The Journal of Systems & Software 186 (2022) 111174

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An empirical comparison of four Java-based regression test selection techniques☆

Min Kyung Shin [a], Sudipto Ghosh [a,*], Leo R. Vijayasarathy [b]

[a] Department of Computer Science, Colorado State University, USA
[b] Department of Computer Information Systems, Colorado State University, USA

ARTICLE INFO

Article history:
Received 26 October 2020
Received in revised form 12 November 2021
Accepted 25 November 2021
Available online 21 December 2021

Keywords:
Regression test selection
Dynamic analysis
Static analysis
Safety
Precision
Test suite reduction

ABSTRACT

Regression testing is a critical but expensive activity that ensures previously tested functionality is not broken by changes made to the code. Regression test selection (RTS) techniques aim to select and run only those test cases impacted by code changes. The techniques possess different characteristics related to their selection accuracy, test suite size reduction, time to select and run the test cases, and the fault detection ability of the selected test cases. This paper presents an empirical comparison of four Java-based RTS techniques (Ekstazi, HyRTS, OpenClover and STARTS) using multiple revisions from five open source projects.

The results show that STARTS selects more test cases than Ekstazi and HyRTS. OpenClover selects the most test cases. Safety and precision violations measure to what extent a technique misses test cases that should be selected and selects only the test cases that are impacted. Using HyRTS as the baseline, OpenClover had significantly worse safety violations compared to STARTS and Ekstazi, and significantly worse precision violations compared to Ekstazi. While STARTS and Ekstazi did not differ on safety violations, Ekstazi had significantly fewer precision violations than STARTS. The average fault detection ability of the RTS techniques was 8.75% lower than the original test suite.

1. Introduction

Regression testing is an essential process in software development to verify that code changes do not break previously tested functionality. However, as software size keeps growing, the number of test cases and time to test also increase. In 2017, Google had 2 billion lines of code in their source code repository, and developers made 16,000 commits and ran 150 million tests a day (Memon et al., 2017). Clearly, it would be time consuming to run every test each time there is a revision of the code. Since regression testing is expensive, researchers have developed techniques, such as test prioritization, test minimization, and regression test selection (Yoo and Harman, 2012) to reduce the cost.

Regression test selection (RTS) reduces regression testing time by selecting and running a subset of the original suite of test cases that are relevant to the code changes. A typical RTS technique requires two main activities: (1) finding dependencies between the code and test cases, and (2) identifying code changes. Depending on the RTS technique, test dependencies can be collected statically (Legunsen et al., 2017, 2016; Shi et al., 2019) or dynamically (Gligoric et al., 2015; Zhang, 2018; Anon, 2019c). Additionally, dependencies can be analyzed at different levels of granularity such as statement, method, class, and module (Zhang, 2018). There are multiple ways to identify code changes (e.g., Unix diff tool, checksums, and tracking code changes in the background of an integrated development environment (IDE)) (Legunsen et al., 2017; Gligoric et al., 2015; Zhang, 2018; Anon, 2019c; Soetens and Demeyer, 2012; Vokolos and Frankl, 1997). An RTS technique is safe if it does not miss any tests that should be selected; it is precise if it selects only those tests that are impacted.

Researchers have developed many RTS techniques that have different characteristics. First, RTS techniques can achieve different amounts of test size reduction. Safe RTS techniques select all the tests impacted by code changes. Because safe RTS techniques can over-estimate the test dependencies to ensure no impacted tests are missing, they can sometimes select more test cases than precise RTS techniques (Legunsen et al., 2016). Second, RTS techniques offer different savings in end-to-end testing time. This is the total time, including test execution time and any time the RTS technique spent before and after the test execution. For example, a method-level RTS technique may spend more time

☆ Editor: A. Bertolino.

* Correspondence to: Department of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA.
E-mail addresses: mkshin@rams.colostate.edu (M.K. Shin), ghosh@colostate.edu (S. Ghosh), leo.vijayasarathy@colostate.edu (L.R. Vijayasarathy).

https://doi.org/10.1016/j.jss.2021.111174
0164-1212/

on dependency analysis and test selection than a class-level RTS technique in larger-sized programs in terms of KLOC and the number of test classes. However, method-level RTS reduces test execution time by running fewer tests than class-level RTS because a finer granularity technique can select tests more precisely than a coarser granularity technique. Third, RTS techniques have different test selection accuracy, which are characterized by their safety and precision. Preferably, RTS balances safety and precision by not wasting time running too many unnecessary tests or miss tests that would reveal bugs (Shi et al., 2014). Depending on how it finds dependencies, an RTS technique can have varying safety and precision. Lastly, the tests selected by RTS techniques differ in their ability to detect faults in the code. Ideally, the selected tests should find as many defects as the original tests. However, the fault detection ability can differ depending on how safe the RTS technique is.

Researchers have published empirical evaluations to compare RTS techniques. In general, cost reduction and fault detection ability are the two evaluation criteria used in many RTS empirical studies (Engström et al., 2008). To be specific, there are four metrics to measure cost reduction: test suite reduction, test execution, end-to-end time reduction, and precision. There are two ways to measure fault detection ability: relative (e.g., safety) and absolute (e.g., mutation score). Over time, the ways to measure those metrics have evolved. Rosenblum and Rothermel (Bible et al., 2001) used code coverage to compare the precision of RTS techniques. To compute safety, Graves et al. (2001) measured how many tests among the set of tests selected by an RTS technique found seeded faults. Relatively recent studies (Legunsen et al., 2016; Shi et al., 2019; Zhu et al., 2019) compute precision violation in terms of how many more tests did an RTS technique select than the current best technique. They compute safety violation in terms of how many fewer tests did an RTS technique select than the current best technique. On the other hand, Rothermel and Harrold (1997b) and Graves et al. (2001) manually seeded faults in a program, while recent studies (Zhu et al., 2019; Soetens et al., 2016) conduct mutation testing to compute fault detection ability.

To evaluate RTS approaches, measuring both cost reduction and fault detection ability is vital because the techniques should reduce the amount of testing time and still be reasonably safe. However, not many RTS empirical evaluations consider both metrics. Engström et al.'s survey (Engström et al., 2008) shows there are 38 empirical studies out of 923 papers relevant to the research area of regression test selection. Most of those studies measure cost reduction, while only 30% of those studies measure both cost reduction and fault detection ability. The rest use one of the metrics. Since Java has become one of the most widely-used programming languages, many Java-based RTS techniques have been proposed (Legunsen et al., 2017, 2016; Shi et al., 2019; Gligoric et al., 2015; Zhang, 2018). The empirical evaluations reported in these papers show the differences between the proposed new RTS technique and the state-of-the-art technique at the time the papers were written. The studies are conducted on different subjects, program versions, and test environments. Furthermore, researchers used different metrics. Most studies (Legunsen et al., 2017, 2016; Zhang, 2018) measure only time reduction, such as end-to-end time reduction and test execution time reduction. Even though several studies evaluated RTS techniques, the studies were performed independently. Thus, there is a need to compare the techniques in the same manner.

This research aims to evaluate four widely used Java-based RTS techniques: Ekstazi, STARTS, HyRTS, and OpenClover in terms of end-to-end time reduction, amount of test size reduction, safety, precision, and fault detection ability. We answer the following research questions (RQ):

RQ1. To what extent can these RTS techniques reduce the test suite size?

RQ2. To what extent can these RTS techniques reduce the end-to-end testing time?

RQ3. What are the safety and precision violations of these RTS techniques?

RQ4. What is the fault detection ability of test suites selected by these RTS techniques?

RQ5. What (if any) is the relationship between the program features (total size in KLOC, total number of classes, percentage of test classes over the total number of classes, and the percentage of classes that changed between revisions) and the performance of the RTS techniques?

This paper is organized as follows. Section 2 summarizes related work on regression test selection. The four RTS techniques evaluated in this paper are described in Section 3. We explain the design of the empirical study and define the metrics used in Section 4. Evaluation results are presented and analyzed in Section 5. We summarize our conclusions and outline directions for future work in Section 6.

2. Related work

Simply executing all the test cases is also called the *RetestAll* strategy. However, running all tests is time-consuming. Surveys (Yoo and Harman, 2012; Engström et al., 2008) show the various strategies used to develop RTS techniques.

2.1. Evolution of regression test selection techniques

Vokolos and Frankl (1997) introduced an RTS technique based on textual differencing for the first time. They used the Unix *diff* command to find which source files are changed in the new revision at the statement level. While using the diff function is safe and fast, it can be imprecise because it does not determine if the change made a difference to the program semantics.

Initially, RTS techniques were based on control flow graphs, data flow graphs, and slicing, and tools were implemented for various programming languages (e.g., C, C++, Java, and AspectJ). Rothermel et al. (2000) developed a C++ based RTS technique using Class Control Flow Graph (CCFG). Later, the Control Flow Graph (CFG) was extended to Java using the Java Interclass Graph (JIG) and AspectJ using the Inter-module Graph (AJIG) (Harrold et al., 2001a; Orso et al., 2004; Lin et al., 2006; Xu and Rountev, 2007).

Soetens and Demeyer (2012) developed ChEOPSJ, which tracks code changes in the background of the Eclipse IDE and finds dependencies between the code and test cases using the FAMIX model. The tool captures the code changes while developers edit the code. Gligoric et al. (2015), Legunsen et al. (2017), and Zhang (2018), on the other hand, computed a smart checksum that ignores changes that do not impact debug information.

Thereafter, researchers developed RTS techniques that are easy to adapt to the programs written in different programming languages. Romano et al. (2018) proposed an RTS technique using lexical similarities to identify changed methods, and method coverage information for dependencies. An RTS technique introduced by Azizi and Do (2018) is not limited to supporting a specific programming language. It uses the failure history of tests, the program changed history, test case diversity, and the textual similarity of program changes.

2.2. Previous RTS empirical evaluations

As RTS techniques evolved, the techniques used for evaluating them (e.g., evaluation goals, comparison targets for a given RTS technique, programs used for empirical studies, and metrics used) also evolved (Kazmi et al., 2017).

Before researchers started comparing RTS techniques with each other (Graves et al., 2001; Rosenblum and Rothermel, 1997; Harrold et al., 2001b), they often evaluated their proposed technique by itself. Some studies compared the new, proposed technique with *RetestAll* with respect to the time required to select and run tests, and also the reduction in the test suite size (Rothermel and Harrold, 1997a). Other empirical studies were conducted using different sized programs to demonstrate that their technique selects fewer tests (Harrold et al., 1998). The early comparative studies of RTS techniques typically included a comparison with *RetestAll* or random selection (Graves et al., 2001). For the subjects used in the empirical evaluation, researchers seeded faults manually or used subjects with known faults using programs generated by other research projects. The Siemens benchmarks contain realistic faults seeded in seven C programs. Hutchins et al. (1994) generated modified versions of programs and seeded faults to compare dataflow testing and control flow testing empirically. Engström et al. (2008) summarized that 70% of RTS-related empirical studies published before 2006 consider the metrics test suite reduction and total testing time.

Rothermel and Harrold (1997b) defined four evaluation criteria: inclusiveness (safety), precision, efficiency, and generality. Efficiency is related to the time (and space) saved by a technique, and generality is about the ability to handle different languages and complex code structures. Many researchers have also used these criteria to compare test selection accuracy, such as safety and precision (Chittimalli and Harrold, 2008). Chittimalli and Harrold (2008) computed false positives and false negatives of selected tests to calculate safety and precision. They had information on which tests should be selected by their tool because the developers provided the programs. Soetens et al. (2013) conducted a dynamic analysis that executes the original test suite to trace the relationship between tests and source code methods. They computed safety and precision by comparing the list of tests selected by their tool and the result of dynamic analysis. Other researchers have calculated the safety violation and precision violation of a new RTS technique with respect to the current best technique (Legunsen et al., 2016; Shi et al., 2019). In this way, researchers can demonstrate whether a new technique is as safe (or precise) as a state-of-the-art technique.

Collecting real faults in programs for research purposes is challenging, so researchers manually seed faults in a program or use mutation testing. Mutation analysis has been applied to empirically evaluate Java-based regression testing techniques to compute the fault detection ability of the selected test cases (Shi et al., 2014; Soetens et al., 2016, 2013). Researchers compared the mutation scores of the tests selected by the RTS technique with that of the original test suite (Romano et al., 2018). PIT is used for mutation testing because Java projects based on Ant or Maven can easily adopt PIT.

Many researchers have compared RTS techniques empirically (Gligoric et al., 2015; Legunsen et al., 2017; Zhang, 2018; Anon, 2019c; Romano et al., 2018). The survey by Kazmi et al. (2017) shows that there are 25 different metrics used in 47 RTS empirical evaluations. Still, many of these studies focus on time reduction, such as end-to-end time reduction and test execution time reduction rather than safety, precision, and fault detection ability. Furthermore, researchers used various open-source projects for empirical evaluation based on the compatibility with the tools (e.g., Java and JUnit versions).

