THESIS

# AN EMPIRICAL COMPARISON OF FOUR JAVA-BASED REGRESSION TEST SELECTION TECHNIQUES

Submitted by

Min Kyung Shin

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2020

## ABSTRACT

## AN EMPIRICAL COMPARISON OF FOUR JAVA-BASED REGRESSION TEST SELECTION TECHNIQUES

Regression testing is crucial to ensure that previously tested functionality is not broken by additions, modifications, and deletions to the program code. Since regression testing is an expensive process, researchers have developed regression test selection (RTS) techniques, which select and execute only those test cases that are impacted by the code changes.

In general, an RTS technique has two main activities, which are (1) determining dependencies between the source code and test cases, and (2) identifying the code changes. Different approaches exist in the research literature to compute dependencies statically or dynamically at different levels of granularity. Also, code changes can be identified at different levels of granularity using different techniques. As a result, RTS techniques possess different characteristics related to the amount of reduction in the test suite size, time to select and run the test cases, test selection accuracy, and fault detection ability of the selected subset of test cases. Researchers have empirically evaluated the RTS techniques, but the evaluations were generally conducted using different experimental settings.

This thesis compares four recent Java-based RTS techniques, Ekstazi, HyRTS, OpenClover, and STARTS, with respect to the above-mentioned characteristics using multiple revisions from five open source projects. It investigates the relationship between four program features and the performance of RTS techniques: total (program and test suite) 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.

The results show that STARTS, a static RTS technique, over-estimates dependencies between test cases and program code, and thus, selects more test cases than the dynamic RTS techniques

Ekstazi and HyRTS, even though all three identify code changes in the same way. OpenClover identifies code changes differently from Ekstazi, HyRTS, and STARTS, and selects more test cases. STARTS achieved the lowest safety violation with respect to Ekstazi, and HyRTS achieved the lowest precision violation with respect to both STARTS and Ekstazi. Overall, the average fault detection ability of the RTS techniques was 8.75% lower than that of the original test suite.

STARTS, Ekstazi, and HyRTS achieved higher test suite size reduction on the projects with over 100 KLOC than those with less than 100 KLOC. OpenClover achieved a higher test suite size reduction in the subjects that had a fewer total number of classes. The time reduction of OpenClover is affected by the combination of the number of source classes and the number of test cases in the subjects. The higher the number of test cases and source classes, the lower the time reduction.

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sudipto Ghosh, for his supervision and support. I was lucky and glad to meet him as my advisor, who is professional and gives me valuable guidance throughout the entire process of this master's thesis.

I would like to thank Dr. Laura Moreno Cubillos and Dr. Leo R. Vijayasarathy for agreeing to be members of my thesis committee. The feedback I got through the software engineering group meetings was really useful.

I would like to thank my parents and brother who always believed in me and my abilities. Without their help, love, encouragement, and financial support, I would not even have been able to start this journey.

I would thank my husband for continuously encouraging me and waiting for this long process to complete.

This research is funded in part through an award (OAC-1931363) from the National Science Foundation.

## DEDICATION

*To my family.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# Introduction

Regression testing is an essential process used in software development to verify that code changes do not break previously tested functionality. However, as the size of software keeps growing, the number of test cases and the time taken to perform testing also increase. In 2017, Google had 2 billion lines of code in its source code repository, and developers made 16,000 commits and ran 150 million test executions a day [1]. Clearly, it would be time-consuming to run every test case each time the code is revised.

Since regression testing is expensive, researchers have developed techniques, such as test prioritization, test minimization, and regression test selection [2] to reduce the cost. Test prioritization reorders test cases based on criteria such as detection of test failures as early as possible, and executes test cases for fault-prone modules earlier than others. Test minimization selects the minimal set of test cases that achieve the same test coverage as the original set of test cases. However, test prioritization and test minimization have limitations. Test prioritization eventually runs all the test cases, so there may be no reduction in test execution time [3]. Rothermel et al. [4] present empirical evidence that test minimization can cause a loss of fault detection ability. Regression test selection (RTS) identifies code changes that occur between revisions, and selects only those test cases that are impacted by the changes. An RTS technique is considered to be safe if it does not miss any test cases that should be selected; it is considered to be precise if it selects only those test cases that are impacted. Rothermel and Harrold [5] state that safe RTS techniques will not miss any fault revealing test case in the original test suite.

A typical RTS technique requires two main activities: (1) computing the dependencies between the test cases and program code, and (2) identifying the code changes between revisions. Depending on the RTS technique, test dependencies can be collected statically [6–8] or dynamically [9–11]. Additionally, dependencies can be analyzed at different levels of granularity such as statement, method, and class. 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)) [6, 9–13]. Code changes can also be identified at different granularity levels (e.g., statement, method, and class).

## 1.1 Motivation

Researchers have developed many RTS techniques. In general, there are five main characteristics: test suite reduction, time reduction, safety, precision, and fault detection ability of the selected test suite. First, RTS techniques can achieve different amounts of test size reduction. Because some RTS techniques can over-estimate the dependencies between the test cases and the elements of the code to ensure that no impacted test cases are missing to achieve safety, they can sometimes select more test cases than precise RTS techniques [7]. For example, only one method may have changed inside a file, but an RTS technique that uses a file as a unit of change may select all the test cases that execute any method in the file.

Second, RTS techniques offer different savings in end-to-end testing time. This is the total time that includes test execution time and any time that the RTS technique spends before and after test execution. For example, a method-level RTS technique may spend more time on dependency analysis and test selection than a class-level RTS technique [9]. However, a method-level RTS technique can reduce test execution time by running fewer test cases than a class-level RTS technique because using a finer granularity level can select test cases more precisely than using a coarser granularity level.

Third, RTS techniques have different test selection accuracy, which are characterized by their safety and precision. Depending on how it computes dependencies and identifies code changes, an RTS technique can have varying safety and precision. Ideally, an RTS technique should balance safety and precision such that the technique does not miss test cases that would reveal faults or waste time running too many unnecessary test cases [14].

Lastly, the test cases selected by RTS techniques differ in their ability to detect faults in the code. Ideally, the selected test cases should find as many faults as the original test suite. However, the fault detection ability can differ depending on the safety of the RTS technique.

Researchers have reported the results of several 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 [15]. Generally four metrics are used to measure cost reduction: test suite reduction, test execution time reduction, 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). Computing safety is a relative way to measure fault detection ability because test suites selected by safe techniques should find as many faults as running the original test suite. Finding real or seeded faults in a program is a direct way and provides an absolute measurement of fault detection effectiveness. Over time, the metrics have evolved. Rosenblum and Rothermel [16] used code coverage to compare the precision of RTS techniques. To compute safety, Graves et al. [17] measured how many test cases among the set of test cases selected by an RTS technique found seeded faults. Relatively recent studies [7,8,18] compute precision violation in terms of how many more test cases did an RTS technique select than the current best technique. They compute safety violation in terms of how many fewer test cases did an RTS technique select than the current best technique. Often it is not possible to provide an analytical argument to show that an RTS technique and its implementation are safe and precise. Since it is generally impossible to manually determine which test cases must be selected for a specific revision of software in an experiment, it is difficult to compute the safety and precision of an RTS technique. Thus, safety and precision violations with respect to another RTS technique can be employed as alternative metrics. Rothermel et al. [19] and Graves et al. [17] manually seed faults in a program, while recent studies [18,20] conduct mutation testing to compute the fault detection ability.

To compare RTS techniques, measuring both cost reduction and fault detection ability is vital because the techniques should reduce the amount of testing time and at the same time, be reasonably safe and not lose fault detection ability. However, not many RTS empirical evaluations consider both metrics. Engström et al.'s survey [15] shows that there are 38 empirical studies out of 923 papers, and only 30% of those studies measure both cost reduction and fault detection ability. The rest use only one of the metrics.

Since Java has become one of the most widely-used programming languages, many Java-based RTS techniques have been proposed [6–10]. The empirical evaluations reported in these papers show the differences between the newly proposed technique and the state-of-the-art at the time the papers were written. The studies are conducted using different subjects, program versions, test environments and different metrics. Most studies [6, 7, 10] 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 did not necessarily compare several techniques together using the same experimental setup.

## 1.2 Research Questions

This thesis aims to evaluate four well-known Java-based RTS techniques: Ekstazi, HyRTS, OpenClover, and STARTS in terms of amount of test size reduction, end-to-end time reduction, safety and precision violations, and fault detection ability to answer the following research questions:

**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?

## 1.3 Contributions

Many RTS techniques have been proposed in the research literature, but there is a lack of systematic empirical comparisons. The main contribution of the thesis is an evaluation and comparison of four recent and widely used RTS techniques using the same experimental conditions. We ran Ekstazi, HyRTS, OpenClover, and STARTS on multiple revisions from five open-source projects and compared the results in terms of test suite size reduction, end-to-end time reduction, safety and precision violation. We also conducted mutation testing to compare the fault detection ability of these four RTS techniques.

We compare the characteristics of those four RTS techniques that have not been systematically compared by other researchers. For example, Gligoric et al. [9] left the proof of the safety of Ekstazi as future work. One of the future works that Zhu et al. [18] left after focusing on finding faults in Ekstazi, OpenClover and STARTS was to compare the three tools with HyRTS. Our results show that among the dynamic techniques, Ekstazi, HyRTS, and OpenClover, OpenClover achieved the lowest safety violation (9.01%) with respect to STARTS, while STARTS achieved the lowest safety violation (0.87%) with respect to Ekstazi. OpenClover, however, achieved the highest precision violation, which is over 60% with respect to both STARS and Ekstazi. STARTS achieved the highest fault detection ability, and HyRTS achieved the lowest.

