential test cases and 1-to-1 redundant test cases, and brings them together with the greedy strategy. The selection of essential test cases removes newly satisfied requirements from the satisfying requirement set of unselected test cases, thus causing newly 1-to-1 redundant test cases. On the other side, the removal of 1-to-1 redundant test cases might in turn generate more essential test cases. Therefore, the GRE technique alternatively applies these two strategies when applicable. When the process cannot continue with any of these two strategies, the GRE heuristic simply uses the greedy technique to break the deadlock and then resumes the process. The heuristic terminates when all the requirements in R have been satisfied. It has been shown that if the greedy technique is never used during the reduction, the reduced test suite by this heuristic is exactly the minimal representative set of the original suite [3]. The time complexity for this heuristic is $O((n+m^2l)\min(m,n))$ [10].

D. The ILP Technique

Black et al. [13] proposed two integer linear programming (ILP) models for test-suite reduction. The first ILP model is for single-objective test-suite reduction, while the second ILP model is for multiple-objective test-suite reduction. As the second ILP model also takes the different fault detection capabilities of different test cases into account, we only introduce the first ILP model in this paper to enable fair comparison with the other traditional test-suite reduction techniques. The first ILP model aims to minimize the number of selected test cases that satisfy the same test requirements with the original test suite, and is defined as:

$$\begin{array}{ll} \text{Objective}: & \text{Minimize}(\sum_{j=1}^{m} x_j), x_j \in [0,1] \\ \text{Constraint}: & \bigwedge_{i=1}^{n}(\sum_{j=1}^{m} s_{ij}x_j \geq 1), s_{ij} \in [0,1] \end{array}$$

where x_j represents whether the jth test case in T is selected in the reduced suite (i.e., $x_j = 1$ denotes the jth test case is selected), and s_{ij} represents whether the ith requirement in R is satisfied by the jth test case in T (i.e., $s_{ij} = 1$ denotes the ith requirement is satisfied by the jth test case). The model encodes finding the minimized number of test cases in T that still satisfy all the test requirements in R, which is exactly the definition of test-suite reduction. Although ILP is an NP-complete problem [19], recent ILP solvers, e.g., *IBM Symphony library* [20], have shown to be effective in practice.

IV. Empirical Study

A. Research Questions

Our empirical study addresses the following research questions:

- **RQ1:** How much do the traditional test-suite reduction techniques reduce the sizes and the fault-detection capability for real-world JUnit test suites?
- **RQ2:** How do the different experimental factors (e.g., the choices of test-suite reduction technique, test-case granularity, and test-coverage level) impact the size of the reduced JUnit test suites?
- **RQ3:** How do the different experimental factors influence the fault-detection capability of the reduced JUnit test suites?

Note that the first research question is mainly concerned with whether there exists a cost-effective reduction for JUnit test suites, while the second and the third questions are concerned with how to achieve cost-effective JUnit test-suite reduction in practice.

B. Considered Independent Variables

In the design of empirical studies, certain selected factors are usually controlled or changed by experimenters to investigate the relationships between these factors and final experimental results. These factors are called *independent variables* (IV). Based on our research questions and the style of JUnit test suites, we consider three independent variables for JUnit test-suite reduction: the test-suite reduction technique, the test-case granularity, and the test-coverage level.

IV1: Reduction Techniques. In this study, we used four traditional test-suite reduction techniques that have been widely used in previous studies on C programs (e.g., [4], [9], [10]), and applied them to real-world JUnit test suites:

- The Greedy Technique (denoted as G), which is presented in detail in Section III-A.
- Harrold et al.'s Heuristic (denoted as H), which is presented in detail in Section III-B.
- The GRE Heuristic (denoted as GRE), which is presented in detail in Section III-C.

- The ILP Technique (denoted as ILP), which is presented in detail in Section III-D.

Note that all the studied techniques use random selection to break ties when two test cases have the same priorities. To make a fair comparison, we run each technique for 100 different random seeds and use the arithmetic mean values as average results for reduction in test-suite size and fault detection capability for each technique.

IV2: Test-Case Granularity. Previous studies on regression testing for JUnit test suites (e.g., JUnit test-case prioritization [21]) have identified test-case granularity as an important factor in their experimental design. Therefore, we also consider test-case granularity as an independent variable in our empirical study. More precisely, following the previous studies, we consider these two types of test-case granularity:

- Test-Class Granularity (denoted as TC), where we consider each JUnit TestCase class as a test case.
- Test-Method Granularity (denoted as TM), where we consider each JUnit test method within each TestCase class as a test case.

Section IV-D describes in further detail the test cases at different granularity in the studied subjects.

IV3: Test-Coverage Level. Method coverage and statement coverage are two commonly used criteria for code coverage in previous regression-testing studies on JUnit test suites [21], [22]. In this study, we also use these two coverage criteria to define test requirements for JUnit test suites:

- Method Level (denoted as Meth), which specifies covering all the Java source methods as the test requirements for test-suite reduction.
- Statement Level (denoted as Stat), which specifies covering all the Java source statements as the test requirements for test-suite reduction.

Note that although statement and method coverage are not very strong criteria, they are widely used in practice and provide larger reduction in test-suite size than stronger criteria.