3. Background

In this section, we explain the four Java-based techniques used in our empirical study. The implementations of these techniques are publicly available. We focus on how the techniques (1) compute the dependencies between the source code and the test cases, and (2) detect changed parts of the code. We present the RTS techniques in chronological order of development. We also present a brief summary of mutation analysis.

3.1. Ekstazi

Ekstazi (Gligoric et al., 2015) is a dynamic, byte-code instrumentation-based RTS technique that uses file-level dependencies. First, Ekstazi compares smart checksums between the previous and current versions of each file to determine whether it changed. Smart checksums ignore debug-related information. Files can be executable code (e.g., class files) and external resources (e.g., configuration files). Then, Ekstazi selects test cases relevant to checksum changed files. All newly added test cases are also selected. During test execution, Ekstazi observes which files are invoked by each test case. Ekstazi collects test dependencies and stores them in a separate file, one per test class. The file includes the names of classes that are accessed during test execution and the class file checksums. In testing subsequent revisions, Ekstazi compares previously saved checksums with the current checksums to determine which files have changed.

Open source projects (e.g., Apache Camel, Commons Math, and CXF) adopted Ekstazi for regression testing. Developers can use Ekstazi by adding a plugin to their projects and there is no need to integrate with version control systems like other RTS tools introduced before Ekstazi such as ChEOPSJ (Soetens et al., 2013). Ekstazi aims to balance the program analysis and test running time. Gligoric et al. (2015) show that Ekstazi on average reduces the end-to-end time compared to retest-all by 32%. Being a dynamic RTS technique, Ekstazi is expected to be more precise than static RTS techniques.

3.2. STARTS

STARTS (Legunsen et al., 2017) is a static RTS technique that uses class-level analysis. In work prior to STARTS, Legunsen et al. (2016) stated that their research was motivated by the usability of Ekstazi (Gligoric et al., 2015). Like Ekstazi, STARTS uses smart checksums to detect changed types such as classes and interfaces. After compiling a new revision, STARTS computes the checksum to determine which types are changed. Then, STARTS eliminates test cases not relevant to the changed types by using the type-to-test mappings that are created during the previous test execution. When there is no previously saved dependency mapping, which is the situation the first time STARTS executes, all types are considered to be changed, and all the test cases are selected for execution. After a test suite execution, STARTS finds test dependencies and updates the mappings for the next revision. Mappings are based on a type dependency graph (TDG), where nodes represent types, and edges indicate the dependencies between types. STARTS utilizes a class firewall technique, and thus, also selects test cases that have dependencies with classes that are impacted by changes in the inheritance hierarchy.

Legunsen et al. (2016) conducted a study that showed class-level static RTS (65.3% of retest-all) was 2.9% faster than Ekstazi (68.2% of the retest-all) in terms of end-to-end testing time. Out of 22 subjects, class-level static RTS had a safety violation with respect to Ekstazi on two revisions, while precision violation was 42.9% on average with respect to Ekstazi. Subsequently Legunsen et al. (2017) demonstrated STARTS can achieve a reduction on

average of 12.4% of the end-to-end testing time compared to retest-all. Since STARTS is a static RTS technique, it is less precise than dynamic RTS techniques. Compared to Ekstazi, STARTS can be unsafe because it does not handle Java reflection.

3.3. HyRTS

Ekstazi and STARTS demonstrated that coarse (e.g., class-level) dependency analysis is faster than fine-grained (e.g., method-level) analysis. STARTS (Legunsen et al., 2016) shows RTS using class-level analysis is ten times faster than RTS using method-level analysis. However, class-level RTS actually selects 2.8 times more test cases than method-level RTS (Gligoric et al., 2015). HyRTS (Zhang, 2018) is a dynamic RTS technique that uses a combination of file-level and method-level analysis. The aim is to implement the fastest RTS by taking advantage of dependency analysis at different levels of granularity.

In HyRTS, newly introduced classes and removed classes are considered as file-level changes. First, HyRTS computes file checksums of the current and old revisions. Only if the file checksums differ, HyRTS computes and compares method checksums. During bytecode instrumentation, HyRTS inserts code to track which methods are invoked during test execution. This enables HyRTS to collect the dependencies between methods and test cases. Class level dependencies can be derived based on the method dependencies since a method belongs to a class. HyRTS provides an offline mode for users who want to get test results faster by collecting dependencies after test execution is over while the online mode is the default option that collects test dependencies during test execution. Zhang (2018) demonstrated the end-to-end time of HyRTS is 21.1% faster while selecting 8.8% fewer test cases than Ekstazi on average. HyRTS is more precise in selecting test cases than class-level RTS and has been proven not to add any new safety issues.

3.4. OpenClover

The Java code coverage tool, Clover (Anon, 2019c), was managed by a software company, Atlassian, and became an open-source project called OpenClover in 2017. OpenClover has an RTS feature called test optimization (Anon, 2019a), which dynamically computes dependencies using source code instrumentation and analyzes the dependencies between test cases and source code at the file level. Ekstazi considers both executable code and external resources, but OpenClover only considers executable code. OpenClover compares file sizes and checksums in the current and previous revisions to identify file changes. File checksums are stored in a variable of type long, which can overflow after a certain value. Thus, both the checksums and file size are used for comparison. During test execution, OpenClover tracks per-test coverage and updates the coverage information in a database to compute dependencies between source files and test cases for the next run.

By default, OpenClover runs a clean build every ten test executions to remove any collected data. In this way, OpenClover detects potential non-Java file changes and updates dependencies that may be missed. Since a clean build removes previous test results, such as saved file checksums, OpenClover runs a full test on the subsequent test execution. However, running a full test may increase the overhead, so users are allowed to change the default number of executions after which clean build should be run. Experiments (Anon, 2019a) show OpenClover runs 10% of full test cases, which takes 30% of build time compared to using a normal build, which executes all the test cases. However, since OpenClover is not a research tool, OpenClover's official website does not provide the details of the experiment.

3.5. Mutation testing

Mutation testing (Jia and Harman, 2010) is a software testing technique used to assess the quality of tests by seeding faults in the code. Mutation testing has a step to create the faulty version of the programs, called mutants. A mutant is a modified version of the original source code based on applying a mutation operator, where mutation operators make changes in the code by modifying constructs such as variables, relational, arithmetic, and logic operators. Because mutants changed from the original program, each mutant is considered a faulty version of the original program. Depending on the type of mutation operators, many mutants can be generated, and each mutant includes only one fault. If any test fails on a mutant, we consider that mutant to be killed. Otherwise, the mutant is live. Researchers have used mutation testing to evaluate tests as a way to measure test quality, and studies have demonstrated that mutation testing could replace manual faults seeding in a program (Andrews et al., 2005). Accordingly, many mutation testing tools have been developed (Kintis et al., 2018), such as MuClipse, MuJava, Major, and PIT. We selected PIT for our evaluation process because PIT is easily adopted by Maven-based Java projects and has been widely used in other research studies (Al-Refai, 2019; Soetens et al., 2016; Shi et al., 2014).

4. Research design

This section describes the design of the empirical study. Section 4.1 defines the five metrics used in the evaluation and the four program features, which may have an impact on the metrics. Section 4.2 describes the subject programs used in our empirical study. Section 4.3 lists the steps to execute RTS tools. In Section 4.4, we address how the mutation testing is conducted. Section 4.5 explains how we extracted raw data from log files, calculated metrics, and visualized the data. Finally, Section 4.6 explains the statistical analysis used to compare RTS techniques.

The empirical study was conducted on a Linux (Fedora) workstation, a 4-core 3.2 GHz machine with 12 GB memory, Java 64-Bit Server version 1.8.0_242. We fully automated the entire experiment to avoid human error, save time, and ensure repeatability.

4.1. Evaluation metrics

We used five metrics to evaluate the four RTS techniques: test-suite reduction, end-to-end time reduction, safety violation, precision violation, and fault detection ability. These metrics are the dependent variables. The independent variables are RTS technique, total size in KLOC, total number of classes, percentage of test classes, and percentage of changed classes. The RTS techniques are Ekstazi, STARTS, HyRTS, OpenClover, and also *RetestAll*, which runs all test cases. The rest of the variables are defined below.

Test-suite reduction. Given a program P, original test suite T, modified version P' and selected test suite T',

$$TestSuiteReduction = \frac{|T| - |T'|}{|T|} \quad (1)$$

Note cardinality refers to the number of tests in a test suite. Higher test suite reduction is better.

End-to-end time reduction. The end-to-end time is the total time, which includes test execution time and any time that the RTS technique spends before or after test execution. Higher reduction is better. Given the original testing time t and the testing time using the RTS technique, t',

$$EndToEndTimeReduction = \frac{t - t'}{t} \quad (2)$$

Safety violation. Assume there are two tools, RTS_1 and RTS_2, which select and run test suites T_1 and T_2, respectively. The safety violation metric calculates the safety violation of RTS_2 with respect to RTS_1. The denominator is the cardinality of the union of two test suites, and the numerator gives the number of tests that RTS_1 selected, but RTS_2 did not select. Here, lower safety violation is better if RTS_1 is known to be safe.

$$SafetyViolation = \frac{|T_1 - T_2|}{|T_1 \cup T_2|} \tag{3}$$

Precision violation. We use the same assumptions as above. Only the numerator changes here because we want to calculate how many extra tests were selected by RTS_2 with respect to RTS_1. If RTS_2 selects more tests that should not be selected, the numerator increases. Thus, lower precision violation is better if RTS_1 is known to be precise.

$$PrecisionViolation = \frac{|T_2 - T_1|}{|T_1 \cup T_2|} \tag{4}$$

Fault detection ability. We calculated the fault detection ability of the test suites selected by all the RTS techniques as well as the original test suite using the mutation score. We did not collect data on equivalent mutants, which is common practice in current research.

$$FaultDetectionAbility = \frac{KilledMutants}{TotalNumberOfMutants} \tag{5}$$

A test suite selected by an RTS technique should kill as many mutants as possible, so higher fault detection ability is better. However, the selected test suite cannot exceed the original test suite's ability to detect faults.

To identify how RTS techniques are affected by program features, we used four features that may have an impact: total (program and test code) size in KLOC, total number of classes, the percentage of test classes out of the all the classes, the percentage of changed classes between revisions.

Total size in KLOC. Total size in KLOC measures the size of a program and test cases by counting the number of code lines. We considered the projects with over 100 KLOC as large-sized programs as other RTS empirical studies (Rothermel and Harrold, 1996; Engström et al., 2010) considered. As such, the projects with less than 100 KLOC are considered as relatively smaller sized programs.

Total number of classes. In object-oriented programming, the number of classes is often used as a metric to measure the size of programs (Tang et al., 1999). This factor measures the total number of classes including program and test cases.

Percentage of test classes. This factor measures the percentage of test classes over total number of classes:

$$= \frac{NumberOfTestClasses}{TotalNumberOfClasses} \times 100 \tag{6}$$

RTS techniques compute dependencies between the code and test cases, so the portion of test classes out of total number test classes is an important factor that may impact their performance.

Percentage of changed classes. This factor measures the percentage of changed classes over total number of classes:

$$= \frac{NumberOfChangedClasses}{TotalNumberOfClasses} \times 100 \tag{7}$$

We used STARTS to count the number of files for which smart checksums changed between revisions. Thus, we ignore the changes that do not affect the program behavior. We explain the details of steps for collecting this data in Section 4.3.

4.2. Subject selection

Table 1 shows the subjects used in our empirical study. These programs were used by other researchers (Gligoric et al., 2015; Legunsen et al., 2017; Zhang, 2018). However, we used different revisions for the comparison. We first found the *head* revision that does not have a build or compile error, and no test failures with the four RTS techniques. We then selected up to a hundred and fifty revisions that successfully ran with all four RTS techniques. In some cases, the revisions gave errors with one or more RTS tools, and were removed. The five open-source Java projects met the prerequisites for the RTS tools: (1) Maven version 3.2.5 or above, (2) Surefire version 2.14 or above, (3) JUnit version 3 or above, (4) Java version 1.8 or above.

For the purpose of this research, it is important to select various qualitative subjects according to program features rather than selecting many subjects. Besides, there is a limitation in finding open-source projects that meet all the prerequisites required by the four RTS techniques. The five subjects used in this research vary in terms of the program features such as total size in KLOC, percentage of test classes over a total number of classes.

Table 1 shows the number of revisions used, the average number of implementation and test classes, the average percentage of test classes in the total number of classes, sizes (Line of Code) of each subject on average over revisions. The subjects range in code size from 16 KLOC to 204 KLOC. In total, the study involves 583 revisions that include 308K test classes and 56 million LOC.