In the empirical evaluation, we identified program characteristics that have interaction effects with the performance of RTS techniques. The results of our evaluation show that STARTS, Ekstazi, and HyRTS achieved a higher test suite size reduction on the programs that have more lines of code. OpenClover achieved the lowest time reduction regardless of the program characteristics. These findings are useful for developers who need to select an appropriate RTS technique based on their priorities and the program they are testing.

## 1.4 Organization

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

# Chapter 2

# Related Work

In this chapter, we discuss RTS techniques related to our research. Section 2.1 presents the evolution of RTS techniques over time. Section 2.2 discusses existing empirical evaluations of RTS techniques.

## 2.1 Evolution of RTS Techniques

Simply executing all the test cases is also called the $RetestAll$ strategy. However, running all test cases is time-consuming. Surveys [2, 15, 21] show the various strategies used to develop RTS techniques, and the tools implemented for different programming languages (e.g., C, C++, Java, and AspectJ).

Changes in development environments is one of the factors that affected the evolution of RTS techniques [8]. Developers' expectations from RTS techniques have changed due to the growth of program size and the move toward rapid development cycles. Thus, while relatively old techniques [12, 16, 19] emphasized safety, more recent techniques [1, 22, 23] are designed to be faster with a little loss of safety.

Figure 2.1 shows that various methods have been used during the past three decades to compute dependencies between test cases and program code, and identify code changes, which are the two main RTS tasks. The top part of Figure 2.1 presents the methods used to identify code changes while the bottom part shows different methods to find test dependencies. Each method introduced in RTS is presented in chronological order. Overall, the methods used to identify code changes have become faster and more efficient. For example, comparing graphs (introduced in the late 1980s) takes relatively longer time than computing file changes using the diff tool (introduced in the late 1990s). Furthermore, computing smart checksums (introduced in the mid-2010s) is more efficient than using the diff tool because a smart checksum only identifies source code changes that affect program behavior. Each method to compute test dependencies has been further extended

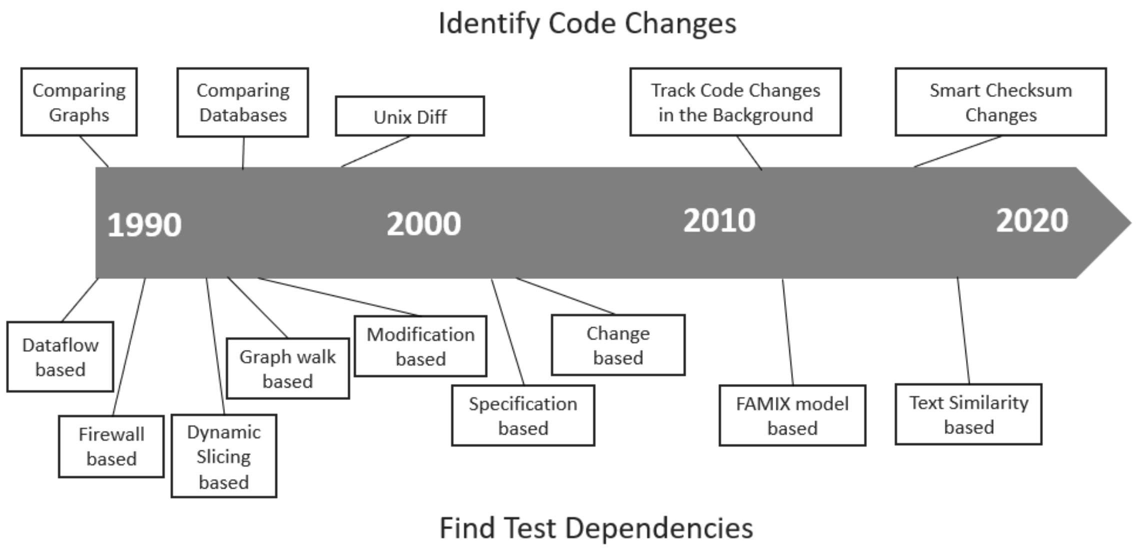

**Figure 2.1:** Timeline Showing Methods Used for Developing RTS Techniques

by other researchers over time for reasons such as changes in the programming languages. For example, RTS techniques in the 1990s used a control flow graph to support programs written in C. In the early 2000s, the control flow graph was extended to Java interclass graph.

The following literature survey shows that RTS techniques have generally evolved by improving upon the limitations of previous techniques, such as testing time, usability, and scope of application. As an example, the recent techniques [6, 9, 10] tend to identify code changes and find test dependencies at a coarser level for achieving higher cost-effectiveness, while older techniques used a finer-grained analysis.

Initially, RTS techniques were based on control flow graphs, data flow graphs, and slicing. The traditional Control Flow Graph (CFG) consists of nodes and edges, where the nodes represent basic blocks, and the edges show the flow of the program [24].

Leung and White [25] proposed a firewall based RTS technique at the module level. A firewall is a concept used to identify the boundary in the program that should be retested. Inside the firewall, source code is changed and there are parts, which are possibly affected by the code changes. The technique conducts both unit testing for the changes within the firewall and integration testing for the interaction between the modified modules. The technique identifies code changes based on the data flow graph and finds test dependencies by analyzing the test execution path. Later, Kung et al. [26] extended the firewall to a class firewall, which handles changes at the class-level, such as class inheritance. The firewall-based technique saves testing time by limiting the source code that needs to be analyzed due to the modification. However, the technique can be unsafe because it does not select test cases from outside of the firewall that may also reveal faults in the program [27].

Chen et al. [28] developed TestTube that is known as a modified code entity based RTS technique. TestTube categorizes program entities into two – functional (executable code such as assignment, if, loop statements) and non-functional (non-executable code such as global variable declarations and macro definitions), and those entities are saved in a database. TestTube computes test dependencies by computing test coverage during the test execution. TestTube compares the two databases for the old and modified versions of the program, and identifies the list of changed entities in the modified program. Later, Rosenblum and Rothermel [29] demonstrated that TestTube is not as precise as control flow graph based technique [19].

For the first time, Vokolos and Frankl [12] introduced an RTS technique based on textual differencing. They used the Unix *diff* command to find which source files were changed in the new revision at the statement level. This technique stores a basic block execution trace for each test case to use as a test dependency. Using the diff function is safe and fast. However, the technique can be imprecise since it does not determine whether the change made a difference to the program semantics.

Rothermel and Harrold [19] implemented a technique called DejaVu using the CFG. DejaVu identifies edges in the new revision that are impacted by program modification and selects test cases that cover the modified edges. CFG-based RTS techniques are more efficient in terms of the time taken to compare graphs and select test cases than data flow based techniques [2]. However, CFG-based techniques may omit fault-revealing test cases due to a lack of data dependency information.

Later, CFGs were extended to support features in object-oriented languages [30–34]. For instance, Rothermel et al. [30] developed an RTS technique for C++ programs using the Interprocedural Control Flow Graph (ICFG) and Class Control Flow Graph (CCFG). While a CFG represents a single method, an ICFG represents the interactions between multiple methods using $call$ and $return$ nodes. To represent programs that have multiple entry points (e.g., classes), the CCFG was proposed. Rothermel et al.'s technique [30] handles changes in both executable (e.g., assignment, conditionals, function calls, and iteration) and non-executable (e.g., declaration) statements such that the selected test cases find faults that the test cases selected by other techniques [19, 27] do not detect, such as faults in variable type changes on non-executable statements.

Harrold et al. [31] first extended the CFG to support Java using the Java Interclass Graph (JIG) to implement the tool, $RETEST$. This tool collects test coverage at the method level by instrumenting the code and identifies code changes by comparing two JIGs of original and modified versions of programs. The JIG handles various Java features (e.g., inheritance, polymorphism, exception handling) and does not require analysis on external classes (e.g., library classes). Techniques that support other object-oriented languages [30, 35] need a complete analysis of external resources if internal classes interact with external classes. However, building program graphs and comparing the execution traces become expensive as the program size increases [36].

Orso et al. [32] developed DejaVOO, which identifies changes at the edge level similar to DejaVu [19] and scales up to large-sized programs. DejaVOO partitions RTS phases into two and uses different graphs in each phase. DejaVOO creates the Interclass Relation Graph (IRG) to quickly identify code changes at the class level. Then, DejaVOO selects test cases at statement level using the JIG for precise selection. DejaVOO saves time in test selection by analyzing only the changed classes but still achieves high safety and precision. However, the empirical study [32] shows that the larger the program size, the higher the overhead in comparing two revisions.

A couple of other researchers also extended CFG to JIG [33, 37]. Further, Xu and Rountev [34] extended the JIG to AspectJ Inter-module Graph (AJIG) to support RTS for AspectJ programs. Generally, graph-based techniques are safe because the techniques are guaranteed to select test cases that traverse modified code, but the computation of graphs may be time-consuming and inefficient for large programs.

Distinct from code-based RTS techniques, Chen et al. [38] developed a black-box RTS technique based on models. Instead of source code analysis, the technique relies on the program specifications and uses traceability between the design and test cases. The UML activity diagram represents program behavior. Chen et al. [38] create a traceability matrix using this UML activity diagram by check the covered paths and nodes for each test case. Similar to CFG based techniques [19], code changes are identified at the edges level in the activity diagram, and the test cases that execute the affected edges are selected using the traceability matrix. Chen et al. [38] state that the techniques using program specifications are useful in industrial programs because those techniques are not limited to supporting a specific programming language and do not require a tester to understand source code.