C. Dependent Variables and Metrics

In empirical studies, *dependent variables* (DV) are used to indicate and measure the interesting aspects of final results. In this study, we use two dependent variables commonly used by traditional test-suite reduction studies [4], [14], [17]:

DV1: Reduction in Test-Suite Size. This variable (abbreviated as RS) denotes the ratio of test cases reduced from the original test suite. Following the traditional test-suite reduction work [4], [14], we use the following metric for measuring this variable:

$$RS = \frac{|T| - |T_{reduced}|}{|T|} * 100$$

where T denotes the set of test cases in the original test suite, while $T_{reduced}$ denotes the set of test cases in the reduced test suite.

DV2: Reduction in Fault-Detection Capability. This variable (abbreviated as RF) denotes the ratio of lowered fault-detection capability. Following the traditional test-suite reduction work [4], [14], we use the following metric for measuring this variable:

$$RF = \frac{|F| - |F_{reduced}|}{|F|} * 100$$

where F denotes the set of faults revealed by the original test suite, while $F_{reduced}$ denotes the set of faults revealed by the reduced test suite.

Note that the first dependent variable is an indicator of the *benefit* brought by test-suite reduction, while the second dependent variable is an indicator of the *cost* of test-suite reduction.

D. Subject Programs, Test Suites, and Faults

We used 19 versions of four real-world Java programs as subjects for this study: 3 versions of *jtopas*, 3 versions of *xml-security* (abbreviated as *xmlsec*), 5 versions of *jmeter*, and 8 versions of *ant*. *jtopas*[1] is code for parsing text data. *xml-security*[2] implements XML signature and encryption standards. *jmeter*[3] is used for load testing and performance measurement. *ant*[4] is a Java-based build tool, similar to the Unix tool *make*. We obtained the successive versions of these four programs from the Software-artifact Infrastructure Repository (SIR)[5] [23]. We chose SIR because it is very widely used. The sizes of the studied subjects range from 1.89 KLoC to 80.44 KLoC, and amount to total of 701.10 KLoC. Table I shows the detailed statistics of the subjects. For each subject, the first two columns show the mapping between the labels we used and the actual subjects, and the next three columns show the number of statements, the number of classes, and the number of methods, respectively.

Each version of each program comes with a JUnit test suite in SIR. We directly used those JUnit test suites as the original test suites for applying test-suite reduction. Note that due to the specific style of organizing JUnit tests, there are two natural types of test-case granularity: the test-class granularity that treats each TestCase class as a test case and the test-method granularity that treats each test method within each TestCase class as a test case. The numbers of test cases at the test-class granularity and the test-method granularity for each subject are shown in columns 6 and 7 of Table I.

Each version of each program also comes with a set of manually *seeded* faults in SIR, and the number of seeded faults is shown in Column 8 of Table I. We used these seeded faults to form faulty versions to simulate software evolution. Then we evaluate the reduction of test-suite sizes (i.e., RS) and the lowering of fault-detection capability (i.e., RF) caused by test-suite reduction to address the research questions. In addition, previous research has shown that it is often appropriate to use automatically *mutated* faults for regression-testing experimentation [22], [24], [25]. In fact, Andrews et al. [24], [25] found that for software-testing experimentation mutated faults are

[1]http://jtopas.sourceforge.net/jtopas/, Accessed in August 2011.
[2]http://santuario.apache.org/, Accessed in August 2011.
[3]http://jakarta.apache.org/jmeter/, Accessed in August 2011.
[4]http://ant.apache.org/, Accessed in August 2011.
[5]http://sir.unl.edu/portal/index.html, Accessed in August 2011.

TABLE I
STATISTICS FOR SUBJECTS

Label	Subject	Number of Statements	Number of Classes	Number of Methods	Number of Test Classes	Number of Test Methods	Number of Seeded Faults	Number of Mutated Faults
S_1	jtopas-v1	1897	19	285	10	126	10	100
S_2	jtopas-v2	2031	21	304	11	128	12	100
S_3	jtopas-v3	5361	50	748	18	209	16	100
S_4	xmlsec-v1	18323	179	1627	15	92	20	100
S_5	xmlsec-v2	18985	180	1629	15	94	19	100
S_6	xmlsec-v3	16878	145	1398	13	84	13	100
S_7	jmeter-v1	33670	334	2919	26	78	19	35
S_8	jmeter-v2	33097	319	2838	29	80	20	100
S_9	jmeter-v3	37271	373	3445	33	78	19	100
S_{10}	jmeter-v4	38357	380	3536	33	78	13	100
S_{11}	jmeter-v5	41052	389	3613	37	97	12	100
S_{12}	ant-v1	25846	228	2511	34	137	11	100
S_{13}	ant-v2	39733	342	3836	51	219	21	100
S_{14}	ant-v3	38810	342	3845	51	219	7	100
S_{15}	ant-v4	61877	532	5684	102	521	26	100
S_{16}	ant-v5	63510	536	5802	105	557	15	100
S_{17}	ant-v6	63578	536	5808	105	559	1	100
S_{18}	ant-v7	80381	649	7520	149	877	29	100
S_{19}	ant-v8	80444	650	7524	149	878	4	100

even more similar to real faults than seeded faults, as seeded faults seem to be harder to detect than real faults. Therefore, we also used mutated faults produced by mutation testing to form faulty versions for evaluating RF values. More precisely, we used all 15 traditional mutation operators of *MuJava*[6] [26] to produce faulty versions (i.e., mutants) to form a mutant pool for each subject. For each subject, shown in Column 9 of Table I, we randomly select 100 mutants that can be killed by the original test suite as *mutated* faults from its mutant pool to evaluate RF values. The only exception is on subject *jmeter-v1*: we only use 35 mutants as *mutated* faults to evaluate RF values, because only 35 mutants can be killed by the original test suite of *jmeter-v1*.