Below, we describe each subject in detail.

Asterisk. Developed since 2006 by Digium, Asterisk is a framework for communication applications. Two Asterisk Git repositories exist for C and Java. For our study, we used the Java version called Asterisk-Java. Henceforth, we will call it Asterisk for convenience. Asterisk is the largest-size (KLOC) subject in the study. It has 825 source classes and 67 test classes on average per revision, meaning that it has the smallest percentage of test classes in the total number of classes among the five subjects.

Commons CLI. Commons CLI provides an API for parsing command-line options passed to programs. Commons CLI has the smallest program size (KLOC) but highest percentage of test classes in the total number of classes over all subjects.

Commons Collections. Commons Collections is an Apache framework that provides data structures in Java. Commons Collection has the most commits, and the second highest number of test classes and percentage of test classes in the total number of classes.

Commons Imaging. Commons Imaging is a Java image handling library that can quickly parse image data and support a variety of image formats. The number of revisions from Commons Imaging is the highest out of all subjects because 96.67% of the considered revisions had build success on all RTS tools.

Commons Net. Commons Net provides network utilities and internet protocols for Java. It has the longest time of running the original test suite though it has a relatively small number of test classes compared to other subjects used in this study.

After selecting the subjects, we paired the Git SVN URL with the head hash for each subject and placed them in a file. Our bash script read the file line by line and downloaded each subject (master branch). In the subject directory, the Git log utility provides historical hash numbers and comments. We specify head hash as the oldest revision. Then, we printed the hashes backwards to get the older version first and newer versions later. Finally, we downloaded revisions of each subject using the list of hashes.

Table 1
Summary of the subjects.

Subject	Revisions	Total number of classes	% of test classes	Total size (KLOC)
Asterisk	129	825	8.12	204
Commons CLI	93	56	55.36	16
Commons Collections	104	791	37.80	129
Commons Imaging	145	578	30.62	57
Commons Net	112	274	24.82	64

4.3. RTS tool execution

We automated the process to run Ekstazi, STARTS, HyRTS, and OpenClover. Given a program P, there are revisions from P_1 to P_n. We created a working directory before repeating the following five steps for each tool.

1. Copy P_i to the working directory.
2. Add RTS tool plugin to `pom.xml`.
3. Run RTS tool and redirect standard output to a file (`LogFile`).
4. Move the RTS tool result (LogFile and directories generated by RTS tool) back to P_i's directory.
5. If P_{i+1} does not exist, clean the working directory and move to the next subject.

We ran the above steps three times for end-to-end time measurement because time measurements can be sensitive to the environment. We took an average of three times of tools execution results.

We also collected a list of changed files in each revision. The lists are used for mutating only the changed program files. The list of changed files is generated by running `STARTS: diff` between steps 2 and 3 when running STARTS. The `diff` command prints the list of files that STARTS identified as changed by computing smart checksums. STARTS (Legunsen et al., 2017) reuses the part of the Ekstazi source code to computes checksum. HyRTS (Zhang, 2018) also computes the checksum in the same way as Ekstazi and STARTS. However, STARTS is the only RTS tool that provides the command-line option to show files identified as changed among the four RTS tools.

4.4. Mutation testing

In our experiment, we conducted mutation testing to compare the fault detection ability of the tests selected by the RTS techniques. Two tasks had to be performed: (1) generate mutants for each revision, and (2) execute the original tests and those selected by each RTS tool on the mutants.

We had a choice of mutation tools such as MuJava, PIT, and Major. Each support different sets of mutation operators. We chose PIT because it worked with the selected subjects and our computing environment. One drawback of not using Major is that we were unable to perform mutations for selected methods and had to run mutations on selected files. PIT mutates all the methods in a class, which is a problem for HyRTS since it only selects test cases that execute changed methods.

First, we created mutants with PIT. We ran PIT if changed classes exist in the program. That is because we specified the classes names for seeding faults only in the changed classes, and PIT crashes if there are no classes to mutate. We only mutate the changed classes since developers can introduce new faults only in the changed classes.

We wrote a script that automatically edited the PIT configuration file by specifying a list of changed files that should be mutated and ran the necessary test cases: all the original tests or only the tests selected by each RTS tool. These tests are extracted from the logs generated during the execution of each RTS tool. At the end of the test, PIT prints the total number of generated mutants, the number of killed mutants, and the details of mutants such as which mutation operator was used. We redirected the standard output and error messages generated from PIT to a file to calculate and compare the fault detection ability in the evaluation step.

4.5. Data collection and visualization

We collected three different formats of data in this experiment. We collected log files obtained from running the RTS tools and PIT (unrefined mutation testing results). We extracted raw data from log files, such as time taken during the testing, test class names selected and run by tools, and the number of killed mutants. We utilized a regular expression to capture the parts after a particular set of words. Then, we saved the raw data into CSV files. Second, we calculated five evaluation metrics and saved them in Excel files. Third, we visualized those metrics using Excel. The experimental software and data are available on the website https://github.com/ericashin221/RTS_comparison for repeatability.

4.6. Statistical data analysis

We conducted both non-parametric and parametric statistical analysis to compare the RTS techniques by testing the following hypothesis (expressed in null form): There are no significant differences in (a) test suite size reduction, (b) reduction in end-to-end time, (c) safety and precision violation, and (d) fault detection ability among the RTS techniques.

We first used the less powerful (i.e., less likely to detect the effect of an independent variable/factor) non-parametric tests because our data was not normally distributed. After confirming the statistical significance of our independent variable/factor (i.e., RTS technique), we corroborated these results using the more conservative parametric tests since our sample size is sufficiently large to meet the requirements for these tests. The parametric test results also offer critical and detailed insights into interaction effects and pair-wise comparisons between RTS techniques and different levels of program characteristics.

We used nparLD to identify if the values of evaluation metrics for each research question were statistically significant. The nparLD is a package for R that provides a function to analyze the non-parametric variance of longitudinal data by means of the Wald-type statistic (Noguchi et al., 2012). The Wald test is used to statistically determine if an independent variable/factor (e.g., RTS technique) shows (or predicts) differences in a dependent variable/metric (e.g. test suite size reduction). A large test statistic (i.e., Wald-type statistic) indicates that the null hypothesis (i.e., there is no difference in the dependent variable/metric by the independent variable/factor) can be rejected.

The nparLD is known for being robust even for smaller sample sizes. We used the nparLD because our data set is not normally distributed, and certain groups (e.g., groups that have multiple

numbers of changed files between revisions) have a smaller sample size than other groups. We used an alpha of 0.05 as the cut-off for our statistical tests.

Akritas and Brunner (1997) defined the Wald-type statistic as follows:

$$Q_n(\mathbf{C}) = n\widehat{\mathbf{p}}^{\top}\mathbf{C}^{\top}[\mathbf{C}\widehat{\mathbf{V}}_n\mathbf{C}^{\top}]^{+}\mathbf{C}\widehat{\mathbf{p}} \quad (8)$$

under the hypothesis H_0^F: $\mathbf{CF} = \mathbf{0}$ where C is a contrast matrix, **F** is the vector of distributions, p is the vector of the relative marginal effects, and $\widehat{\mathbf{V}}_n$ is the empirical covariance matrix of the ranks. The Wald-type statistic is a popular and robust method for testing both simple and composite null hypotheses (Ghosh et al., 2016). The non-parametric test was conducted on RStudio using the R software package provided by Noguchi et al. (2012). However, the statistical test results using nparLD do not show pair-wise differences between groups. Thus, as a post-hoc analysis, we conducted a Bonferroni test for multiple comparisons to identify significant pair-wise differences. The Bonferroni correction accounts for multiple comparisons by adjusting the significance level for testing each hypothesis. Specifically, the significance threshold is reduced as follows: α/n, where n is the number of hypotheses tested. This correction reduces the likelihood of Type 1 errors when conducting multiple comparison tests. The Bonferroni test is the most simple and widely used test for multiple comparisons (Ludbrook, 1998).

We corroborated the non-parametric test results produced by the nparLD package by conducting parametric repeated measures MANOVA (multivariate analysis of variance) tests. Compared to their non-parametric counterparts, parametric tests are generally more conservative, and less likely to make statistical errors (e.g., false positives) (Chan, 2003). We conducted mixed factorial MANOVA to test for both within-subject effects (i.e., differences in evaluation metrics by RTS technique) and between-subject effects (i.e., differences in evaluation metrics by program characteristics). In addition to testing the main effects of the within-subject factor (i.e., RTS technique) and between-subject factors (i.e., program characteristics), we tested for interaction effects between the two. We accounted for a violation of the assumption of sphericity by correcting the degrees of freedom using the Huynh–Feldt's estimates of sphericity. The Huynh–Feldt correction is best known for producing a more accurate significance *p*-value for the MANOVA test (Baguley, 2004). For the post hoc tests that involved pair-wise comparisons between RTS techniques and the different levels of program characteristics, we used Bonferroni corrections. The parametric tests were conducted using SPSS, a commercial statistical and data analysis software tool.

5. Results

In this section, we present answers to the five research questions and then discuss several threats to validity. Sections 5.1–5.5 show the results of the empirical study and answer the research questions. Section 5.6 discusses the results of our empirical evaluation and compares them with the results from other studies. The threats to validity of the empirical study are discussed in Section 5.7.

5.1. Reduction in test suite size

The calculation of test suite reduction is illustrated using an example. Asterisk has 252 original test cases in revision 6. The numbers of test cases selected by STARTS, Ekstazi, HyRTS, and OpenClover are 49, 61, 4, and 18, respectively. Thus, the test suite reductions obtained by the tools are $80.556 = \frac{252-49}{252}x100$, $75.794 = \frac{252-61}{252}x100$, $98.413 = \frac{252-4}{252}x100$, and $92.857 = \frac{252-18}{252}x100$, respectively.

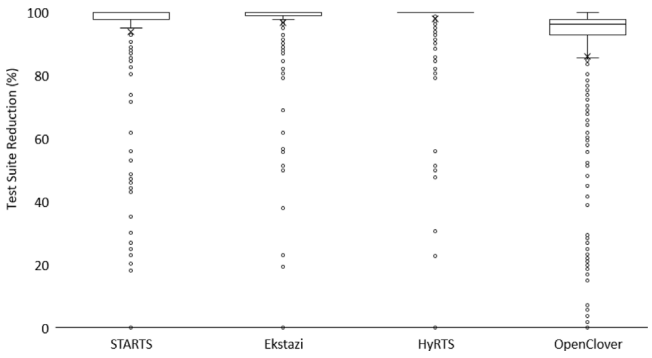

Fig. 1. Test suite reduction.

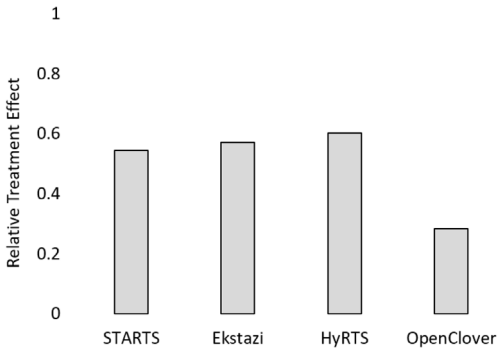

Fig. 2. Non-parametric test result for test suite size reduction.

The total number of test cases in the original revision of the five subjects is 26,438. The test suite reduction mean values for STARTS, Ekstazi, HyRTS and OpenClover are 94.087, 96.978, 97.983 and 86.197, respectively.

The boxplots shown in Fig. 1 display the percentage of test suite reduction obtained by each tool for all the revisions considered in the study. Note that the box plots represent the mean values using the symbol 'x'.

OpenClover's median test suite reduction is 3.54% lower than STARTS, Ekstazi, and HyRTS. The third quartiles of STARTS, Ekstazi, and HyRTS are the same or close to their respective median and maximum values because 64.35% of revisions do not have files whose smart checksums changed. The mean values show Ekstazi and HyRTS select fewer test cases than STARTS. STARTS being a static technique, over-estimates test dependencies and selects more test cases. Even though OpenClover is a dynamic technique like Ekstazi and HyRTS, OpenClover's mean value is 7.89% higher than the static technique, STARTS. The reason is that OpenClover considers multiple elements to identify code changes. Thus, OpenClover identifies more source files as having changed and accordingly selects more test cases. OpenClover's median value also shows this tool selects and runs test cases from the revisions on which the other tools did not run any test case.

Fig. 2 shows the bar chart for the non-parametric test result for differences in the test suite size reduction achieved by the four tools on all the revisions that were considered in the study. The relative treatment effects (RTE) appear in the following order: OpenClover < STARTS < Ekstazi < HyRTS. This can be interpreted to mean that HyRTS achieved the highest test suite size reduction while OpenClover achieved the lowest. For this non-parametric test, the *p*-value of the Wald-Type statistic was 4.08739e−263 ($p < 0.05$) indicating the differences in test suite size reduction are statistically significant.