Soetens et al. [13] 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. While previous RTS techniques [32, 34] were evaluated on relatively small-sized programs (e.g., open source libraries, and sample packages that comes with tools), Soetens et al. [13] evaluated ChEOPSJ on larger and more complex industrial programs. However, ChEOPSJ can be unsafe because it does not support certain Java features (e.g., polymorphism) and can miss relevant test cases [39].

Gligoric et al. [9], Legunsen et al. [6], and Zhang [10], on the other hand, computed smart checksums which ignore changes that do not impact debug information. The techniques [6, 9, 10] are used in our empirical comparison, and we describe details in Section 3.

Recently, researchers have developed RTS techniques that are easy to adapt to different programming languages. Romano et al. [40] proposed SPIRITuS, which uses lexical similarities to identify changed methods, and method coverage information for dependencies. In addition to being able to handle any programming language, the technique is also flexible because users can change the test selection threshold if they want to select more or fewer test cases. Depending on how users set a threshold, the technique can be unsafe.

ReTEST, introduced by Azizi and Do [41], is not limited to supporting a specific programming language. ReTEST compares two versions of the program using a diff tool to collect terms from the part of the changed code to construct queries. In this context, queries, often called user queries, are formal statements of information needs in information retrieval. Then, instead of collecting test dependencies, ReTEST uses the failure history of tests, test case diversity, the program change history, and the textual similarity of program changes. The fault detection ability of the tool is affected by a slight difference of many factors, such as the similarity score (adjustable by users) and the ratio of the number of tests to the number of queries. Azizi and Do [41] empirically demonstrate that the performance of ReTEST is consistent regardless of the growth of the number of test cases while some RTS techniques [32, 36] are affected by the size of the program. This is because ReTEST is based on the test case graph database, where the database efficiently stores nodes (test cases), edges (diversity between test cases), and properties (e.g., test failure history).

Recently, companies in the industry proposed RTS techniques that aim to shorten test time and scale-up to industrial programs that may be less safe [8]. To develop such techniques, Google [1] utilized features that were not used previously, such as test execution frequencies and information of developers. Facebook [22] applied machine-learning to RTS. Microsoft used project level test dependencies (Jar granularity changes and test dependencies). Microsoft [42] demonstrated that project-level RTS selects as much as 17.4% fewer test cases than class-level RTS.

## 2.2 Previous Empirical Evaluations of RTS techniques

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 [21]. Table 2.1 shows that those techniques used to conduct empirical studies have become more diverse over time. For example, researchers [6, 17, 32, 39, 43, 44] have demonstrated the cost reduction of RTS techniques compared with the original test suite. As more RTS techniques have been developed and are publicly available, relatively recent empirical studies [10, 18, 40, 41, 45] used other RTS techniques as comparison targets more often. Also, programs from Siemens benchmarks occasionally used as subject programs in empirical evaluations [17, 19, 43] of RTS techniques around the late 1990s, while recent studies [6, 9, 10] use 20-30 various open-source programs.

**Table 2.1:** Summary of Past Empirical Evaluations of RTS Techniques

| **Reference (Year)** | **PL(s)** | **Comparison Target(s)** | **Subject Program(s)** | **Metrics Used** |
|---|---|---|---|---|
| Rothermel and Harrold [43] (1997) | C | Retest all | Siemens | end-to-end time red.<br>test suite size red. |
| Rothermel and Harrold [19] (1997) | C | n/a | Siemens | end-to-end time red.<br>test suite size red. |
| Harrold et al. [46] (1998) | C | Aristotle | Unix utilities | graph size red. |
| Graves et al. [17] (2001) | C | Retest all<br>Random | Siemens | test suite size red.<br>fault detection ability |
| Rothermel et al. [30] (2001) | C++ | n/a | C++ library | test suite size red. |
| Orso et al. [32] (2004) | Java | Retest all | open-source projects | test exec time red.<br>test suite size red.<br>precision |
| Xu and Rountev [34] (2007) | AspectJ | AspectJ-compiler | AspectJ-compiler-example package | test suite size red.<br>end-to-end time |

| | | | | |
|---|---|---|---|---|
| Chittimalli and Harrold [45] (2008) | C | DejaVoo | real-world-applications | test suite size red.<br>safety<br>precision |
| Soetens et al. [39] (2013) | Java | Retest all | open-source projects | test suite size red.<br>safety<br>precision<br>fault detection ability |
| Shi et al. [14] (2014) | Java | Retest all<br>Random | open-source projects | test suite size red.<br>fault detection ability |
| Gligoric et al. [9] (2015) | Java | Retest all<br>FaultTracer | open-source projects | end-to-end time red.<br>test exec time red.<br>test suite size red. |
| Soetens et al. [20] (2016) | Java | Retest all | open-source projects | end-to-end time red.<br>test suite size red.<br>fault detection ability |
| Legunsen et al. [6] (2017) | Java | Retest all | open-source projects | end-to-end time red.<br>test suite size red. |
| Zhang [10] (2018) | Java | Ekstazi | open-source project | end-to-end time red.<br>test exec time red.<br>test suite size red. |
| Romano et al. [40] (2018) | Any-language | Unix Diff<br>Random<br>Ekstazi | open-source projects | test suite size red.<br>safety<br>fault detection ability |
| Azizi and Do [41] (2018) | Any-language | Modified-DejaVu | open-source projects | test suite size red.<br>fault detection ability |

| | | | | |
|---|---|---|---|---|
| Zhu et al. [18] (2019) | Java | Clover<br>Ekstazi<br>STARTS | open-source projects | safety violation<br>precision violation<br>generality violation |
| Fu et al. [44] (2019) | C++ | Retest all | open-source projects | end-to-end time<br>test suite size red. |

Before researchers started comparing RTS techniques with each other [17, 29, 47], they often evaluated their proposed technique by itself. Some studies compared the newly proposed technique with $RetestAll$ with respect to the time required to select and run test cases, and also the reduction in the test suite size [43]. Other empirical studies were conducted using different-sized programs to demonstrate that their technique selects fewer test cases [46]. The early comparative studies of RTS techniques occasionally included a comparison with random selection [17]. For the subjects used in the empirical evaluation, researchers seeded faults manually or used subjects that have known faults, such as the Siemens benchmarks that contain realistic faults seeded in seven C programs. Engström et al. [15] summarized that 70% of RTS-related empirical studies published before 2006 consider the metrics test suite reduction and total testing time.

Rothermel and Harrold [19] defined four evaluation criteria: inclusiveness, precision, efficiency, and generality. Inclusiveness measures if a technique is safe by computing the number of selected test cases that traverse modification code. 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, such as safety and precision [45], to compare test selection accuracy. Chittimalli and Harrold [45] computed false positives and false negatives of the selected test cases to calculate safety and precision. They had ground-truth information about which test cases should be selected because the developers provided the programs. Soetens et al. [39] implemented scripts to conduct a dynamic analysis that executes the original test suite to trace the relationship between test cases and source code methods. Then, they computed the safety and precision by comparing the list of test cases selected by their tool and those obtained from 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 [7, 8]. In this way, researchers can demonstrate whether a new technique is as safe (or precise) as the 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. Andrews et al. [48] show that mutation faults can be effectively used instead of real faults in Software Engineering experiments. Mutation testing has been applied to empirically evaluate Java-based regression testing techniques to compute the fault detection ability of the selected test cases [14, 20, 39]. Researchers compared the mutation scores of the test cases selected by the RTS technique with that of the original test suite [40]. Many researchers used PIT for mutation testing because Java projects based on Ant or Maven can easily adopt PIT.

Zhu et al. [18] developed a framework, called RTSCheck, to verify if RTS tools themselves contain faults. This research mainly focuses on examining the reliability of RTS techniques by conducting empirical studies. RTSCheck computes the safety violation, precision violation, and generality violation of RTS tools. Generality violation detects if RTS techniques include unexpected behavior, such as the occurrence of test failure even though there was no test failure in the original test suite. As a result, RTSCheck found 27 bugs in recent Java-based RTS tools, such as the inability to detect changes in non-Java files (e.g., configuration files) and unexpected behaviors with specific annotations.

Many researchers have compared RTS techniques empirically [6, 9–11, 40]. The survey by Kazmi et al. [21] 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. Furthermore, researchers used various open-source projects for empirical evaluation based on the compatibility with the tools (e.g., Java and JUnit versions).

# Chapter 3

# Background

In this chapter, we explain the four Java-based techniques that we 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. The sections present the RTS techniques in chronological order of development.

## 3.1 Ekstazi

Ekstazi [9] 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 that are 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 [39]. Ekstazi aims to balance the program analysis and test running time. Gligoric et al. [9] 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 [6] is a static RTS technique that uses class-level analysis. In work prior to STARTS, Legunsen et al. [7] stated that their research was motivated by the usability of Ekstazi [9]. 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. [7] conducted a study that showed that class-level static RTS (65.3% of the retest-all time) was 2.9% faster than Ekstazi (68.2% of the retest-all 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. [6] demonstrated that STARTS can achieve a reduction on average of 12.4% of the end-to-end 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 [7] shows that 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 [9]. HyRTS [10] 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 [10] demonstrated that 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 [11], 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 [49], 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 [49] show that 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.

# Chapter 4

# Research Design

This chapter 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 mutation testing was 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 3.2GHz machine with 12GB memory, Java 64-Bit Server version 1.8.0_242. We fully automate 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.

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

$$TestSuiteReduction = \frac{|T| - |T'|}{|T|} \tag{4.1}$$

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, $t'$,

$$EndToEndTimeReduction = \frac{t - t'}{t} \tag{4.2}$$

**Safety violation** Assume that 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.

$$SafetyViolation = \frac{|T_1 - T_2|}{|T_1 \cup T_2|} \tag{4.3}$$

**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.

$$PrecisionViolation = \frac{|T_2 - T_1|}{|T_1 \cup T_2|} \tag{4.4}$$

**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.

$$FaultDetectionAbility = \frac{KilledMutants}{NumberOfTotalMutants} \tag{4.5}$$

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.