E. Implementation and Supporting Tools

We implemented the basic coverage-collection component for JUnit test-suite reduction based on the *Eclipse JDT toolkit*[7] and the *ASM byte-code manipulation framework*[8]. Based on the *Eclipse JDT toolkit*, the component automatically distinguishes source classes and JUnit TestCase classes for programs under test to enable different instrumentation. Based on the *ASM framework*, the component instruments each JUnit test method to record the test method identifier and the identifier of its corresponding test class, and instruments each source method and statement to record all the statements or methods executed by each JUnit test method. With the instrumentation, whenever corresponding JUnit test methods are executed, their identifiers and statements or methods executed by them will be automatically traced. With the traced coverage information for each test-method granularity test case, the component simply composes the coverage information of test methods from a common test class to form the coverage information for each test-class granularity test case.

[6]http://www.cs.gmu.edu/ offutt/mujava/, Accessed in August 2011.
[7]http://www.eclipse.org/jdt/, Accessed in August 2011.
[8]http://asm.ow2.org/, Accessed in August 2011.

Based on the basic coverage-collection component, all the studied test-suite reduction techniques are implemented by the first author strictly according to their original algorithmic details. The G, H, and GRE techniques are simply implemented without external libraries, while the ILP technique is implemented based on the *IBM Symphony library*[9] [20], which provides an API for solving mixed integer linear programming problems and has been widely used in other studies [5], [9].

F. Data and Analysis

As discussed in Section IV-C, our empirical study considers both the reduction in test-suite size (RS) and the lowering in fault-detection capability (RF) of reduced suites. The RS values achieved by each combination of independent variables (IV) for each subject are shown in Table II. In this table, rows 1 to 3 present all the possible choices for the three IVs, where "Meth" and "Stat" are the abbreviations for the method-level and statement-level coverage, respectively. These three rows together indicate each column as a combination of the IVs. In this section, we use a tuple ⟨IV1, IV2, IV3⟩ to denote each combination of the three IVs. For example, ⟨G, TM, Meth⟩ of Column 2 denotes applying the greedy technique on the test-method granularity of test cases using the method-level coverage. Similarly, Table III shows the RF values achieved by each combination of IVs for each subject on the *mutated* faults, while Table IV shows the RF values achieved by each combination of IVs for each subject on the *seeded* faults. Recall that each RF or RS value is the average result over 100 random seeds for the corresponding combination on the corresponding subject.

RQ1: Reduction effectiveness. As shown in Table II, on average, all the 16 combinations of the three IVs offer benefit in reducing the size of test suites, ranging from 12.46% to 65.52%. More precisely, tuples of the form ⟨*, TM, Meth⟩ (i.e.,

[9]http://www.coin-or.org/SYMPHONY/, Accessed in August 2011.