The non-parametric test result generated by nparLD does not show if differences in test suite size reductions between pairs of

Table 2
Bonferroni test result for test suite size reduction.

Comparison	Difference	p-value	Sig.
Ekstazi - HyRTS	−1.20925	1.0000	
Ekstazi - OpenClover	10.78555	0.0000	***
Ekstazi - STARTS	2.88739	0.0122	*
HyRTS - OpenClover	11.99479	0.0000	***
HyRTS - STARTS	4.09663	0.0001	***
OpenClover - STARTS	−7.89816	0.0000	***

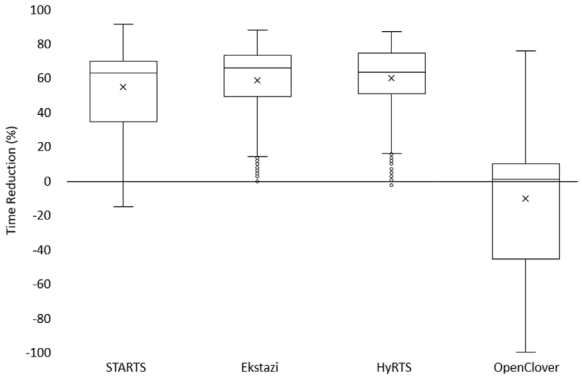

Fig. 3. End-to-end time reduction.

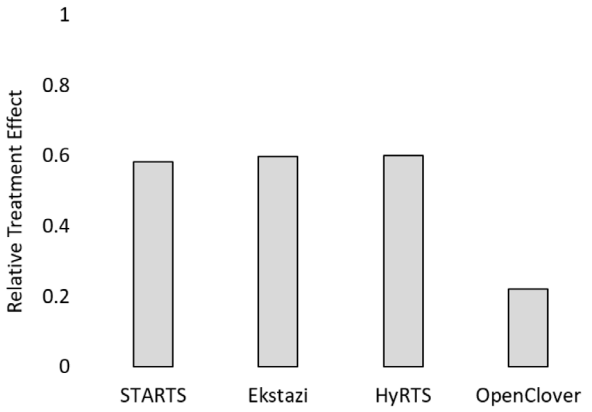

Fig. 4. Non-parametric test result for time reduction.

Table 3
Bonferroni test result for end-to-end time reduction.

Comparison	Difference	p-value	Sig.
Ekstazi - HyRTS	−0.90925	1.0000	
Ekstazi - OpenClover	67.41118	0.0000	***
Ekstazi - STARTS	2.53394	1.0000	
HyRTS - OpenClover	68.32043	0.0000	***
HyRTS - STARTS	3.44320	0.5416	
OpenClover - STARTS	−64.87724	0.0000	***

RTS techniques are statistically different. For example, the difference in RTE between HyRTS and STARTS is 0.058. But this difference does not indicate if STARTS achieved lower test suite size reduction than HyRTS. Similarly, although OpenClover achieved the lowest test suite size reduction with an RTE value of 0.2826, it is not certain if the differences between the number of test cases selected by OpenClover and the other techniques are statistically significant.

Therefore, we conducted Bonferroni tests, and the results are shown in Table 2. Each row in the table shows if the difference in test suite size reduction between two RTS techniques is statistically significant. The last column shows the significance of the p-values visually, with the number of '*' symbols indicating the significance level (*: $p <= 0.05$, **: $p <= 0.01$, ***: $p <= 0.001$). The results indicate that (a) OpenClover had lower test suite size reduction compared to the other techniques, (b) STARTS selects more test cases than Ekstazi and HyRTS, and (c) the test suite reductions achieved by Ekstazi and HyRTS are not significantly different from each other.

The parametric test with Huynh–Feldt correction confirmed the significant effect of the within-subject factor (i.e., RTS technique) on test suite size reduction ($F = 40.11$, $df = 1.37$, $p < 0.000$).

5.2. Reduction in end-to-end time

The sum of the total testing time taken to run the original revision of the five subjects is 226.867 s. The end-to-end time reduction mean values for STARTS, Ekstazi, HyRTS and OpenClover are 55.370, 58.301, 59.404 and −10.403, respectively.

Fig. 3 shows boxplots for the reduction in end-to-end time achieved by each of the four tools on all the revisions that were considered in the study. It shows that the highest median and mean values for end-to-end time reduction are achieved by Ekstazi (65.44%) and HyRTS (59.40%), respectively. OpenClover achieved the lowest median end-to-end time reduction at 0.58%. The mean value of end-to-end time reduction achieved by Ekstazi is 2.93% higher than STARTS. This pattern is consistent with Ekstazi achieving a 2.89% higher test suite size reduction than STARTS.

It is notable that the RTS techniques do not guarantee a reduction in testing time in all instances. We observed that STARTS, Ekstazi, and OpenClover achieved negative end-to-end time reduction in some revisions. This indicates that using RTS took a longer time than running the original test suite. In particular, OpenClover took longer to run the original test suite on 16 times more revisions than STARTS. OpenClover's official website (Anon, 2019b) indicates this technique's performance, in terms of testing time and memory usage, deteriorates as the number of class files and test cases increase. This limitation is attributed to the number of per-test coverage files generated by OpenClover being equal to the number of class files multiplied by the number of test cases.

Fig. 4 shows the non-parametric test result for differences in end-to-end time reduction among the four RTS techniques. OpenClover (0.2206) and HyRTS (0.5995) have the smallest and largest RTE, respectively. The Wald-Type statistic and its *p*-value indicated the difference in end-to-end time reduction among the RTS techniques is statistically significant.

Table 3 shows the result of the Bonferroni test for end-to-end time reduction achieved by the RTS techniques. OpenClover's end-to-end- time reduction is significantly lower in comparison to each of the other three techniques.

The parametric tests confirmed the non-parametric results. The end-to-end time reductions are significantly different across the RTS techniques ($F = 1143.64$, $df = 1.45$, $p < 0.0000$). Further, the pair-wise comparisons indicate OpenClover has the lowest time reduction among the four techniques, and the time reduction achieved by STARTS is significantly lower than Ekstazi and HyRTS.

5.3. Safety and precision violation

Figs. 5 and 7 show the safety and precision violations of the tools with respect to STARTS, Ekstazi, and HyRTS. We used these tools as baselines because they are considered to be the state-of-art RTS techniques (Al-Refai, 2019). In Figs. 5–8 and Tables 4 and

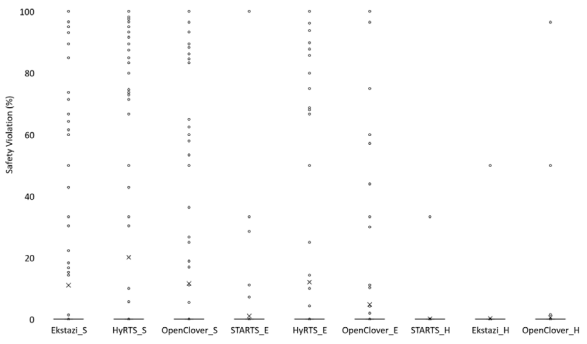

Fig. 5. Safety violation.

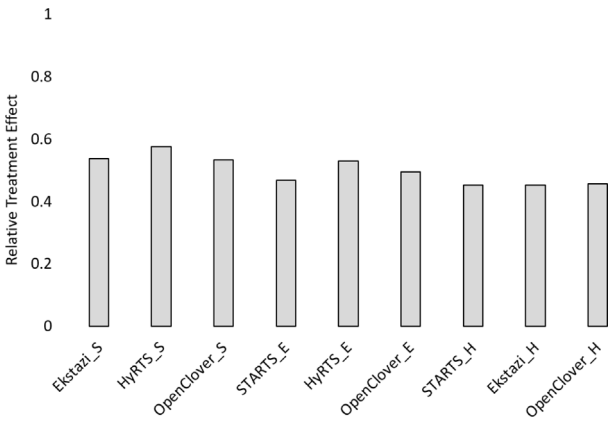

Fig. 6. Non-parametric test result for safety violation.

5, the symbols *_S*, *_E* and *_H* denote the violations computed with STARTS, Ekstazi and HyRTS, respectively.

Fig. 5 shows the median safety violation values of the four RTS techniques is 0 but there are many outliers. Considering the mean value of test suite size reduction of the four RTS techniques is 93.81%, 9.08 test cases are selected from each revision on average. This means the safety violations of RTS techniques increase by around 8.29% for every test case that should have been selected but was not. The mean safety violation of STARTS is 0.87% with respect to Ekstazi, which is the lowest safety violation among all the nine violations. HyRTS achieved the best test suite size reduction but has the highest safety violation. The mean safety violation of HyRTS is 6.38% higher than Ekstazi with respect to STARTS and 8.16% higher than STARTS with respect to Ekstazi. OpenClover selects the most test cases, but OpenClover's safety violation is 9.01% with respect to STARTS and 2.61% with respect to Ekstazi. This means approximately 8% of test cases OpenClover selects are irrelevant to the code changes that STARTS and Ekstazi identified.

Fig. 6 shows the gap between the largest (0.5467 for HyRTS with respect to STARTS) and the smallest (0.4481 for STARTS with respect to Ekstazi) RTE is less than 0.1. Despite the small differences in RTE among the techniques, the *p*-value of the Wald-Type statistic is less than 0.05, indicating significant differences in safety violations.

The Bonferroni test result (Table 4) shows safety violations for HyRTS computed with respect to STARTS and Ekstazi are higher compared to its counterparts. Further, the safety violations for STARTS computed with respect to Ekstazi and HyRTS are lower compared to OpenClover. This can be explained by the differences in test suite reduction achieved by the RTS techniques. STARTS, being a static RTS technique, selects more test cases than dynamic

Table 4
Bonferroni test result for safety violation.

Comparison	Difference	p-value	Sig
Ekstazi_S - HyRTS_S	−15.114	0.000	***
HyRTS_S - OpenClover_S	12.563	0.000	***
Ekstazi_S - OpenClover_S	−2.551	0.487	
HyRTS_E - OpenClover_E	10.185	0.000	***
HyRTS_E - STARTS_E	18.685	0.000	***
OpenClover_E - STARTS_E	8.500	0.000	
STARTS_H - Ekstazi_H	−0.091	1.000	***
STARTS_H - OpenClover_H	−4.777	0.000	***
Ekstazi_H - OpenClover_H	−4.686	0.000	***

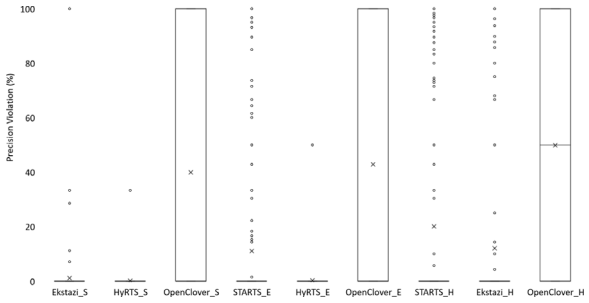

Fig. 7. Precision violation.

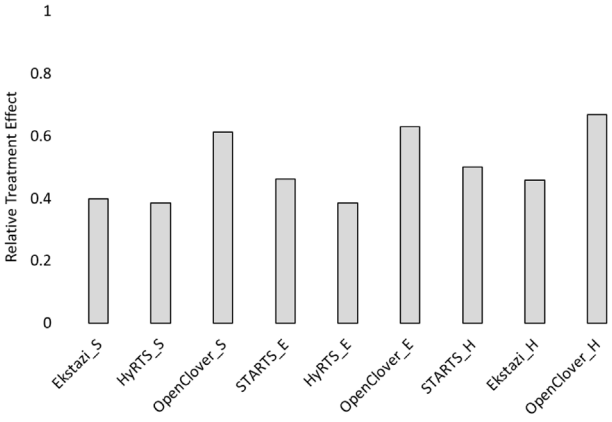

Fig. 8. Non-parametric test result for precision violation.

RTS techniques. The parametric test with Huynh–Feldt correction confirmed that RTS technique has significant effects on safety violations with respect to STARTS ($F = 126.261$, $df = 2.77$, $p < 0.000$), Ekstazi ($F = 73.885$, $df = 1.83$, $p < 0.000$) and HyRTS ($F = 31.026$, $df = 1.09$, $p < 0.000$).

In Fig. 7, the average precision violations of HyRTS with respect to both STARTS and Ekstazi are both close to zero. HyRTS and Ekstazi select 4.10% and 2.89% fewer test cases than STARTS. This can be explained by the observation that HyRTS and Ekstazi do not have many outliers and have low average values with respect to STARTS. OpenClover's average precision violations are the highest among all the violations (higher than 60%). The reason is that OpenClover's third quartile of precision violation is 100%. As we saw in the discussion of safety violation, the average number of test cases selected by the four RTS techniques in each revision is 9.08. We observed that OpenClover selects more than 18 test cases in 103 revisions. This means OpenClover selected two times more test cases than the average test case selection in 17.82% of total revisions. This also explains why there is a 100% precision violation.