We found four program features that may have interaction effects with the performance of RTS techniques: 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 that have over 100 KLOC as large-sized programs as other RTS empirical studies [27, 50] considered. As such, the projects that have 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 [51]. This factor measures the total number of classes including programs and test cases.

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

$$PercentageOfTestClasses = \frac{NumberOfTestClasses}{NumberOfTotalClasses} \times 100 \tag{4.6}$$

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 the performance of RTS techniques.

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

$$PercentageOfChangedClasses = \frac{NumberOfChangedClasses}{NumberOfTotalClasses} \times 100 \quad (4.7)$$

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 4.1 shows the subjects used in our empirical study. These programs were used by other researchers [6, 9, 10]. 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.

**Table 4.1:** 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 |

Table 4.1 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 the test classes in the total number of classes compares to the rest of other 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 the 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 compares 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.

## 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 collect 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 [6] reuses the part of the Ekstazi source code to computes checksum. HyRTS [10] also computes the checksum in the same way with Ekstazi and STARTS. However, STARTS is the only RTS tool that provides the command-line option to show files that are identified as changed among the four RTS tools.

## 4.4 Mutation Testing

Mutation testing [52] 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. 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 [48]. Accordingly, many mutation testing tools have been developed [53], 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 [14, 20, 54].

In our experiment, we conducted mutation testing to compare the fault detection ability of the tests selected 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.

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.

When executing PIT, we edited the configuration to run 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 redirect 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.

## 4.6 Statistical Data Analysis

We conducted both non-parametric and parametric statistical analysis to compare the RTS techniques.We used non-parametric tests because our data is not normally distributed. But we also used the more conservative parametric tests to corroborate the non-parametric test results. We first 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 [55]. The nparLD is known for being robust even for a small sample size. 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 small sample size than other groups. We used an alpha of 0.05 as the cut-off for our statistical tests.

Akritas et al [56] defined the Wald-type statistic as follows:

$$Q_n(\mathbf{C}) = n\widehat{\mathbf{p}}^\top \mathbf{C}^\top [\mathbf{C}\widehat{\mathbf{V}}_n \mathbf{C}^\top]^+ \mathbf{C}\widehat{\mathbf{p}} \tag{4.8}$$

under the hypothesis $H_0^F$: $\mathbf{CF} = \mathbf{0}$ where C is a contrast matrix, $\mathbf{F}$ 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 [57]. The non-parametric test was conducted on RStudio using the R software package provided by Noguchi et al [55]. 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: $\alpha$/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 [58].

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) [59]. 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 [60]. 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.

# Chapter 5

# Results and Discussion

In this chapter, we present the answers to the five research questions and then discuss several threats to validity. Sections 5.1-5.5 show the results of 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

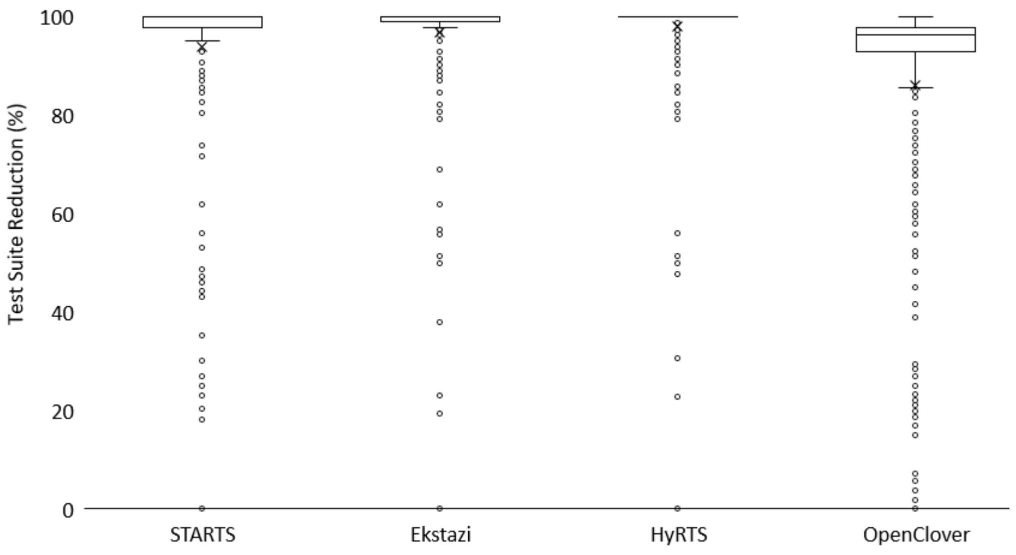

**Figure 5.1:** Test Suite Reduction Rate

The boxplots shown in Figure 5.1 display the percentage 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 quartile of STARTS, Ekstazi, and HyRTS is the same or close to the median and max value because 64.35% of revisions do not have files whose smart checksums changed. The mean value shows that 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 that OpenClover selects and runs test cases from the revisions on which the other tools did not run any test case.

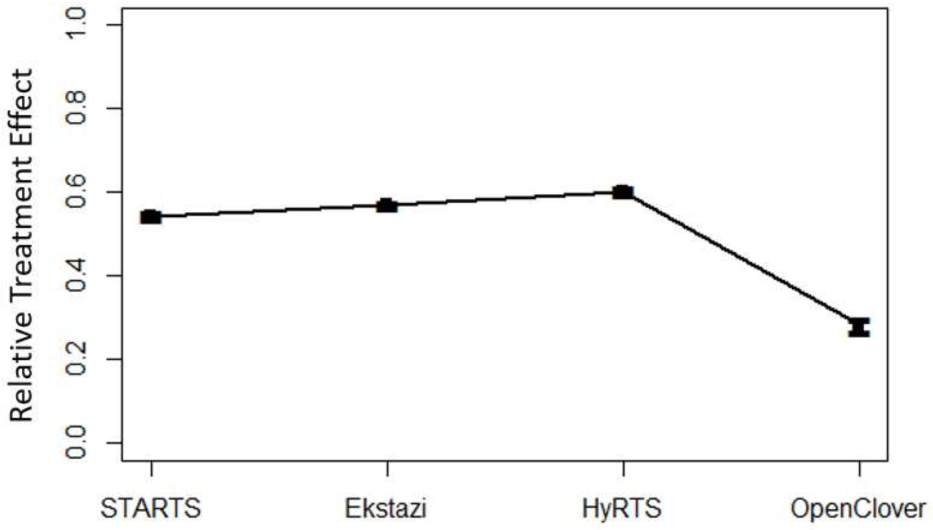

**Figure 5.2:** Non-parametric Test Result with Test Suite Size Reduction

Figure 5.2 shows the line plot for the non-parametric test result using the test suite size reduction achieved by the four tools on all the revisions that were considered in the study. The points of the relative treatment effect (RTE) appear in an order of OpenClover < STARTS < Ekstazi < HyRTS. This can be interpreted to mean that HyRTS achieved the highest test suite size reduction while OpenClover achieved the lowest. As a validation of the non-parametric test, the p-value of Wald-Type statistic was 4.08739e-263. This shows that there are statistical differences in four test suite size reductions because the p-value is less than 0.05.

However, the result of the non-parametric test using nparLD does not show whether differences in test suite size reductions between pairs of RTS techniques are statistically different. For example, the differences 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, it is not clear if the differences between the number of test cases selected by OpenClover and the other techniques are statistically significant.

**Table 5.1:** The Bonferroni Test Result with Test Suite Size Reduction

| **Comparison** | **Difference** | **p-value** |
|---|---|---|
| Ekstazi - HyRTS | -1.209246 | 1.0000 (p > 0.05) |
| Ekstazi - OpenClover | 10.785545 | 0.0000 (*** p <= 0.001) |
| Ekstazi - STARTS | 2.887387 | 0.0122 (* p <= 0.05) |
| HyRTS - OpenClover | 11.994792 | 0.0000 (*** p <= 0.001) |
| HyRTS - STARTS | 4.096633 | 0.0001 (*** p <= 0.001) |
| OpenClover - STARTS | -7.898159 | 0.0000 (*** p <= 0.001) |

Therefore, we conducted the Bonferroni tests, and the results are shown in Table 5.1. Each row in the table shows if the differences in two RTS techniques is statistically significant. The symbols below p-value shows the p-value visually with * for p<=0.05, ** for p <= 0.01, and *** for p <= 0.001. The results indicates 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

Figure 5.3 shows the 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

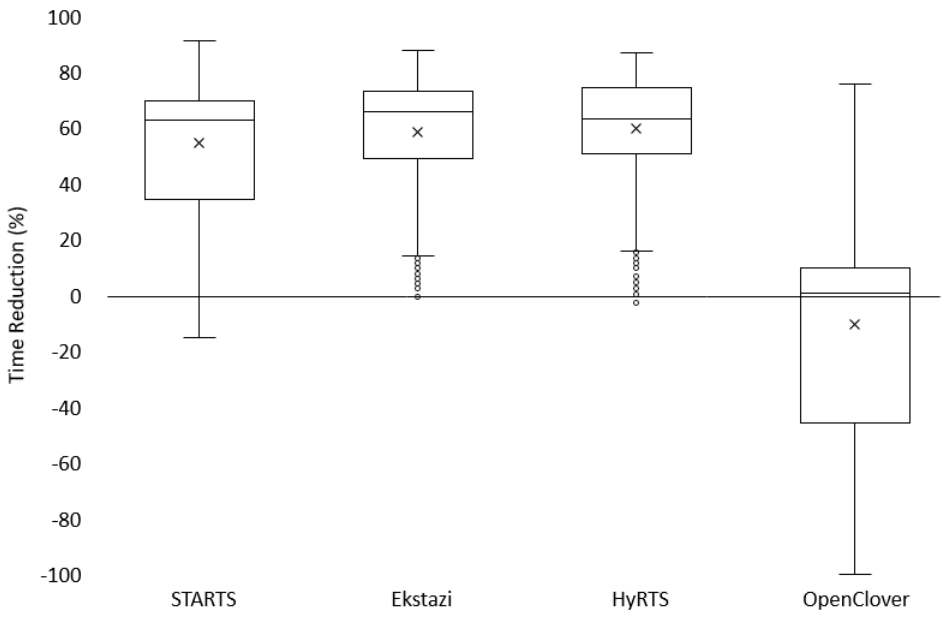

**Figure 5.3:** End-to-end Time Reduction Rate

and mean value of the end-to-end time reduction is achieved by Ekstazi (65.44%) and HyRTS (59.40%), respectively. OpenClover achieved the lowest median end-to-end time reduction at 0.58%. Because the mean test suite reduction of STARTS is 2.89% higher than Ekstazi, the mean value of end-to-end time reduction achieved by Ekstazi is 2.93% higher than STARTS.