TABLE II
RS VALUES ACHIEVED BY COMBINING DIFFERENT INDEPENDENT VARIABLES

Sub.	The Greedy Technique				Harrold et al.'s Heuristic				The GRE Heuristic				The ILP Technique			
	Test Method		Test Class		Test Method		Test Class		Test Method		Test Class		Test Method		Test Class	
	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat
S_1	57.69	30.76	12.50	12.50	57.69	30.76	12.50	12.50	57.69	30.76	12.50	12.50	57.69	30.76	12.50	12.50
S_2	57.14	28.57	22.22	11.11	57.14	28.57	22.22	11.11	57.14	28.57	22.22	11.11	57.14	28.57	22.22	11.11
S_3	56.36	21.81	50.00	16.66	56.36	23.63	50.00	16.66	56.36	23.63	50.00	16.66	56.36	23.63	50.00	16.66
S_4	73.91	64.13	33.33	33.33	73.91	64.13	33.33	33.33	73.91	64.13	33.33	33.33	73.91	64.13	33.33	33.33
S_5	74.75	65.95	33.33	33.33	75.53	65.95	33.33	33.33	75.53	65.95	33.33	33.33	75.53	65.95	33.33	33.33
S_6	78.96	69.04	38.46	38.46	79.76	69.04	38.46	38.46	79.76	69.04	38.46	38.46	79.76	69.04	38.46	38.46
S_7	62.02	44.30	20.83	8.33	62.02	45.56	20.83	8.33	62.02	45.56	20.83	8.33	62.02	45.56	20.83	8.33
S_8	58.74	41.25	11.99	4.00	58.74	41.25	11.99	4.00	58.74	41.25	11.99	4.00	58.74	41.25	11.99	4.00
S_9	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71
S_{10}	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71	48.71	42.30	14.28	10.71
S_{11}	56.70	49.48	18.75	15.62	56.70	49.48	18.75	15.62	56.70	49.48	18.75	15.62	56.70	49.48	18.75	15.62
S_{12}	67.48	34.81	17.64	2.94	68.61	35.03	17.64	2.94	68.61	35.03	17.64	2.94	68.61	35.03	17.64	2.94
S_{13}	68.65	38.02	15.68	0.00	69.48	39.43	15.68	0.00	69.48	39.43	15.68	0.00	69.48	39.43	15.68	0.00
S_{14}	68.60	37.84	15.68	0.00	69.48	39.43	15.68	0.00	69.48	39.43	15.68	0.00	69.48	39.43	15.68	0.00
S_{15}	69.80	44.53	16.66	6.86	69.90	45.24	17.64	6.86	70.09	45.24	17.64	6.86	70.09	45.24	17.64	6.86
S_{16}	71.57	46.78	16.19	5.71	72.10	47.28	17.14	5.71	72.28	47.46	17.14	5.71	72.28	47.46	17.14	5.71
S_{17}	71.49	46.39	16.19	5.71	72.33	47.19	17.14	5.71	72.51	47.37	17.14	5.71	72.51	47.37	17.14	5.71
S_{18}	72.84	50.87	20.13	10.73	73.17	51.56	20.80	10.73	73.17	51.56	20.80	10.73	73.17	51.68	20.80	10.73
S_{19}	73.47	51.87	20.80	10.06	74.01	52.55	21.47	10.06	74.01	52.66	21.47	10.06	74.01	52.78	21.47	10.06
Avg.	65.13	44.78	21.52	12.46	65.49	45.29	21.74	12.46	65.52	45.32	21.74	12.46	65.52	45.33	21.74	12.46

TABLE III
RF VALUES ON MUTATED FAULTS BY COMBINING DIFFERENT INDEPENDENT VARIABLES

Sub.	The Greedy Technique				Harrold et al.'s Heuristic				The GRE Heuristic				The ILP Technique			
	Test Method		Test Class		Test Method		Test Class		Test Method		Test Class		Test Method		Test Class	
	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat
S_1	24.67	17.00	16.61	15.00	24.93	17.00	16.56	15.00	28.99	17.00	17.99	15.00	25.59	17.00	16.17	15.00
S_2	23.99	13.00	15.00	10.99	23.99	13.00	15.00	10.99	23.99	13.00	15.00	10.99	23.99	13.00	15.00	10.99
S_3	2.00	0.53	2.00	0.00	2.00	0.48	2.00	0.00	2.00	0.00	2.00	0.00	2.00	0.44	2.00	0.00
S_4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_6	4.99	0.00	0.00	0.00	4.99	0.00	0.00	0.00	4.99	0.00	0.00	0.00	4.99	0.00	0.00	0.00
S_7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{10}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{11}	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
S_{12}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{13}	12.71	0.00	6.00	0.00	12.69	0.00	6.00	0.00	11.99	0.00	6.00	0.00	11.03	0.00	6.00	0.00
S_{14}	7.75	1.00	3.00	0.00	7.56	1.00	3.00	0.00	7.00	1.00	3.00	0.00	7.46	1.00	3.00	0.00
S_{15}	6.74	0.00	3.00	0.00	6.59	0.00	3.00	0.00	7.00	0.00	3.00	0.00	6.45	0.00	3.00	0.00
S_{16}	5.87	1.00	1.00	0.00	5.52	1.00	1.00	0.00	4.99	1.00	1.00	0.00	5.51	1.00	1.00	0.00
S_{17}	3.63	2.00	0.00	0.00	3.69	2.00	0.00	0.00	4.00	2.00	0.00	0.00	3.00	2.00	0.00	0.00
S_{18}	3.77	3.79	1.00	1.00	3.74	3.82	1.00	1.00	4.00	4.00	1.00	1.00	3.81	3.73	1.00	1.00
S_{19}	2.41	0.00	0.00	0.00	2.41	0.00	0.00	0.00	1.49	0.00	0.00	0.00	2.40	0.00	0.00	0.00
Avg.	5.23	2.06	2.55	1.47	5.21	2.06	2.55	1.47	5.33	2.05	2.63	1.47	5.11	2.06	2.53	1.47

techniques applied on the test-method granularity test cases using the method-level coverage) have the highest RS values, while tuples of the form ⟨*, TC, Stat⟩ (i.e., techniques applied on the test-class granularity test cases using the statement-level coverage) have the lowest RS values. This observation shows that all combinations of IVs can effectively reduce the sizes of JUnit test suites.

We further analyze the cost of test-suite reduction, i.e., the lowering of fault-detection capability (represented by the RF values) on mutated faults. As shown in Table III, on average, all the 16 combinations of the 3 IVs have positive RF values, ranging from 1.47% to 5.33%, which are smaller than the RS values achieved. For example, tuple ⟨ILP, TM, Meth⟩ reduces the test-suite sizes by 65.52% while lowering fault-detection capability 5.11%, and tuple ⟨H, TM, Stat⟩ reduces the test-suite sizes by 45.29% while lowering fault-detection capability only 2.06%. Moreover, all combinations have zero RF values, i.e., *no lowering* of fault-detection capability, for 7 of the 19 subjects. These results show that various combinations can provide cost-effective JUnit test-suite reduction.