In Fig. 8, the precision violations of OpenClover are the highest among all the violations with respect to STARTS, Ekstazi,

Table 5
Bonferroni test result for precision violation.

Comparison	Difference	p-value	Sig
Ekstazi_S - HyRTS_S	1.206	0.121	
HyRTS_S - OpenClover_S	−18.795	0.000	***
Ekstazi_S - OpenClover_S	−17.589	0.000	***
HyRTS_E - OpenClover_E	−22.255	0.000	***
HyRTS_E - STARTS_E	−17.426	0.000	***
OpenClover_E - STARTS_E	4.829	0.744	
STARTS_H - Ekstazi_H	12.691	0.000	***
STARTS_H - OpenClover_H	−3.097	1.000	
Ekstazi_H - OpenClover_H	−15.788	0.000	***

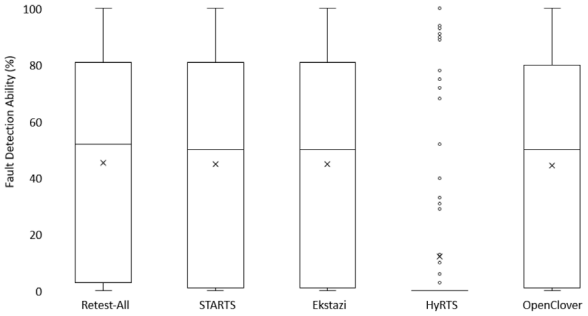

Fig. 9. Fault detection ability.

and HyRTS with RTE values of 0.6115, 0.6295, and 0.6684. The post-hoc test results in Table 5 show the precision violations of OpenClover are significantly higher than other techniques. In addition, the precision violation of HyRTS with respect to Ekstazi is statistically lower than STARTS, and the precision violation of Ekstazi with respect to HyRTS is lower than STARTS. The parametric test with Huynh–Feldt correction confirmed the significant effect of the within-subject factor (i.e., RTS technique) on precision violations with respect to STARTS ($F = 45.206$, $df = 1.07$, $p < 0.000$), Ekstazi($F = 55.867$, $df = 1.55$, $p < 0.000$), and HyRTS ($F = 87.403$, $df = 2.09$. $p < 0.000$).

5.4. Fault detection ability

Fig. 9 shows the boxplots for the fault detection ability scores obtained by running all the test cases (RetestAll) and the test cases selected by the four tools. PIT ran successfully on 146 revisions out of 578 revisions, and a total of 30,354 mutants were generated by PIT. The low number (25.26%) can be attributed to two reasons.

First, 35.47% of all the revisions used in our study have files whose smart checksums changed between revisions. As explained in Section 4.4, we conducted mutation testing only on the revisions that have changed files. This means 64.53% of revisions were excluded from the mutation testing. Second, mutation testing failed on some revisions due to a phantom error. This is an issue that occasionally occurs with PIT due to the causes listed here (Anon, 2019d): PIT configuration problem, mismatched configuration between test and PIT, hidden order of test cases, and a compatibility issue with PIT and JUnit. We were unable to determine the cause and resolve this issue because the list only presents the most common causes, but there may be other reasons that cause PIT failure.

The mean value of the fault detection ability of STARTS is 0.43% less than that of the original test suite. Ekstazi and OpenClover achieved 0.10% and 0.47% less fault detection ability than STARTS. This is because STARTS is a safe static RTS technique, and it selects more test cases than dynamic RTS techniques. Safety violation shows a similar result. HyRTS, on the other hand, killed only

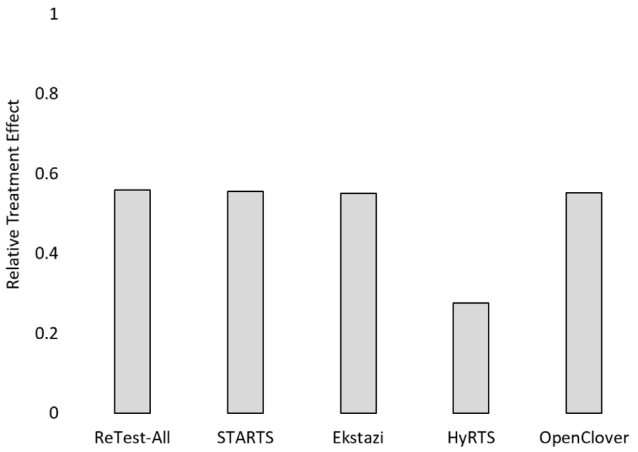

Fig. 10. Non-parametric test result for fault detection ability.

Table 6
Bonferroni test result for fault detection ability.

Comparison	Difference	p-value	Sig.
Ekstazi - HyRTS	32.60690	0.0000	***
Ekstazi - OpenClover	0.38621	1.0000	
Ekstazi - RetestAll	−0.52414	1.0000	
Ekstazi - STARTS	−0.08966	1.0000	
HyRTS - OpenClover	−32.22069	0.0000	***
HyRTS - RetestAll	−33.13103	0.0000	***
HyRTS - STARTS	−32.69655	0.0000	***
OpenClover - RetestAll	−0.91034	1.0000	
OpenClover - STARTS	−0.47586	1.0000	
RetestAll - STARTS	0.43448	1.0000	

12.25% of mutants. The safety violation of HyRTS is 16.62%, which is around two times higher than Ekstazi and OpenClover's safety violation with respect to STARTS. While STARTS, Ekstazi, and OpenClover did not kill any mutants on some of the revisions, we observed that HyRTS killed no mutants on the majority of revisions (80.82%). Unfortunately, HyRTS does not provide a function that determines which files changed, so we could not determine if the problem was a result of misidentifying changed files or finding test dependencies. Even though HyRTS computes smart checksums like Ekstazi and STARTS, Zhang (2018) states that Ekstazi was not open source at that time, so they implemented their own way to compute the smart checksum. HyRTS is not open source, so we could not inspect the code.

Fig. 10 shows the non-parametric test result with the fault detection ability scores obtained by running all the test cases (RetestAll) and the test cases selected by the four tools. The RTE value of the fault detection ability of STARTS is 0.0038 lower than the original test suite. With the exception of HyRTS, which had a significantly smaller RTE value, the other three techniques had comparable RTE values.

The Bonferroni test result (Table 6) confirmed HyRTS has a significantly lower fault detection ability compared to each of the other three techniques. This result is not surprising since Fig. 10 clearly shows HyRTS killed the smallest number of mutants.

A major reason for the low fault detection achieved by HyRTS is that it works at the method level. PIT mutations are performed by mutating files that were identified as changed, and the mutations were spread throughout the files. However, HyRTS selects test cases based on which methods are determined to have changed, and thus, it does not select test cases that cover unchanged methods, even though PIT may have mutated these methods. Thus, the results are only applicable to the other RTS tools.

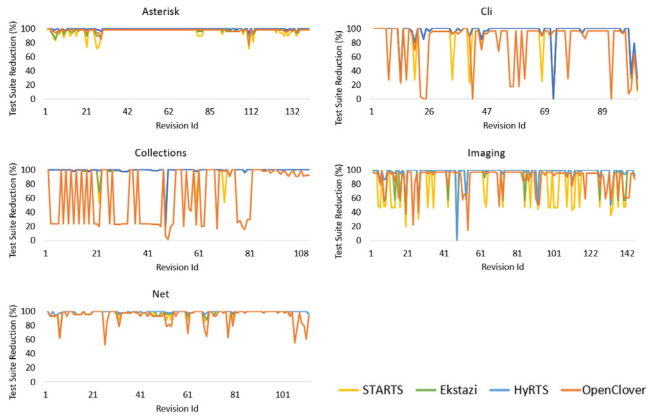

Fig. 11. Test suite reduction per subject.

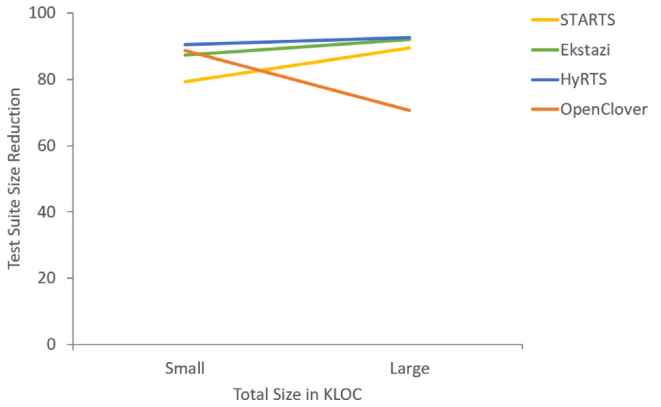

Fig. 12. Interaction between RTS technique and total size in KLOC on test suite size reduction.

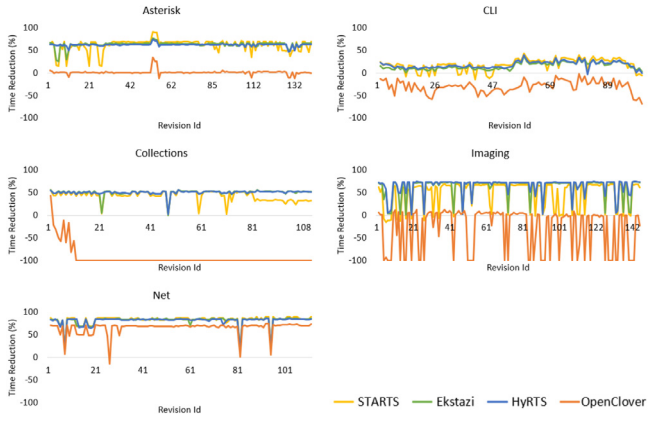

Fig. 13. End-to-end time reduction per subject.

5.5. Interaction effects between program characteristics and RTS techniques on performance metrics

In Section 4.1, we defined four program features that can potentially moderate the effects of RTS techniques on performance. These features are: total size in KLOC, total number of classes, percentage of test classes over the total number of classes, and the percentage of classes that changed between revisions. In this section, we analyze the relationship between those program features and RTS techniques with respect to their effect on three of the performance metrics by testing the following hypotheses (expressed in null form):

1. There is no significant interaction effect between RTS techniques and total size in KLOC on (a) test suite size reduction, (b) reduction in end-to-end time, and (c) fault detection ability.
2. There is no significant interaction effect between RTS techniques and total number of classes on (a) test suite size reduction, (b) reduction in end-to-end time, and (c) fault detection ability.
3. There is no significant interaction effect between RTS techniques and percentage of test classes in the total number of classes on (a) test suite size reduction, (b) reduction in end-to-end time, and (c) fault detection ability.
4. There is no significant interaction effect between RTS techniques and percentage of changed classes between revisions on (a) test suite size reduction, (b) reduction in end-to-end time, and (c) fault detection ability.

We do not analyze the safety and precision violation metrics in this section because the fault detection ability is related to safety violations, and the amount of test suite size reduction is related to precision violation (Engström et al., 2008). This section has four parts, one for each program feature. Note that to minimize redundancy, we only report the results of parametric test results.

5.5.1. Total size in KLOC

Test suite size reduction. Fig. 11 depicts the test suite size reduction for each subject. Ekstazi, STARTS, and HyRTS achieved higher test suite size reduction in projects with over 100 KLOC than in projects with less than 100 KLOC. The difference in reduction achieved in projects over 100 KLOC and less than 100 KLOC was biggest in STARTS (5.41%) and smallest in HyRTS (2.36%). Therefore, the lines that represent STARTS, Ekstazi, and HyRTS are barely visible for subjects that have over 100 KLOC (Asterisk and Commons Collections)

Note that the line graphs are stacked in the order of STARTS, Ekstazi, HyRTS, and OpenClover. Thus, if several techniques achieved the same reduction on a specific revision, only the last stacked line is visible on the graph. For example, revision 72 in Commons CLI and revision 48 in Commons Imaging appear to indicate HyRTS (blue line) is the only technique that achieved 0% test suite reduction, even though STARTS and Ekstazi also achieved comparable test suite size reductions.

The parametric between-subjects test for difference in test suite size reduction by total size in KLOC (i.e., small vs large programs) was not significant ($F = 0.08$, $df = 1$, $p = 0.776$). However, the interaction effect between total size in KLOC and RTS technique on test suite size reduction was significant ($F = 74.70$, $df = 1.37$, $p < 0.000$). Figure 5.12 shows the interaction effect graphically. STARTS, Ekstazi, and HyRTS achieved higher test suite size reduction in projects with over 100 KLOC than in smaller projects. In contrast, OpenClover selected more test cases on projects with over 100 KLOC and fewer cases on smaller projects.