However, these techniques do not guarantee a reduction in testing time. We observed that STARTS, Ekstazi, and OpenClover achieved minus end-to-end time reduction in some revisions. The values below zero indicate using RTS took a longer time than running the original test suite. In particular, OpenClover spent a longer time than running the original test suite on 16 times more revisions than STARTS. OpenClover's official website [61] explains the limitation that the more class files and test cases exist in the project, the worse OpenClover's performance is in terms of the testing time and memory usage. That is because the number of per-test coverage files generated by OpenClover equals the number of class files multipled by the number of test cases.

Figure 5.4 shows the non-parametric test result based on end-to-end time reduction of the four RTS techniques. The points of the RTE appear from OpenClover with the smallest (0.2206) to HyRTS with the largest (0.5995) in the same order of test suite size reduction. We identified

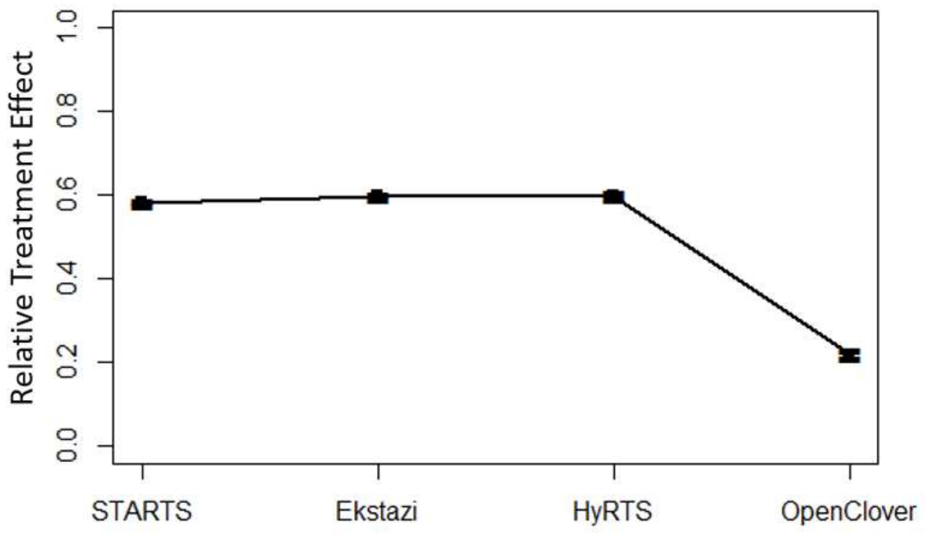

**Figure 5.4:** Non-parametric Test Result with Time Reduction

that the end-to-end time reduction achieved by the four RTS techniques is statistically significant because the p-value of Wald-Type statistic was 0.

**Table 5.2:** The Bonferroni Test Result with End-to-end Time Reduction

| Comparison | Difference | p-value |
|---|---|---|
| Ekstazi - HyRTS | -0.9092542 | 1.0000 (p > 0.05) |
| Ekstazi - OpenClover | 67.4111792 | 0.0000 (*** p < 0.001) |
| Ekstazi - STARTS | 2.5339409 | 1.0000 (p > 0.05) |
| HyRTS - OpenClover | 68.3204334 | 0.0000 (*** p < 0.001) |
| HyRTS - STARTS | 3.4431951 | 0.5416 (p > 0.05) |
| OpenClover - STARTS | -64.8772382 | 0.0000 (*** p < 0.001) |

Table 5.2 shows the result of the Bonferroni test conducted with end-to-end time reduction achieved by the RTS techniques. Even though Figure 5.4 shows that the end-to-end time reduction was STARTS < Ekstazi < HyRTS, Table 5.2 represents that the end-to-end time reductions of those

three techniques are actually not statistically different. On the other hand, OpenClover achieved a statistically lower end-to-end time reduction than other techniques.

There were similarities and differences between the results of the parametric test and the non-parametric test. We confirmed with a parametric test with Huynh-Feldt correction that there are significant effect of the within-subject factor on end-to-end time reduction (F = 1143.64, df = 1.45, p < 0.0000). On the other hand, the pairwise comparisons show that the p-value of Ekstazi and HyRTS pair is 0.547. That means the parametric test indicates that the end-to-end time reduction achieved by Ekstazi and HyRTS are statistically similar but different from STARTS and OpenClover.

## 5.3 Safety and Precision Violation

Figures 5.5 and 5.7 show the safety and precision violations of the tools with respect to STARTS and Ekstazi. We used STARTS and Ekstazi as baselines because they are considered to be the state-of-art RTS techniques [54]. In both the figures, the three box plots with the symbol _$S$ show the violations computed with respect to STARTS. The three box plots with the symbol _$E$ are violations computed with respect to Ekstazi.

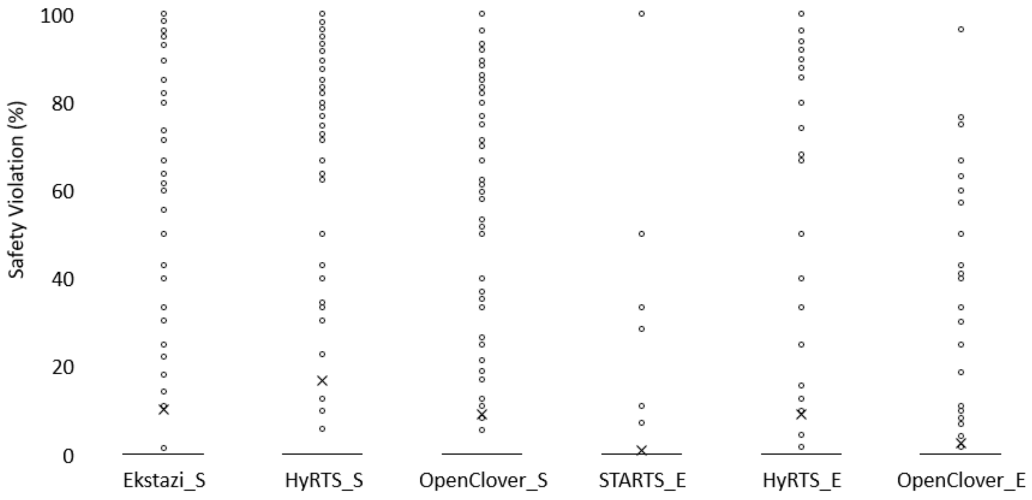

**Figure 5.5:** Safety Violation

Figure 5.5 shows that the median safety violation values of the four RTS techniques is 0 but there are many outliers. Considering that 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. That means the safety violations of the 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 six 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. That means approximately 8% of test cases that OpenClover selects are irrelevant to the code changes that STARTS and Ekstazi identified.

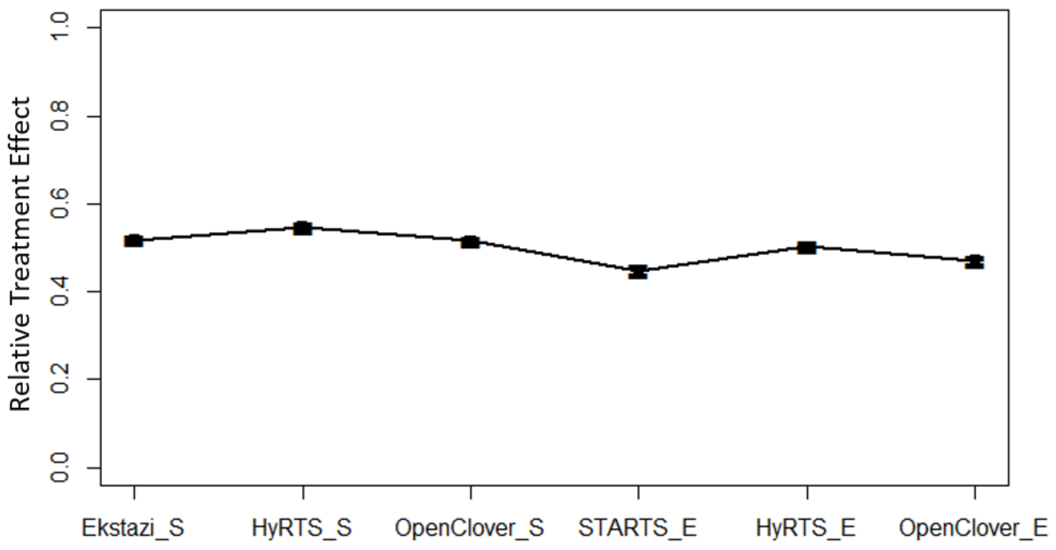

**Figure 5.6:** Non-parametric Test Result with Safety Violation

Figure 5.6 shows that the gap between the largest (0.5467 by HyRTS with respect to STARTS) and the smallest (0.4481 by STARTS with respect to Ekstazi) RTE of the safety violation is less than 0.1. Despite the small RTE gaps between the techniques, the Wald-Type statistic still gives less than 0.05 p-value (4.2162523e-27).