The results on seeded faults in Table IV further confirm this conclusion. On average, the RV values for all the 16 combinations of three IVs range from 0.00% to 12.96%, which are also smaller than the RS values achieved. In addition, the same relative comparisons among combinations of the same technique on mutated faults often hold for the seeded faults. For example, for the ILP technique, ⟨ILP, TM, Meth⟩> ⟨ILP, TC, Meth⟩>⟨ILP, TM, Stat⟩>⟨ILP, TC, Stat⟩ on RF

TABLE IV
RF VALUES ON SEEDED FAULTS BY COMBINING DIFFERENT INDEPENDENT VARIABLES

Sub.	The Greedy Technique				Harrold et al.'s Heuristic				The GRE Heuristic				The ILP Technique			
	Test Method		Test Class		Test Method		Test Class		Test Method		Test Class		Test Method		Test Class	
	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat	Meth	Stat
S_1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_3	14.28	0.00	14.28	0.00	14.28	14.28	14.28	0.00	14.28	14.28	14.28	0.00	14.28	14.28	14.28	0.00
S_4	28.57	0.00	0.00	0.00	28.57	0.00	0.00	0.00	28.57	0.00	0.00	0.00	28.57	0.00	0.00	0.00
S_5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_7	25.00	25.00	0.00	0.00	39.50	0.00	0.00	0.00	25.00	11.00	0.00	0.00	38.00	13.00	0.00	0.00
S_8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_9	0.00	0.00	0.00	0.00	4.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.12	0.00	0.00	0.00
S_{10}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{11}	33.33	33.33	0.00	0.00	33.33	33.33	0.00	0.00	33.33	33.33	0.00	0.00	33.33	33.33	0.00	0.00
S_{12}	25.99	23.66	33.33	0.00	25.66	22.66	33.33	0.00	33.33	33.33	33.33	0.00	26.99	20.66	33.33	0.00
S_{13}	33.33	0.00	33.33	0.00	33.33	0.00	33.33	0.00	33.33	0.00	33.33	0.00	33.33	0.00	33.33	0.00
S_{14}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{15}	19.99	0.00	19.99	0.00	19.99	0.00	19.99	0.00	19.99	0.00	19.99	0.00	19.99	0.00	19.99	0.00
S_{16}	20.63	3.09	9.09	0.00	23.72	2.81	9.09	0.00	18.18	0.00	9.09	0.00	22.45	3.27	9.09	0.00
S_{17}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S_{18}	23.58	8.33	8.33	0.00	23.50	8.33	8.33	0.00	25.00	8.33	8.33	0.00	23.66	8.33	8.33	0.00
S_{19}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Avg.	11.82	4.91	6.22	0.00	12.96	4.28	6.22	0.00	12.15	5.27	6.22	0.00	12.82	4.88	6.22	0.00

values of both seeded and mutated faults, where $>$ denotes that one technique achieves higher RF value than the other. However, there are also differences on RF values for these two types of faults. The RF values achieved on mutated faults are on average smaller than the RF values achieved on seeded faults, indicating less lowering of fault-detection capability on mutated faults than on seeded faults. This supports the conclusions from previous studies that mutated faults and real faults are on average easier to reveal than seeded faults [24], [25]. The only exceptions are the combinations of the form $\langle$*, TC, Stat$\rangle$. We think the reason for the exceptions is the small number of seeded faults which allows test suites reduced by those combinations to occasionally reveal all seeded faults by chance.

For this research question, our results on JUnit test suites are more consistent with the Wong et al.'s study on C programs [14], [15] while greatly differing with the Rothermel et al.'s study on C programs [4]. Rothermel et al.'s study shows that test-suite reduction can compromise the fault-detection capability substantially in many cases, e.g., having RF values of over 50% and even up to 100%. A precise analysis of the causes for this difference would be outside the scope of this paper, but we suggest several potential causes. First, the subject programs used for these two studies differ substantially: the subjects used by Rothermel et al. are small C programs, ranging from 138 to 516 lines of code, while the subjects used in our study are large-scale Java programs, ranging from 1.89 KLoC to 80.44 KLoC. Second, the test suites used are generated differently: the test suites used by Rothermel et al. are automatically generated by a black-box technique and then complemented with manually created tests to achieve certain coverage criteria, resulting in hundreds or thousands of test cases for the small programs. In contrast, the test suites used in our study are mainly JUnit test suites accumulated during real software evolution. Third, the different seeded faults used in these two studies could also be a potential cause.