End-to-end time reduction. Fig. 13 shows the time reduction for each subject. Ekstazi and STARTS reduce more time on programs with higher KLOC. Even though there is an exception for Commons Net, Ekstazi reduced 18.31% more time on the subject with the largest KLOC (Asterisk) than the smallest one (Commons CLI). Similarly, STARTS reduced 19.11% more time on the subject with the largest KLOC. Hence, the lines representing STARTS and Ekstazi are mostly above 50% for Asterisk and for some revisions, even below zero for CLI. .

The parametric between-subjects test for differences in the end-to-end time reduction by total size in KLOC was statistically

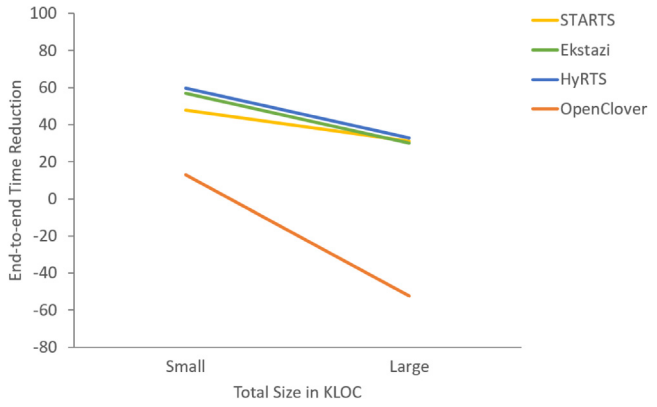

Fig. 14. Interaction between total size in KLOC and RTS technique on end-to-end time reduction.

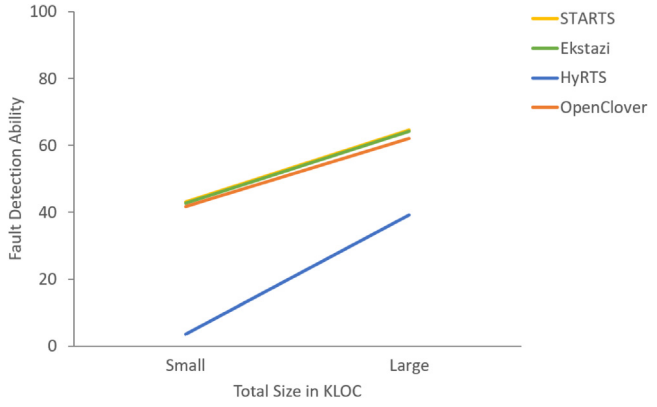

Fig. 15. Interaction between RTS technique and total size in KLOC on fault detection ability. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

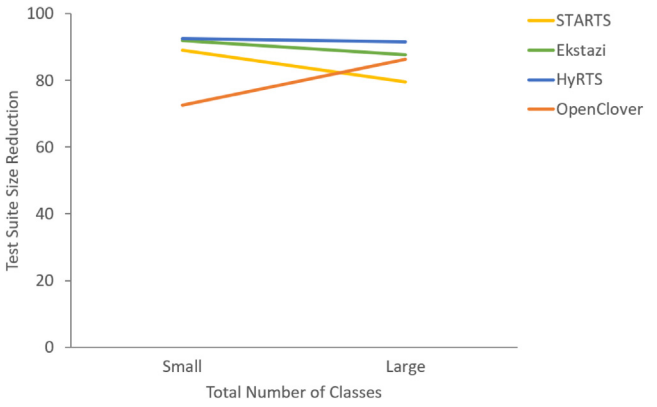

Fig. 16. Interaction between RTS technique and total number of classes on test suite size reduction.

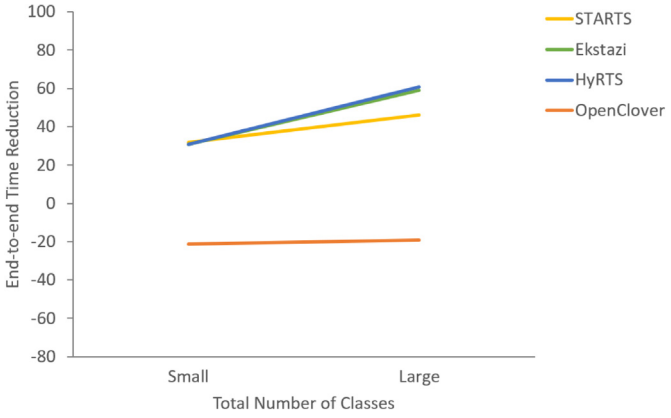

Fig. 17. Interaction between RTS technique and total number of classes on end-to-end time reduction. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

significant ($F = 342.590$, $df = 1$, $p < 0.000$). Further, the interaction effect between total size in KLOC and RTS technique on end-to-end time reduction was also significant ($F = 180.282$, $df = 1.45$, $p < 0.000$).

Fig. 14 shows the end-to-end time reductions achieved by the four RTS techniques for the two different sizes in KLOC. The overall pattern is similar in that RTS techniques tend to reduce more time on programs with fewer KLOC. However, compared to the other three techniques, the time reductions achieved by OpenClover is significantly lower. In addition, OpenClover's time reduction performance is also significantly worse on larger programs.

Fault detection ability. The parametric between-subjects test for differences in fault detection ability by total size in KLOC was statistically significant ($F = 14.676$, $df = 1$, $p < 0.000$), while the interaction effect between total size in KLOC and RTS Technique on fault detection ability was not significant ($F = 2.317$, $df = 1.09$, $p = 0.13$).

Fig. 15 illustrates the significant effect of program size on the faculty detection ability of each RTS technique. In general, the fault detection ability of RTS techniques is significantly better for larger programs. Compared to other techniques, the fault detection ability of HyRTS (blue line) appears to be much lower. However, for reasons discussed in Sections 4.4 and 5.4, the scores for HyRTS may not accurately reflect its fault detection capability. The yellow line for STARTS is not displayed well in Fig. 15 because the plot values of STARS and Ekstazi are almost identical.

5.5.2. Total number of classes

Test suite size reduction. The parametric between-subjects test for difference in test suite size reduction by total number of classes was not significant ($F = 0.20$, $df = 1$, $p = 0.65$), while the interaction effect between total number of classes and RTS technique on test suite size reduction was significant ($F = 42.32$, $df = 1.37$, $p < 0.000$).

The interaction effect of RTS technique and total number of classes on test suite size reduction is shown in Fig. 16. While STARTS, Ekstazi and HyRTS select more test cases for projects with larger number of classes, the pattern is reversed for OpenClover. It selects more test cases for projects with smaller number of classes and fewer test cases for projects with larger number of classes.

Even though both total size in KLOC and total number of classes are indicators of project size, the interaction effect between each of these two measures and RTS technique on test suite reduction size show different patterns (Figs. 12 and 16). This difference underscores the fact that the number of classes and lines of code in a project may not always be positively correlated.

End-to-end time reduction. The parametric between-subjects test for difference in end-to-end time reduction by total number of classes was significant ($F = 96.60$, $df = 1$, $p < 0.000$). The interaction effect between total number of classes and RTS technique on end-to-end time reduction was also significant ($F = 69.291$, $df = 1.45$, $p < 0.000$).

Fig. 17 shows the interaction between RTS technique and total number of classes on end-to-end time reduction. The most

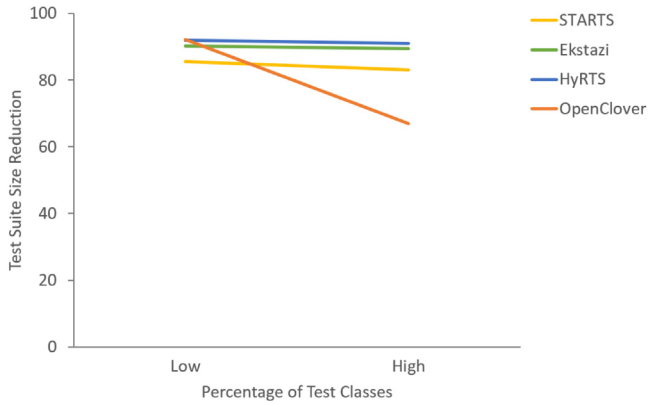

Fig. 18. Interaction between RTS technique and percentage of test classes on test suite size reduction.

noticeable difference between techniques is the orange line that shows OpenClover achieved significantly lower end-to-end time reduction than the other techniques. Further, while the time reductions for OpenClover are similar for both small and large projects in terms of total number of classes, the time reductions achieved by STARTS, Ekstazi and HyRTS are significantly higher for larger projects.

Fault detection ability. The parametric between-subjects test for differences in fault detection ability by total number of classes was significant ($F = 47.68$, $df = 1$, $p < 0.000$). However, the interaction effect between total number of classes and RTS technique on fault detection ability was not significant ($F = 1.29$, $df = 1.09$, $p = 0.26$).

5.5.3. Percentage of test classes in the total number of classes

Test suite size reduction. The main effect of the percentage of test classes in the total number of classes and the interaction effect between the percentage of test classes and RTS technique on test suite size reduction were statistically significant. The parametric between-subjects test for difference in test suite size reduction by the percentage of test classes was significant ($F = 66.88$, $df = 1$, $p < 0.000$). Further, the interaction effect between the percentage of test classes and RTS technique on test suite size reduction was also significant ($F = 97.23$, $df = 1.37$, $p < 0.000$).

The interaction effect between RTS technique and percentage of test classes on test suite size reduction is presented in Fig. 18. The figure shows that test suite reductions achieved by STARTS, Ekztazi and HyRTS are relatively similar between projects with low and high percentage of test classes. In contrast, OpenClover has significantly lower test suite size reduction for projects that had a higher percentage of test classes.

End-to-end time reduction. The parametric between-subjects test for differences in end-to-end time reduction by the percentage of test classes was significant ($F = 1208.88$, $df = 1$, $p < 0.000$). The interaction effect between the percentage of test classes and RTS technique on end-to-end time reduction was also significant ($F = 497.723$, $df = 1.45$, < 0.000).

Fig. 19 shows the interaction between RTS technique and percentage of test classes in the total number of classes on end-to-end time reduction. The overall trend is similar across the four RTS techniques: higher time reduction in projects with a lower percentage of test cases than projects with a higher percentage of test cases. This pattern is magnified for OpenClover, which shows a dramatic drop in performance on end-to-end time reduction for projects with a higher percentage of test classes.

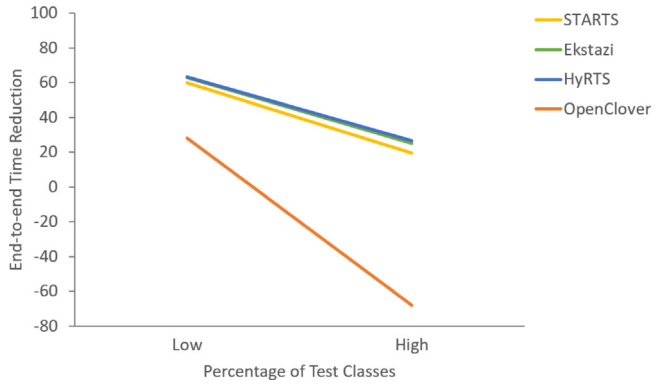

Fig. 19. Interaction between RTS technique and percentage of test classes on end-to-end time reduction.

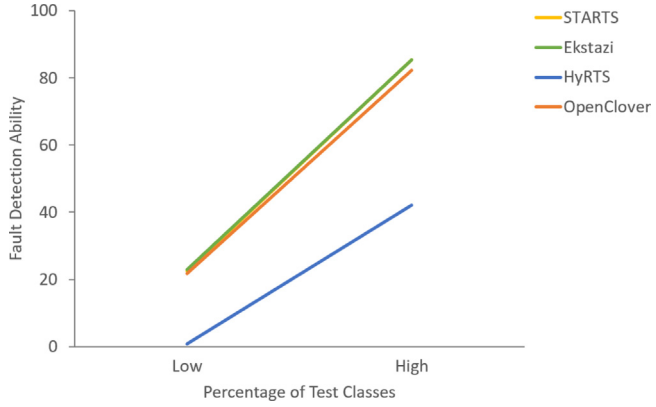

Fig. 20. Interaction between RTS technique and percentage of test classes on fault detection ability.

Fault detection ability. Both the main effect of the percentage of test classes and the interaction effect between the percentage of test classes and RTS technique on fault detection ability were statistically significant. The parametric between-subjects for difference in fault detection ability by the percentage of test classes in the total number of classes was significant ($F = 98.38$, $df = 1$, $p < 0.000$). The interaction effect between the percentage of test classes and RTS technique on the fault detection ability was significant ($F = 6.19$, $df = 1.09$, $p = 0.01$).