**Table 5.3:** The Bonferroni Test Result with Safety Violation

| Comparison | Difference | p-value |
|---|---|---|
| Ekstazi_S - HyRTS_S | -6.37645289 | 1e-04 (*** p < 0.001) |
| HyRTS_S - OpenClover_S | 7.61132727 | 0e+00 (*** p < 0.001) |
| Ekstazi_S - OpenClover_S | 1.23487438 | 1e+00 (p > 0.05) |
| HyRTS_E - OpenClover_E | 6.41737521 | 0e+00 (*** p < 0.001) |
| HyRTS_E - STARTS_E | 8.15797851 | 0e+00 (*** p < 0.001) |
| OpenClover_E - STARTS_E | 1.74060331 | 1e+00 (p > 0.05) |

We conducted the Bonferroni test accordingly, and Table 5.3 shows the test result. The safety violation of STARTS with respect to Ekstazi is the lowest among the six violations as Figure 5.6 shows. However, Table 5.3 present that the safety violation of STARTS is higher than HyRTS, but the safety violation of STARTS is not statistically different from OpenClover with respect to Ekstazi. This can be explained with the test suite size reduction. STARTS, being a static RTS technique, selects more test cases than dynamic RTS techniques. The parametric test with Huynh-Feldt correction shows that there is a significant effect of the within-subject factor on safety violation of RTS techniques with respect to both STARTS (F = 155.715, df = 2.56, p < 0.000) and Ekstazi (F = 94.75, df = 1.49, p < 0.000).

In Figure 5.7, 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. That can be explained by the observation that HyRTS and Ekstazi have not 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 four RTS technique in each revision is 9.08. We observed that OpenClover selects more than 18 test cases in 103 revisions. That means OpenClover selected

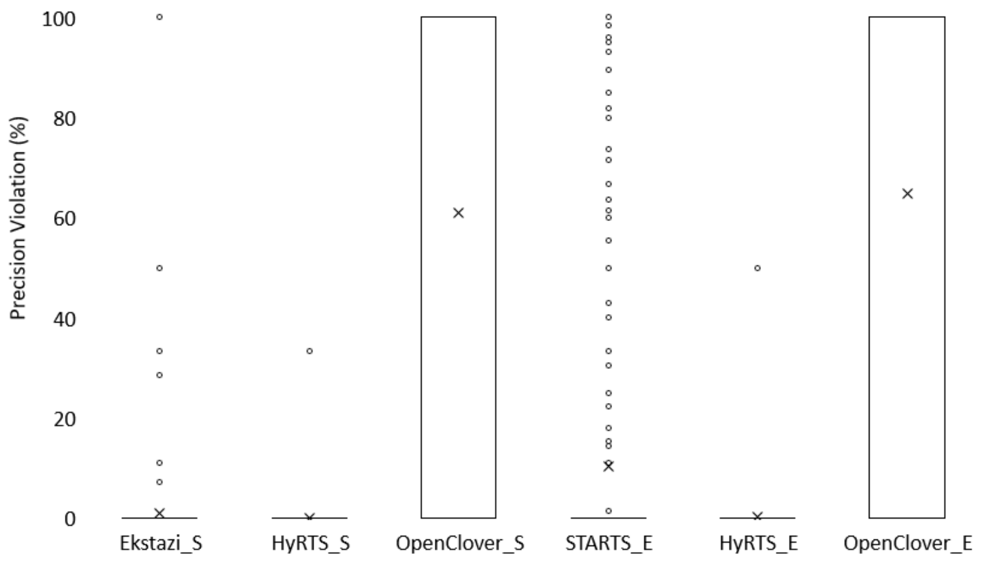

**Figure 5.7:** Precision Violation

two times more test cases than the average test case selection in 17.82% of total revisions. This explains why there is a 100% precision violation. That is also shown in the test suite size reduction.

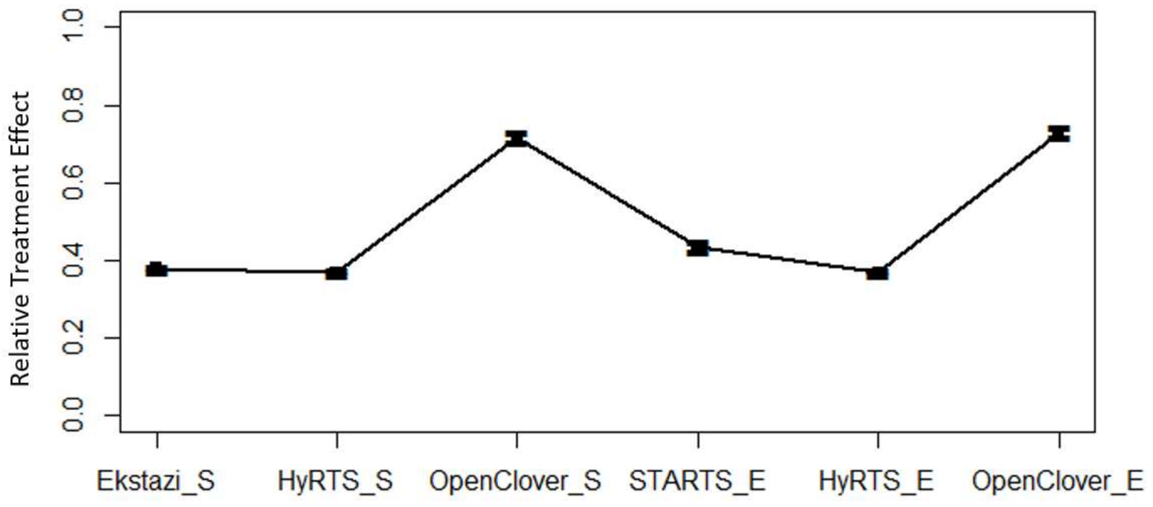

**Figure 5.8:** Non-parametric Test Result with Precision Violation

In Figure 5.8, the precision violations of OpenClover are the highest among all the violations with respect to both STARTS and Ekstazi. The post-hoc test results in Table 5.4 shows that the precision violations of OpenClover are statistically different from other techniques. Based on Fig-

**Table 5.4:** The Bonferroni Test Result with Precision Violation

| Comparison | Difference | p-value |
|---|---|---|
| Ekstazi_S - HyRTS_S | 0.81513884 | 1.0000 (p > 0.05) |
| HyRTS_S - OpenClover_S | -60.80159504 | 0.0000 (*** p < 0.001) |
| Ekstazi_S - OpenClover_S | -59.98645620 | 0.0000 (*** p < 0.001) |
| HyRTS_E - OpenClover_E | -64.77200165 | 0.0000 (*** p < 0.001) |
| HyRTS_E - STARTS_E | -10.16311570 | 0.0000 (*** p < 0.001) |
| OpenClover_E - STARTS_E | 54.60888595 | 0.0000 (*** p < 0.001) |

ure 5.8 and Table 5.4, the precision violation of HyRTS with respect to Ekstazi is statistically lower than STARTS. The Huynh-Feldt correction confirmed that the precision violations with respect to STARTS (F = 178.05, df = 1.07, p < 0.000) and Ekstazi (F = 167.46, df = 1.53, p < 0.000) have a significant effect on the within-subject.

## 5.4 Fault Detection Ability

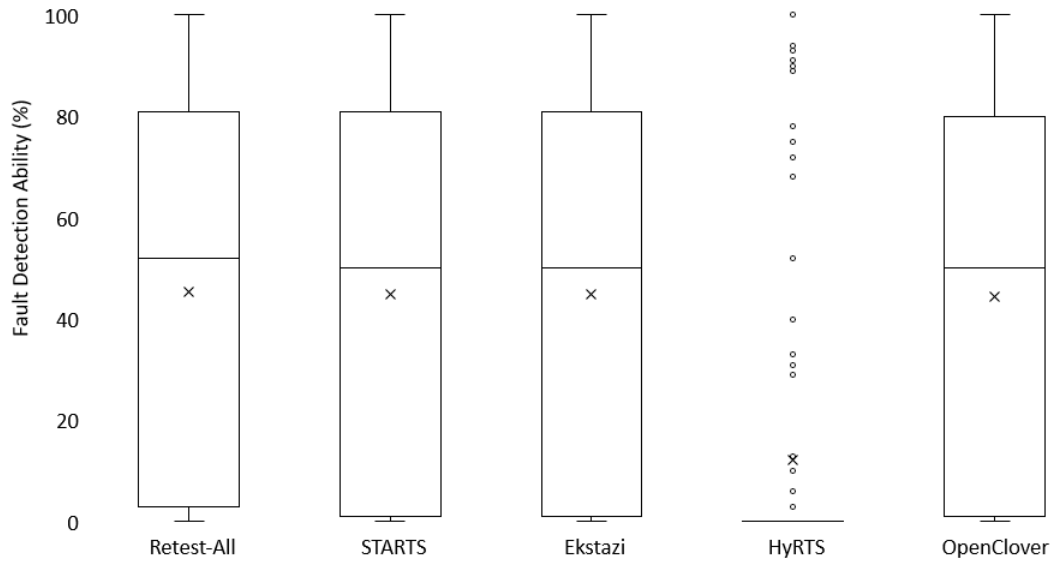

**Figure 5.9:** Fault Detection Ability

Figure 5.9 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.