TABLE V
STATISTICAL ANALYSIS FOR INDEPENDENT VARIABLES' INFLUENCE ON RS

Descriptive Statistics					
IV	Choice	Size	Mean	SD	SE of Mean
IV1	G	76	35.978	23.20	2.66
	H	76	36.250	23.31	2.67
	GRE	76	36.263	23.33	2.68
	ILP	76	36.266	23.33	2.68
IV2	TC	152	17.076	11.30	0.92
	TM	152	55.302	14.65	1.19
IV3	Meth	152	43.554	23.88	1.94
	Stat	152	28.824	19.96	1.62

ANOVA Analysis (at the significance level 0.05)					
Source	DF	SS	MS	F Value	Prob$>$ F
IV1	3	4.54	1.51	0.003	0.9998
IV2	1	111057.39	111057.39	649.04	0.0000
IV3	1	16488.47	16488.47	34.05	$<$ 0.0001
Total	303	162732.60			

RQ2: Effects of IVs on RS values. RS values denote the ratios of test cases reduced by test-suite reduction. Investigating effects of different IVs on the RS values can help to achieve better reduction on sizes of test suites in practice. For each IV, we divide all the RS values by all combinations of IVs on all subjects into groups according to the different choices of this IV. For example, for IV1 we divide all the RS values into four groups, while for IV2 and IV3 we divide all the RS values into two groups. Then, we apply statistical analysis to find the differences between different groups divided by each IV. The statistical results are shown in Table V, where the top part shows the descriptive statistics for each group divided by each IV while the bottom part shows the ANOVA analysis for comparing different groups divided by the same IV. In the top

part, Column 1 presents different IVs, Column 2 shows the choices for each IV, Column 3 shows the sample size for each group divided by the corresponding IV choices, and columns 4 to 6 show the Mean, Standard Deviation, and Standard Error of Mean for each group. In the bottom part, Column 1 shows the IV based on which we divide groups, and columns 2 to 6 show Degree of Freedom, Sum of Squares, Mean Square, F Value, and p value for each group division. From the descriptive statistics, we find that different choices on IV1 achieve similar mean values, while different choices on IV2 and IV3 achieve clearly different mean values. The ANOVA analysis further shows that at the 0.05 significance level, there are no significant differences among groups divided based on different choices of IV1, while there are significant differences between groups divided based on different choices of IV2 and IV3. This result shows that the choice of four studied techniques has no significant influence on the ratios of sizes reduced. In contrast, test-class granularity for JUnit test cases is significantly less effective than test-method granularity, and statement-level coverage is significantly less effective than method-level coverage in reducing the size of original JUnit test suites.

For the four compared test-reduction techniques, although there are no statistically significant differences, there are still slight differences. On all test-case levels and requirement levels, ILP always achieves the largest RS values for all subjects, GRE and H can achieve competitive RS values compared to ILP, while G is slightly inferior to the other techniques. Therefore, we can formulate the performance comparison among them as ILP≅GRE≅H>G, where ≅ denotes performing approximately the same. This comparison results between different techniques obtained on JUnit test suites are mainly consistent with Zhong et al. [9]'s study on C programs. They also find that ILP always achieves the minimal representative subset, and there are no significant differences between ILP, GRE, and H.

RQ3: Effects of IVs on RF values. Similarly with the analysis for RQ2, we perform statistical analysis on RF values achieved on groups divided by different IVs. The top part of Table VI shows the descriptive statistics for the different groups divided by different IVs on seeded faults and mutated faults, and the bottom part of the table shows the ANOVA analysis among groups divided by different choices of the same IV on seeded faults and mutated faults. Similar as with the results on RS values, the ANOVA analysis also shows that there are no significant differences among groups divided by IV1 at the 0.05 significance level, and there are significant differences among groups divided by IV2 and among groups divided by IV3. Every group achieving significantly higher RS values also achieves higher RF values, indicating the fact that higher reduction rate in sizes consistently incurs higher lowering in fault-detection capability of reduced test suites.

Although the ANOVA analysis of RF values is mainly the same with as of RS values in RQ2, there are different trends for the effects of different IVs on the RF values from their effects on the RS values. For the groups divided by IV1, the RF values of different groups have different distributions for seeded and mutated faults. For seeded faults, techniques G and H have smaller average RF values than techniques GRE and ILP, while for mutated faults techniques ILP and H have smaller average RF values than techniques GRE and G. Also technique H can achieve approximately the best reduction in suite size, so we suggest this technique as a reasonable choice for achieving cost-effective JUnit test-suite reduction.

For IV3, on average, changing from Stat to Meth increases the mean RS value from 28.825 to 43.547, while increasing the mean RF value for seeded faults from 2.421 to 9.337 and increasing the mean RF value for mutated faults from 1.768 to 3.899. The rates of increase in RF values on both mutated and seeded faults are consistently larger than that in RS values, indicating that changing IV3 from Stat to Meth might not be so cost-effective. For IV2, changing from TC to TM increases the mean RS value from 17.076 to 55.302 while increasing the mean RF value for seeded faults from 3.114 to 8.643 and increasing the mean RF value for mutated faults from 2.022 to 3.645. Although reducing at the test-case granularity of TM is able to effectively reduce test-suite sizes, the fault-detection capability also drops to approximately the same extent. Therefore, we believe the testers should decide between test-case granularity based on their specific situations: if they care more about the efficiency, they can choose TM; otherwise they can choose TC.