Fig. 20 depicts the interaction between RTS technique and percentage of test classes on fault detection ability. There is a similar pattern of fault detection ability among RTS techniques. Specifically, the techniques perform better in projects with a higher percentage of test classes.

5.5.4. Percentage of changed classes between revisions

Fig. 21 is a distribution chart showing the percentage of revisions based on the percentage of files changed in each revision. The categorization is used as a factor to observe the relationship between the percentage of changed files and performance of RTS techniques.

First, we considered revisions where no files changed (category C_1) and those where at least one file changed. This differentiation is needed because RTS techniques should not select any test case when no files are changed. Then, we analyzed the distribution of revisions where one or more files were changed. 28.35% of revisions (category C_2) have less than 1% changed files, and was the next highest percentage after the category where no files changed. The revisions with multiple changed files are grouped in one category called C_3. We did not divide the revisions

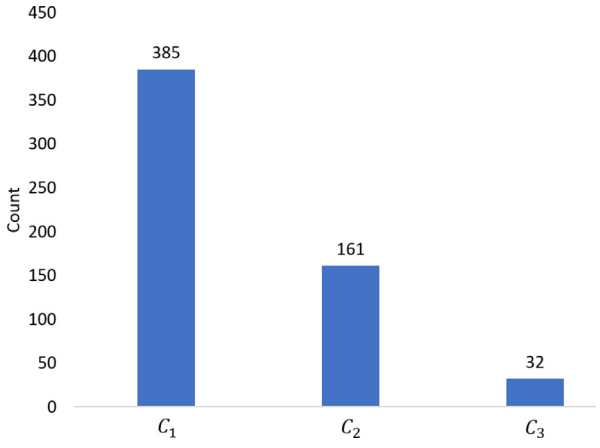

Fig. 21. Distribution of changes in subjects.

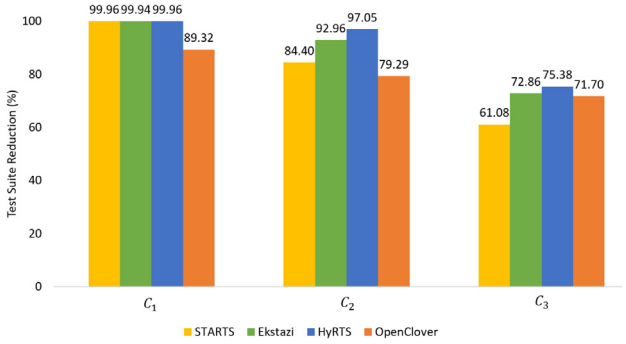

Fig. 22. Number of changed files and test suite reduction.

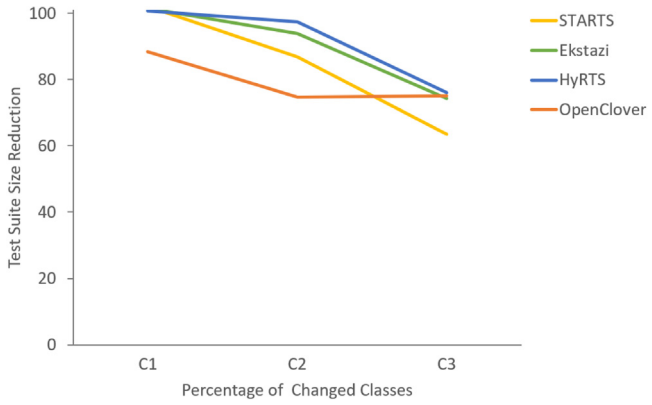

Fig. 23. Interaction between RTS technique and percentage of changed classes in revisions on test suite reduction.

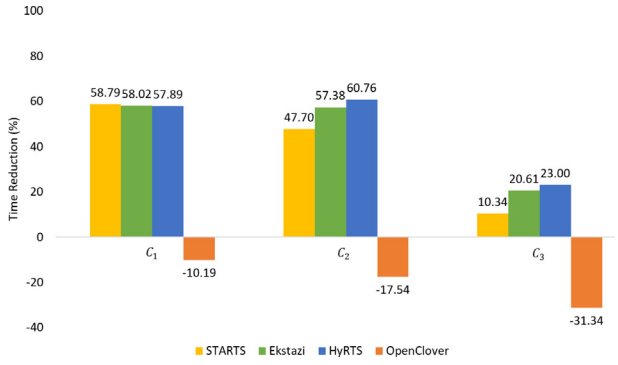

Fig. 24. Number of changed files and time reduction.

in the category C_3 further because C_3 has only 5% of the total revisions. Statistically, the sample size in C_3 is already quite small.

Test suite size reduction. Fig. 22 shows the relationship between test suite reduction and percentage of changes over the total number of files. Category C_1 includes revisions with no file changes, and the bar chart shows that the test suite reductions achieved by all four RTS techniques are not 100% in this category. This is because the RTS techniques used in our study are designed to select test cases that are (1) relevant to the code changes and (2) the test cases that were newly added in the revision. In category C_1, OpenClover selected and ran 10.63% more test cases that were ignored by STARTS, Ekstazi, and HyRTS. We observed that revisions, such as revision 46 in Commons CLI, revision 73 in Commons Collections, and multiple revisions in Commons Imaging, do not have changed files, but the total number of test cases increased. This means new test cases were added to those revisions. We confirmed that all four RTS techniques selected test cases on those revisions.

Overall, RTS techniques run more test cases when there are more changed class files. Compared to C_1, STARTS selected 15.56% more test cases in the category C_2, while Ekstazi and HyRTS select 6.98% and 2.91% more test cases. OpenClover's performance in reducing test cases is comparable to that of other techniques in revisions with more changed files. In category C_3 where there are multiple changed files, OpenClover actually reduces 10.62% more test cases than STARTS.

We observed that test suite reduction is also affected by the type of changed files in addition to the number of changed files. This means RTS techniques do not necessarily select fewer test cases because fewer files are changed. For example, revision 99 in Commons CLI has one changed file adding annotations (override and deprecated) on the existing methods, and 73.21% test cases are selected on average for all RTS techniques.

The parametric between-subjects test for difference in test suite reduction by the percentage of changed classes was significant ($F = 161.37$, $df = 2$, $p < 0.000$). The interaction effect between the percentage of changed classes and RTS technique on test suite size reduction was also significant ($F = 23.03$, $df = 2.73$, $p < 0.000$).

Fig. 23 shows the interaction between the percentage of changed files between revisions and RTS technique on test suite size reduction. The pattern in this figure is similar to that of the bar chart in Fig. 22 – the number of test cases selected by RTS techniques increases as the number of changed class files rises. On revisions with no changed files, STARTS, Ekstazi and HyRTS achieved similar test suite size reductions, while OpenClover's performance is significantly lower. In contrast, OpenClover achieved higher test suite size reduction than STARTS and its performance is comparable to Ekstazi and HyRTS in category C3.

End-to-end time reduction. In Fig. 24, we show how much time was reduced compared to running the original test suite when there are different number of changes. HyRTS and OpenClover achieved the highest and lowest time reductions, respectively. As expected, RTS tools save more time when there are few file changes because they select and run more test cases when there are more changed files and impacted test cases.

The parametric between-subjects test for difference in the end-to-end time reduction by the percentage of changed classes was significant ($F = 72.48$, $df = 2$, $p < 0.000$). The interaction effect between the percentage of changed classes and RTS technique on the end-to-end time reduction was also significant ($F = 19.31$, $df = 2.89$, $p < 0.000$).

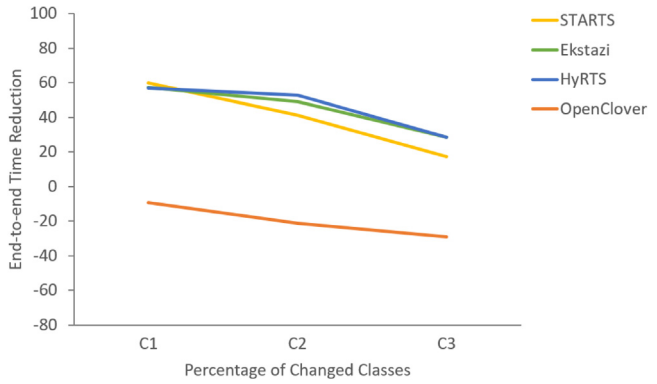

Fig. 25. Interaction between RTS technique and percentage of changed classes in revisions on time reduction.

The overall pattern of end-to-end time reductions shown in Fig. 25 is similar to the test suite size reductions displayed in Fig. 23. The figure also shows that OpenClover achieved significantly lower time reductions than the other techniques in all three categories.

Fault detection ability. The parametric between-subjects test for difference in the fault detection ability by the percentage of changed classes was significant ($F = 7.01$, $df = 1$, $p = 0.01$). However, the interaction effect between the percentage of changed classes and RTS technique on fault detection ability was not significant ($F = 1.54$, $df = 1.09$, $p = 0.22$). These results indicate that while there were differences in fault detection ability among the categories of changed classes, these differences were consistent across the four RTS techniques.

5.6. Discussion

Our results are in agreement with results from studies conducted by other researchers (Zhang, 2018; Zhu et al., 2019). Zhang (2018) shows HyRTS achieved higher test suite size reduction and time reduction than Ekstazi on average in their empirical study. The study conducted by Zhu et al. (2019) demonstrates that not every RTS technique saves time. Our empirical results show that sometimes OpenClover takes longer than running the original test suite, and the study conducted by Zhu et al. (2019) demonstrates similar results that using OpenClover often takes longer than the original test suite. Overall, HyRTS selects the least number of test cases among the four techniques (Section 5.1), and the highest mean value of end-to-end time reduction is achieved by HyRTS at 59.40% (Section 5.2). OpenClover spent 93.63% more time than running the original test suite of the Commons Collection subject (Section 5.5.2).

We computed safety and precision violations with respect to STARS, Ekstazi, and HyRTS. We used the three techniques as baselines for the following reasons. When two techniques select the same number of test cases, it is difficult to know whether those test cases are the same ones selected by both techniques. For example, out of 10 test cases, if techniques A and B select one test case each, x and y respectively, then both techniques appear to have the same test suite size reduction ability of 90%. However, a comparison of safety and precision violations for each technique shows techniques A and B have selected different test cases. Therefore, RTS techniques should be compared based on their (1) dependency analysis type (e.g., dynamic, static) and (2) dependency determination level (e.g., method, class, file). Both STARTS and Ekstazi determine dependencies at class level, while STARTS is a static technique and Ekstazi is a dynamic technique. The dynamic technique may be more precise than the static technique, and vice versa. Thus, STARTS and Ekstazi were selected as a baseline. On the other hand, HyRTS is also used as a baseline because it is the only method-level RTS technique among the four techniques. HyRTS is particularly the best choice for safety violation baseline because file-level techniques may select more test cases than necessary.

5.7. Threats to validity

5.7.1. Internal validity

These threats are related to the implementations used in the study. We followed the manuals researchers provide on their official websites. The process of conducting the empirical study was fully automated and carefully reviewed. We investigated why OpenClover selects 15% more test cases than the other tools. We could not inspect all the changes, but we manually inspected some of the suspicious revisions and found two reasons why OpenClover selects more test cases. First, computing smart checksums misses the identification of some changed files and this can potentially change debug information. We found that STARTS, Ekstazi, and HyRTS do not identify some of the code changes (e.g., static enum A to enum A). OpenClover has a known bug (Zhu et al., 2019) that selects any test case with certain annotations. Zhu et al. (2019) show that OpenClover has several safety issues in finding test dependencies.

A major threat to internal validity is that we mutated entire files using PIT as long as the file was identified by the tools as being modified. This type of mutation is unfair to HyRTS because it selects test cases using method-level information.

5.7.2. External validity

Our findings may not be generalizable to all programs. To reduce this threat, we used a diverse set of programs in terms of the number of test cases and the number of lines of code. We selected version numbers not used in other studies to avoid bias.

The fault detection results may be different if we used other mutation tools. Moreover, instead of using mutation faults, one could use real faults that were reported for the revisions. Thus, the fault detection results may not generalize to real faults. However, studies (Andrews et al., 2005) have shown the usefulness of mutation faults in software engineering experiments.

5.7.3. Construct validity

The use of end-to-end test time can be a threat to construct validity. Other metrics can be used to measure time reduction, such as test case selection time and test execution time. We used end-to-end test time because Ekstazi, HyRTS, and OpenClover only output aggregated test time data. We also recognize that having a gold standard is necessary for correctly using safety and precision violation formulas. If the tools used as baselines are not safe (or precise), the results can be misleading.