PIT ran only on 25.26% of total revisions because of two reasons. First, 35.47% of the all the revisions used in our study have files whose smart checksums changed between revisions. As explained in Chapter 4.4, we conducted mutation testing only on the revisions that have changed files. That means 64.53% of revisions were excluded from the mutation testing. Second, mutation testing failed on some revisions due to the error saying that test failure exists even though there was no test failure when running JUnit tests with the original test suite. This is an issue that occasionally appear with PIT due to these causes listed here [62]: 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. That 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 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 in which files are determined as changed files, so we could not determine if the problem is a result of misidentifying changed files or finding test dependencies. Even though HyRTS computes smart checksums like Ekstazi and STARTS, Zhang [10] 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.

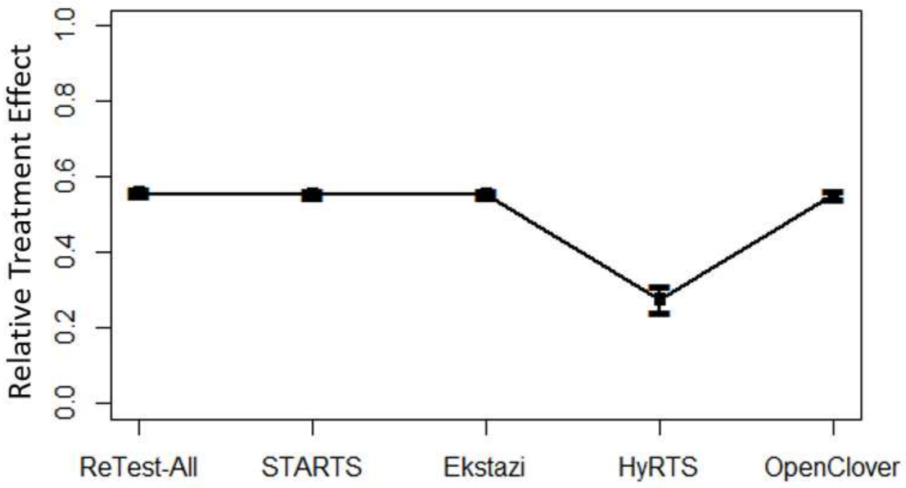

**Figure 5.10:** Non-parametric Test Result with Fault Detection Ability

Figure 5.10 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. Ekstazi and OpenClover achieved as high RTE as STARTS (less than 0.002 difference). Because the gap between RetestAll, STARTS, Ekstazi, and OpenClover are small, it is difficult to identify their differences in Figure 5.10. The p-value of Wald-Type statistic was 6.000201e-37, so we conducted post-hoc test.

Table 5.5 shows the Bonferroni test result with fault detection ability. The table shows that the fault detection ability achieved by HyRTS is statistically significant from other techniques. Figure 5.10 also presents that HyRTS killed the smallest number of mutants. The parametric test also shows that there are statistical differences existing in the fault detection ability achieved by the RTS techniques (F = 57.24, df = 1.09, $p < 0.000$).

**Table 5.5:** Bonferroni Test Result with Fault Detection Ability

| Comparison | Difference | p-value |
|---|---|---|
| Ekstazi - HyRTS | 32.60689655 | 0<br>(*** p < 0.001) |
| Ekstazi - OpenClover | 0.3862069 | 1<br>(p > 0.05) |
| Ekstazi - RetestAll | -0.52413793 | 1<br>(p > 0.05) |
| Ekstazi - STARTS | -0.08965517 | 1<br>(p > 0.05) |
| HyRTS - OpenClover | -32.22068966 | 0<br>(*** p < 0.001) |
| HyRTS - RetestAll | -33.13103448 | 0<br>(*** p < 0.001) |
| HyRTS - STARTS | -32.69655172 | 0<br>(*** p < 0.001) |
| OpenClover - RetestAll | -0.91034483 | 1<br>(p > 0.05) |
| OpenClover - STARTS | -0.47586207 | 1<br>(p > 0.05) |
| RetestAll - STARTS | 0.43448276 | 1<br>(p > 0.05) |

## 5.5 Interaction Effects Between Program Characteristics and Performance of RTS Techniques

In Section 4.1, we defined four program features that can potentially have interaction effects with the performance of RTS techniques: 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 three of the metrics achieved by RTS techniques, test suite size reduction, end-to-end time reduction, and fault detection ability. However, 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 [15].

We divided this section into four parts, one for each program feature.

### 5.5.1 Total Size in KLOC.

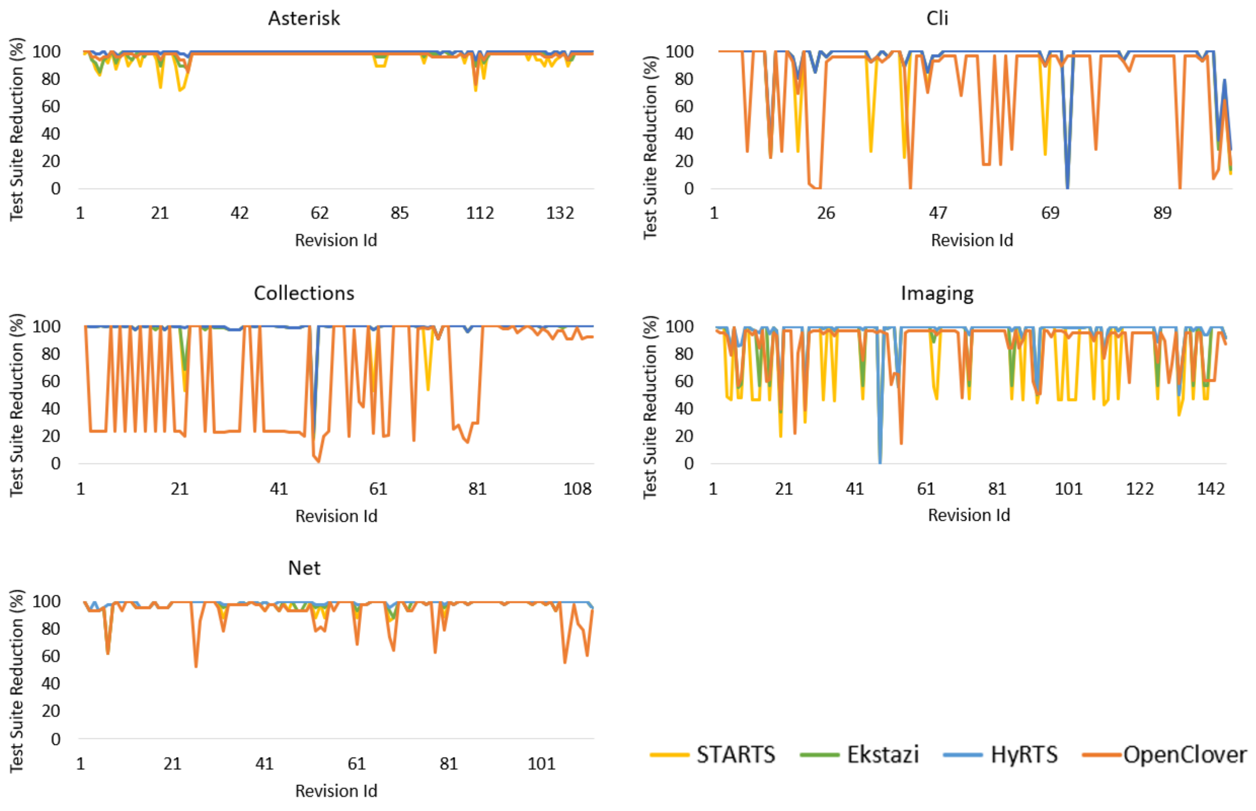

**Figure 5.11:** Test Suite Reduction per Subject

**Test suite size reduction.** Figure 5.11 depicts the test suite size reduction per subject. Ekstazi, STARTS, and HyRTS achieved higher test suite size reduction in projects with over 100 KLOC than in projects that have less than 100 KLOC. The difference of reduction achieved in the projects between over 100 KLOC and less than 100 KLOC was biggest in STARTS (5.41%) and smallest in HyRTS (2.36%). Because of that, the lines that represent STARTS, Ekstazi, and HyRTS are barely visible on the subjects that have over 100 KLOC (Asterisk and Commons Collections) than the rest of other subjects in Figure 5.11. The test suite reduction achieved by OpenClover does not show a pattern with change in KLOC.

Note that the line graphs are stacked in the order of STARTS, Ekstazi, HyRTS, and OpenClover. Thus, if several techniques achieve the same reduction on 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 look like HyRTS is the only technique that achieved 0% test suite reduction. However, STARTS, Ekstazi, and HyRTS ran all test cases on those revisions, but it looks like the blue line is the only one achieved zero reduction because of how the order appears on the line graph.