Guideline for JUnit test-suite reduction. Our empirical study shows there exists various combinations of IVs that can effectively reduce JUnit test suites without severely lowering the fault-detection capability. The observations in the above sections also provide the following suggestions for JUnit test-suite reduction in practice:

- All the four techniques studied are not significantly different in RS and RF values, and each of them could be used for reducing JUnit test suites. However, technique H always achieves nearly the largest reduction in test-suite sizes while achieving nearly the least reduction in fault-detection capability on both seeded and mutated faults. Therefore, choosing technique H could be the most cost-effective choice in practice.
- Different test-case granularity levels have significant impacts on RS and RF values. However, the one achieving larger RS values also increases RF values to approximately the same degree, indicating more reduction in fault-detection capability of reduced test suites. Therefore, choosing test-case granularity should be decided based on other constraints. If the testers care more about the efficiency, TM should be used; otherwise, TC should be used.
- Different test-coverage levels also have significant impacts on RS and RF values. As for IV2, the one achieving larger RS values also has larger RF values. However, the rate of increase in RF values when changing from Stat to Meth is faster than the rate of increase in RS values. Therefore, the testers could prefer Stat to Meth for test coverage for JUnit test suite reduction.

TABLE VI
STATISTICAL ANALYSIS FOR INDEPENDENT VARIABLES' INFLUENCE ON RF

Descriptive Statistics										
	Seeded Faults					Mutated Faults				
IV	Choice	Size	Mean	SD	SE of Mean	Choice	Size	Mean	SD	SE of Mean
IV1	G	76	5.743	10.81	1.24	G	76	2.835	5.57	0.64
	H	76	5.870	11.11	1.27	H	76	2.828	5.58	0.64
	GRE	76	5.916	11.12	1.28	GRE	76	2.874	5.84	0.67
	ILP	76	5.986	11.03	1.27	ILP	76	2.797	5.56	0.64
IV2	TC	152	3.114	8.31	0.67	TC	152	2.022	4.49	0.36
	TM	152	8.643	12.52	1.02	TM	152	3.645	6.45	0.52
IV3	Meth	152	9.337	12.81	1.04	Meth	152	3.899	6.49	0.53
	Stat	152	2.421	7.28	0.59	Stat	152	1.768	4.33	0.35

ANOVA Analysis (at the significance level 0.05)										
	Seeded Faults					Mutated Faults				
Source	DF	SS	MS	F Value	Prob> F	DF	SS	MS	F Value	Prob> F
IV1	3	2.39	0.80	0.007	0.9993	3	0.23	0.08	0.0024	0.9998
IV2	1	2323.10	2323.10	20.57	< 0.0001	1	200.38	200.38	6.48	0.0114
IV3	1	3635.28	3635.28	33.48	< 0.0001	1	345.08	345.08	11.34	< 0.0010
Total	303	36429.23				303	9535.85			

In summary, to achieve cost-effective reduction in practice, we suggest using heuristic H (i.e., Harrold et al.'s heuristic [2]) as the reduction technique and Stat (i.e., the statement level) as the test-coverage level. For the test-case granularity, we suggest the testers decide based on real situations between TM (i.e., the test-method granularity) and TC (i.e., the test-class granularity).

G. Threats to Validity

In this section, we describe the internal, external, and construct threats to the validity of our experimentation.

Internal Validity. Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. There are two key threats to internal validity for this empirical study. The first threat involves the potential faults in the implementation of various techniques. To reduce this threat, we implemented all the compared techniques strictly following their original algorithmic details, and we used well-known, third-party libraries. We also carefully analyzed and tested these implementations. The second threat is concerned with the faults used in the studied subjects. To control this threat, we use all the seeded faults that come with the subjects in SIR. In addition, as it has been shown that mutated faults are suitable for use in regression-testing experimentation [22], [24], [25], we also use large number of mutated faults for each subject in the study.

External Validity. Threats to external validity are about whether the observed experimental results and conclusion are generalizable to other subjects. To alleviate these threats for this study, we use as our experimental subjects 19 versions of four real-world Java programs with sizes ranging from 1.89 KLoC to 80.44 KLoC. We obtained both the subject programs and their JUnit test suites from the SIR repository.

Construct Validity. Threats to construct validity are about whether the measurements used in the experimental study reflect the real-world situation. To reduce these threats, we use the metrics RS [9], [10] and RF [4], [14], which were widely used in traditional test-suite reduction works to evaluate the benefits and costs of test-suite reduction. We apply these metrics on JUnit test suites.

V. RELATED WORK

Researchers have investigated many topics on effective and efficient regression testing as summarized in a recent survey [8]. While there is a large amount of work related to our study, we mainly discuss the most related work on test-suite reduction.

Many test-suite reduction techniques have been proposed for reducing test suites, mostly for C programs. There are a number of heuristics for finding minimal representative subsets for original test suites [2], [3], [27], [28]. Harrold et al. [2] are inspired by the fact that essential test cases should be selected as early as possible, and propose a heuristic for iteratively selecting more essential test cases. Chen et al. [3] further combine the features of both essential test cases and 1-to-1 redundant test cases, and propose to reduce test cases by iteratively applying these two strategies. When the procedure cannot continue with these two strategies, Chen et al.'s heuristic just breaks the deadlock with the greedy strategy. There are also works using evolutionary algorithms [29] and integer linear programming (ILP) [12], [13] for test-suite reduction. Mansour et al. [29] aim to find minimal representative sets of regression test suites based on simulated annealing and genetic algorithms. Black et al. [13] consider ILP models for reducing regression test suites. There is additional work aiming to improve the fault-detection capability of reduced test suites. For example, Jeffrey et al. [17], [30] modified the heuristic algorithm proposed by Harrold et al. [2] by selecting redundant test cases to maintain the fault-detection capability.