6. Conclusions

Our research goal was to compare four recent Java-based RTS techniques in terms of the amount of test suite size reduction, end-to-end time reduction, safety and precision violations, and fault detection ability. We also aimed to investigate program features that affect the performance of RTS techniques, that can help practitioners determine the most appropriate technique for their specific requirements. To achieve these goals, we ran RTS techniques on multiple revisions of five open source projects and analyzed the results using statistical tests to answer our research questions. The statistical tests helped to (a) verify the significance of visually observed differences in the evaluation metrics among

the RTS techniques and (b) explore interaction effects between the techniques and program features.

We found that average test suite size reduction ranges from 86.14% to 98.13%. The test suite size reductions achieved by HyRTS and Ekstazi are statistically similar, while OpenClover selects significantly more test cases than the other RTS techniques. Sometimes, the RTS techniques take a longer time than running the original test suite, but the average end-to-end time reduction of the four RTS techniques was 40.49%. Using HyRTS as the baseline, OpenClover had significantly worse safety violations compared to STARTS and Ekstazi, and significantly worse precision violations compared to Ekstazi. In addition, while STARTS and Ekstazi did not differ on safety violations, Ekstazi had significantly fewer precision violations than STARTS.

STARTS, Ekstazi, and OpenClover killed as many mutants as running the original test suite. However, this is due to the fact that we mutated all the methods inside a modified file irrespective of whether a method was modified or not. HyRTS selects test cases only on the basis of whether they executed changed methods, and thus, would not select test cases that executed unchanged methods, leading to a low mutation score.

We chose total program size in KLOC, total number of classes, percentage of test classes in the total number of classes, and percentage of changed classes as program characteristics and explored their influence on the performance of RTS techniques. We found OpenClover has an opposite pattern from other techniques with respect to test suite size reduction. For example, STARTS, EKstazi, and HyRTS achieved higher test suite size reduction in subjects with over 100 KLOC than in those with less than 100 KLOC, while OpenClover selected more test cases in subjects with over 100 KLOC. In general, OpenClover selected more test cases than other techniques but it selected fewer test cases on the programs with fewer test classes.

The statistical test results offered insights that are not necessarily apparent in the plots. For example, since there are many outliers in safety violations, it is not easy to compare the techniques on this metric. By conducting the statistical tests, we found that HyRTS achieved higher safety violations than Ekstazi and OpenClover with respect to STARTS while STARTS, Ekstazi, and OpenClover achieved statistically similar safety violations.

In conclusion, Ekstazi performed the best in all the metrics among the four techniques, especially when the program size is over 100 KLOC. OpenClover should be avoided if reduction of end-to-end time is a goal.

7. Future work

Future work could evaluate other Java-based techniques to derive a clear conclusion regarding static versus dynamic techniques. Furthermore, several machine learning-based RTS techniques have been developed recently that emphasize the selection of fewer test cases as the main objective rather than considering the safety of test selection (Shi et al., 2019). Thus, comparing the currently widely used techniques with machine learning-based RTS techniques will be useful. Empirical evaluations can be performed for tools across different programming languages.

Another possible extension is to use more subjects and revisions to achieve greater generalizability. Comparing RTS techniques with real industrial programs will be also useful.

During the study, we found that RTS techniques have unexpected compatibility issues with some open-source projects. Also, one or more RTS techniques had build failures on several revisions even though the original test suite ran successfully. Future work can measure the generality of RTS techniques.

We selected four program features to examine their impact on the performance of RTS techniques. There are additional features that could be worth investigating. For example, our results show that the type of changes affects test suite size reduction. Therefore, analyzing the performance of RTS techniques based on the type of changes in revisions, such as adding new test cases, adding new parameters, modifying conditions could reveal interesting results.

The fault detection study can be done using actual faults reported by users and developers instead of mutation faults. Moreover, the faults (real or mutation) can be selectively seeded in methods that were changed rather than inside the whole class.

CRediT authorship contribution statement

Min Kyung Shin: Software, Data curation, Visualization, Writing - original draft. **Sudipto Ghosh:** Conceptualization, Methodology, Writing - review & editing. **Leo R. Vijayasarathy:** Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported in part by funding from NSF under Award Number OAC 1931363.

References

Akritas, M.G., Brunner, E., 1997. A unified approach to rank tests for mixed models. J. Statist. Plann. Inference 61 (2), 249–277.
Al-Refai, M.N., 2019. Towards Model-Based Regression Test Selection (Ph.D. thesis). Colorado State University. Libraries.
Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: 27th International Conference on Soft. Engr.. ACM, pp. 402–411. http://dx.doi.org/10.1145/1062455.1062530.
Anon, 2019a. Clover 4 test optimization, https://confluence.atlassian.com/clover/about-test-optimization-169119919.html/. (Accessed 19 April 2019).
Anon, 2019b. Open Clover User Guide, http://openclover.org/doc/manual/4.2.0/ant--coverage-recorders.html. (Accessed 3 June 2019).
Anon, 2019c. OpenClover, https://openclover.org/. (Accessed 19 April 2019).
Anon, 2019d. PITest, https://pitest.org/. (Accessed 8 July 2019).
Azizi, M., Do, H., 2018. ReTEST: A cost effective test case selection technique for modern software development. In: 29th International Symposium on Software Reliability Engineering. IEEE, pp. 144–154.
Baguley, T., 2004. An Introduction to Sphericity, Vol. 1. p. 2008, September.
Bible, J., Rothermel, G., Rosenblum, D.S., 2001. A comparative study of coarse-and fine-grained safe regression test-selection techniques. ACM Trans. Softw. Eng. Methodol. 10 (2), 149–183.
Chan, Y., 2003. Biostatistics 102: Quantitative data-parametric & non-parametric tests. Blood Press 140 (24.08), 79.
Chittimalli, P.K., Harrold, M.J., 2008. Regression Test Selection on System Requirements. In: 1st India Software Engr Conf., pp. 87–96.
Engström, E., Runeson, P., Skoglund, M., 2010. A systematic review on regression test selection techniques. Inf. Softw. Technol. 52 (1), 14–30.
Engström, E., Skoglund, M., Runeson, P., 2008. Empirical evaluations of regression test selection techniques: A systematic review. In: 2nd ACM-IEEE International Symposium on Empirical Softw. Eng. and Measurement, pp. 22–31.
Ghosh, A., Mandal, A., Martín, N., Pardo, L., 2016. Influence analysis of robust Wald-type tests. J. Multivariate Anal. 147, 102–126.
Gligoric, M., Eloussi, L., Marinov, D., 2015. Practical regression test selection with dynamic file dependencies. In: International Symposium on Software Testing and Analysis, pp. 211–222.
Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A., Rothermel, G., 2001. An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Methodol. 10 (2), 184–208.
Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A., 2001a. Regression test selection for Java software. ACM Sigplan Not. 36 (11), 312–326.
Harrold, M.J., Jones, J.A., Rothermel, G., 1998. Empirical studies of control dependence graph size for c programs. Empirical Softw. Eng. 3 (2), 203–211.

Harrold, M.J., Rosenblum, D., Rothermel, G., Weyuker, E., 2001b. Empirical studies of a prediction model for regression test selection. IEEE Trans. Softw. Eng. 27 (3), 248–263.
Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow-and controlflow-based test adequacy criteria. In: 16th International Conference on Soft. Engr.. IEEE, pp. 191–200.
Jia, Y., Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.
Kazmi, R., Jawawi, D.N., Mohamad, R., Ghani, I., 2017. Effective regression test case selection: A systematic literature review. ACM Comput. Surv. 50 (2), 1–32.
Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Le Traon, Y., 2018. How effective are mutation testing tools? An empirical analysis of java mutation testing tools with manual analysis and real faults. Empirical Softw. Eng. 23 (4), 2426–2463.
Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D., 2016. An extensive study of static regression test selection in modern software evolution. In: 24th ACM SIGSOFT International Symposium on Foundations of Soft. Engr., pp. 583–594.
Legunsen, O., Shi, A., Marinov, D., 2017. STARTS: Static regression test selection. In: 32nd International Conference on Automated Soft. Engr.. ASE, IEEE, pp. 949–954.
Lin, F., Ruth, M., Tu, S., 2006. Applying safe regression test selection techniques to java web services. In: International Conference on Next Generation Web Services Practices. IEEE, pp. 133–142.
Ludbrook, J., 1998. Multiple comparison procedures updated. Clin. Exp. Pharmacol. Physiol. 25 (12), 1032–1037.
Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J., 2017. Taming google-scale continuous testing. In: 39th International Conference on Soft. Engr.: Soft. Engr. in Practice Track. IEEE, pp. 233–242.
Noguchi, K., Gel, Y.R., Brunner, E., Konietschke, F., 2012. NparLD: An R software package for the nonparametric analysis of longitudinal data in factorial experiments. J. Stat. Softw. 50 (12).
Orso, A., Shi, N., Harrold, M.J., 2004. Scaling regression testing to large software systems. ACM SIGSOFT Softw. Eng. Notes 29 (6), 241–251.
Romano, S., Scanniello, G., Antoniol, G., Marchetto, A., 2018. SPIRITuS: A SimPle information retrieval regression test selection approach. Inf. Softw. Technol. 99, 62–80.
Rosenblum, D., Rothermel, G., 1997. A comparative study of regression test selection techniques. In: 2nd International Workshop on Empirical Studies of Software Maintenance. IEEE Computer Society Press.
Rothermel, G., Harrold, M.J., 1996. Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. 22 (8), 529–551.
Rothermel, G., Harrold, M.J., 1997a. Experience with regression test selection. Empirical Softw. Eng. 2 (2), 178–188.
Rothermel, G., Harrold, M.J., 1997b. A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. 6 (2), 173–210.
Rothermel, G., Harrold, M.J., Dedhia, J., 2000. Regression test selection for C++ software. Softw. Test. Verif. Reliab. 10 (2), 77–109.
Shi, A., Gyori, A., Gligoric, M., Zaytsev, A., Marinov, D., 2014. Balancing trade-offs in test-suite reduction. In: 22nd ACM SIGSOFT International Symposium on Foundations of Soft. Engr., pp. 246–256.
Shi, A., Hadzi-Tanovic, M., Zhang, L., Marinov, D., Legunsen, O., 2019. Reflection-aware static regression test selection. In: Conference on Object-Oriented Programming, Systems, Languages, and Application, Vol. 3. ACM New York, NY, USA, pp. 1–29.
Soetens, Q.D., Demeyer, S., 2012. ChEOPSJ: Change-based test optimization. In: Euromicro Conference on Software Maintenance and Reengineering, pp. 535–538.
Soetens, Q.D., Demeyer, S., Zaidman, A., 2013. Change-based test selection in the presence of developer tests. In: 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 101–110.
Soetens, Q.D., Demeyer, S., Zaidman, A., Pérez, J., 2016. Change-based test selection: An empirical evaluation. Empirical Softw. Eng. 21 (5), 1990–2032.
Tang, M.-H., Kao, M.-H., Chen, M.-H., 1999. An empirical study on object-oriented metrics. In: International Software Metrics Symposium. IEEE, pp. 242–249.
Vokolos, F.I., Frankl, P.G., 1997. Pythia: A regression test selection tool based on textual differencing. In: Reliability, Quality and Safety of Software-Intensive Systems. Springer, pp. 3–21.
Xu, G., Rountev, A., 2007. Regression test selection for aspectj software. In: 29th International Conference on Soft. Engr.. ICSE'07, IEEE, pp. 65–74.
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.
Zhang, L., 2018. Hybrid regression test selection. In: 40th International Conference on Soft. Engr.. ICSE, IEEE, pp. 199–209.
Zhu, C., Legunsen, O., Shi, A., Gligoric, M., 2019. A framework for checking regression test selection tools. In: 41st International Conference on Soft. Engr.. ICSE, IEEE, pp. 430–441.

Min Kyung (Erica) Shin received her MS degree in Computer Science from Colorado State University in 2020. She received her Bachelor's degree in Computer Engineering from Hansung University in 2015. Her research interests are in Software Engineering.

Sudipto Ghosh is a Professor of Computer Science at Colorado State University, USA. He received the Ph.D. degree in Computer Science from Purdue University, USA, in 2000. His teaching and research interests include model-based software development and software testing. He is on the editorial boards of IEEE Transactions on Reliability, Information and Software Technology, and Software Quality Journal. More information about Dr. Ghosh and all his publications are available from his web page at http://www.cs.colostate.edu/~ghosh, and he can be contacted at ghosh@colostate.edu for any question.

Leo R. Vijayasarathy is Professor and Department Chair of Computer Information Systems at Colorado State University. He earned an MBA from Marquette University and his Ph.D. from Florida International University. His research focuses on the development, use, and consequences of information systems. He serves on the editorial advisory board of Internet Research. More information about Dr. Vijayasarathy and his publications are available at https://biz.colostate.edu/about/directory/colostate-vijayasa, and he can be contacted at leo.vijayasarathy@colostate.edu.