A number of empirical studies have been available for traditional test-suite reduction. Some empirical studies do not consider the lowering of fault-detection capability caused by test-suite reduction. Chen et al. [10] conduct a simulation study to investigate the ratios of test-suite sizes reduced by different techniques, and show the effectiveness of each technique in various simulated situations. Zhong et al. [9]

study both the size reduction and time taken for reduction by different techniques on C programs, and show that the ILP technique [13] always achieves the minimum representative subset sizes among all compared techniques, while the GRE technique [3] and Harrold et al.'s heuristic [2] perform about the same.

Some empirical studies do take into account the lowering of fault-detection capability caused by test-suite reduction. Wong et al. [14], [15] found that test-suite reduction does not severely influence the fault-detection capability. However, Rothermel et al. [4], [16] found that test-suite reduction can severely reduce the fault detection capability.

Despite the previous works, there are limited studies on test-suite reduction for larger programs and real-world test suites in different test paradigms. In addition, there are limited studies extensively investigating the influence of different factors on both the benefits and costs of test-suite reduction. Our study aims to extensively study the benefits and costs of traditional test-suite reduction techniques on real-world JUnit test suites for larger Java programs.

VI. Conclusions

In this study, we investigated the performance of traditional test-suite reduction techniques on larger programs with more realistic test suites than in previous studies. We implemented and applied four representative traditional test-suite reduction techniques to Java programs with JUnit test suites. We designed and conducted an empirical study which evaluated both the benefits and costs of these test-suite reduction techniques on real-world JUnit test suites. Based on the empirical results, we discussed the similarities and differences between the findings in this study and those in previous studies. Moreover, we also provided a guideline for achieving cost-effective reduction on JUnit test suites in practice.

Acknowledgements

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CNS-0958231, CNS-0958199, CCF-0845628, CCF-0746856, IIS-0438967, and AFOSR grant FA9550-09-1-0351.

References

[1] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: An empirical study," in *Proceedings of International Conference on Software Maintenance*. IEEE, 1999, pp. 179–188.

[2] M. Harrold, R. Gupta, and M. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[3] T. Chen and M. Lau, "A new heuristic for test suite reduction," *Information and Software Technology*, vol. 40, no. 5, pp. 347–354, 1998.

[4] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of International Conference on Software Maintenance*. Published by the IEEE Computer Society, 1998, p. 34.

[5] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of International Symposium on Software Testing and Analysis*, 2009, pp. 213–224.

[6] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing junit test cases in absence of coverage information," in *Proceedings of International Conference on Software Maintenance*. IEEE, pp. 19–28.

[7] M. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi, "Regression test selection for java software," in *ACM SIGPLAN Notices*, vol. 36, no. 11, 2001, pp. 312–326.

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, 2010.

[9] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[10] T. Chen and M. Lau, "A simulation study on some heuristics for test suite reduction," *Information and Software Technology*, vol. 40, no. 13, pp. 777–787, 1998.

[11] A. Aho, J. Hopcroft, and J. Ullman, "The design and analysis of computer algorithms," *Addison-Wesley Series in Computer Science and Information Processing, Reading, MA: Addison-Wesley*, vol. 1, 1974.

[12] J. Hartmann and D. Robson, "Revalidation during the software maintenance phase," in *Proceedings of International Conference on Software Maintenance*. IEEE, 1989, pp. 70–80.

[13] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings of International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 106–115.

[14] W. Wong, J. Horgan, S. London, and A. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of International Conference on Software Engineering*, 1995, pp. 41–50.

[15] W. Wong, J. Horgan, A. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application," *Journal of Systems and Software*, vol. 48, no. 2, pp. 79–89, 1999.

[16] G. Rothermel, M. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[17] D. Jeffrey and N. Gupta, "Test suite reduction with selective redundancy," in *Proceedings of International Conference on Software Maintenance*. IEEE, 2005, pp. 549–558.

[18] T. Chen and M. Lau, "Heuristics towards the optimization of the size of a test suite," in *Proceedings of International Conference on Software Quality Management*, vol. 2, pp. 415–424.

[19] H. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Proc. of International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 419–429.

[20] T. Ralphs and M. Guzelsoy, "The symphony callable library for mixed integer programming," in *Proceedings of the Ninth Conference of the INFORMS Computing Society*. Citeseer, 2005.

[21] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing junit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, no. 1, pp. 33–70, 2006.

[22] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, pp. 733–752, 2006.

[23] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[24] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of International Conference on Software Engineering*, 2005, pp. 402–411.

[25] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, pp. 608–624, 2006.

[26] Y. Ma, J. Offutt, and Y. Kwon, "Mujava: An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[27] A. Offutt, J. Pan, and J. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of International Conference on Testing Computer Software*. Citeseer, 1995.

[28] J. Horgan and S. London, "Atac: A data flow coverage testing tool for c," in *Proceedings of Symposium of Quality Software Development Tools*, 1992, pp. 2–10.

[29] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, pp. 19–34, 1999.

[30] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Transactions on Software Engineering*, pp. 108–123, 2007.