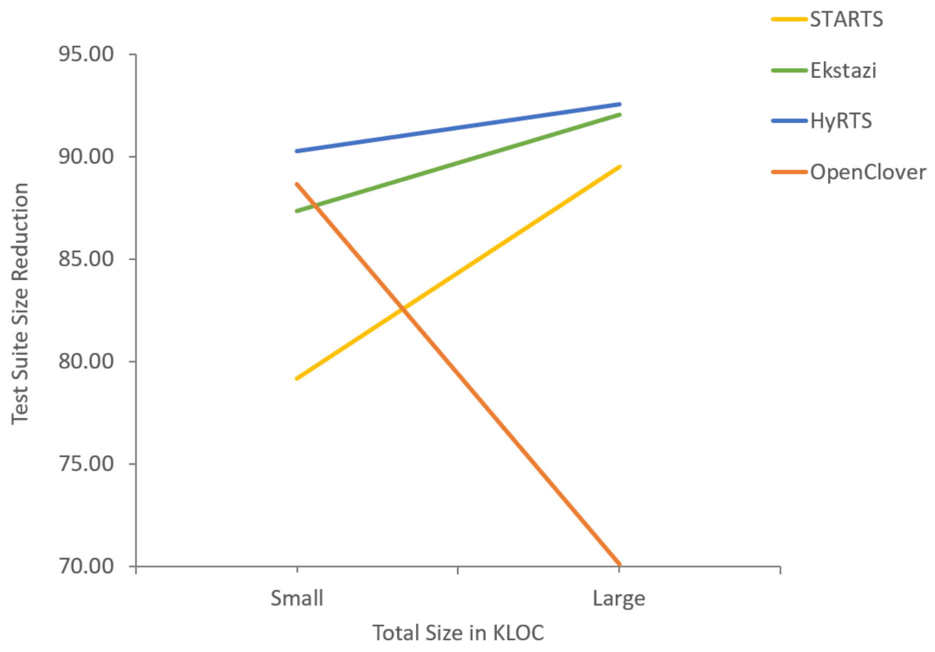

**Figure 5.12:** Two-way Interaction Between RTS Technique and Total Size in KLOC on Test Suite Size Reduction

We can find two statistical results from the non-parametric test: (1) If there are statistical differences in test suite size reduction where program sizes are small vs large, and (2) if an interaction effect exists between total size in KLOC and the test suite size reduction of RTS techniques. The p-values for the Wald-Type statistics were 3.729602e-02 and 3.064411e-02, respectively, over 0.05 in both results. Thus, the non-parametric test result indicates that there is no statistical interaction effect between test suite size reduction and total size in KLOC.

The non-parametric test results showed that neither the main effect of total size in KLOC nor the interaction effect between total size in KLOC and RTS technique on test suite size reduction was statistically significant (i.e., p-values for the Wald-Type statistics were >= 0.05). The parametric between-subjects test for difference in test suite size reduction by total size in KLOC was also 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).

As shown in Figure 5.12, STARTS, Ekstazi, and HyRTS achieved higher test suite size reduction in projects with over 100 KLOC than in projects that have less than 100 KLOC. In contrast, OpenClover selected more test cases on projects with over 100 KLOC and fewer cases on projects that had less than 100 KLOC.

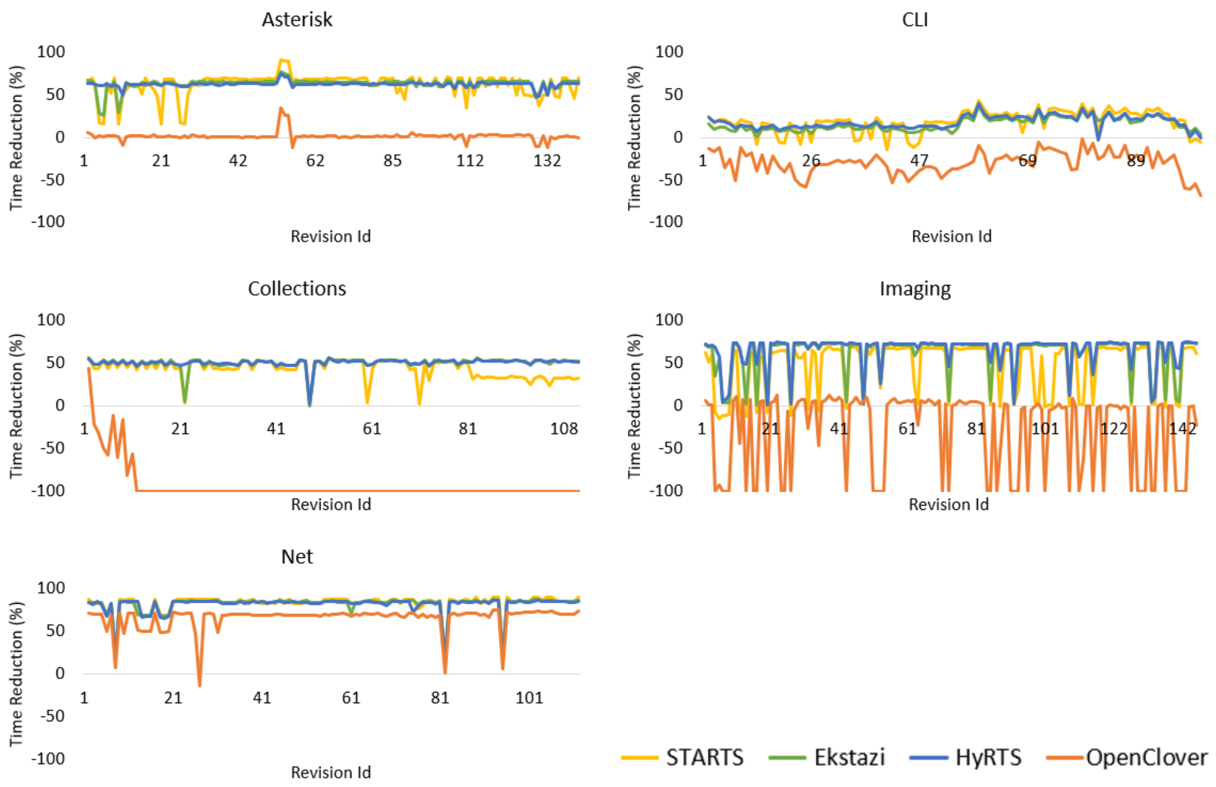

**Figure 5.13:** End-to-end Time Reduction per Subject

**End-to-end time reduction.** Figure 5.13 shows the time reduction per subject. Ekstazi and STARTS tend to reduce more time on the programs that have 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 that represent STARTS and Ekstazi in Asterisk are mostly placed over 50%, while those lines appear even below zero in CLI in Figure 5.13.

Our non-parametric test results showed that the end-to-end time reduction in the projects that had less than 100 KLOC and the projects that had over 100 KLOC are statistically different. The parametric between-subjects test for differences in the end-to-end time reduction by total size in KLOC was statistically significant (F = 342.590, df = 1, p < 0.000). The interaction effect between total size in KLOC and RTS technique on the end-to-end time reduction was significant (F = 180.282, df = 1.45, p < 0.000).

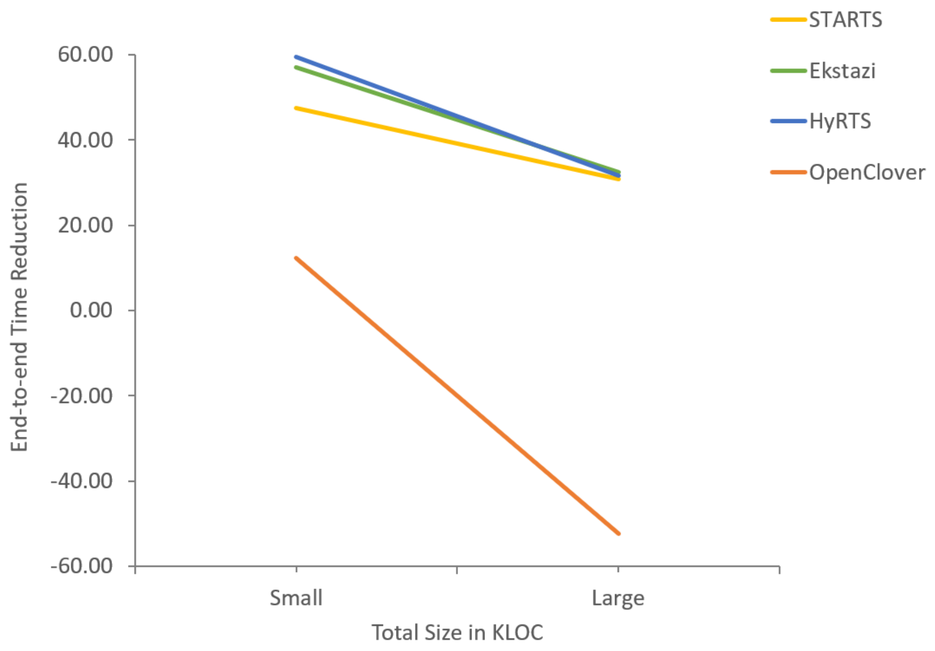

**Figure 5.14:** Two-way Interaction Between RTS Technique and Total Size in KLOC on End-to-end Time Reduction

Figure 5.14 shows the pairwise comparison result based on the end-to-end time reduction achieved by the four RTS techniques under different size in KLOC. The dots at the end of each line represents the estimated marginal means of an average of three executions of end-to-end time reduction achieved by RTS techniques when the total size in KLOC is large or small. The overall pattern is similar in that RTS techniques tend to reduce more time on the programs that have less KLOC. The p-values show that the time reduction of OpenClover is statistically lower than other techniques in both over 100 KLOC and less than 100 KLOC.

**Fault detection ability.** The parametric test for the main effect of size in KLOC on faulty detection ability was significant. But the interaction effect between size in KLOC and RTS technique on fault detection ability was not significant. The parametric between-subjects test for differences in test suite size reduction by the fault detection ability in KLOC was statistically significant (F = 14.676, df = 1, p < 0.000). The interaction effect between total size in KLOC and the fault detection ability of RTS techniques was not statistically significant (F = 2.317, df = 1.09, p = 0.13).

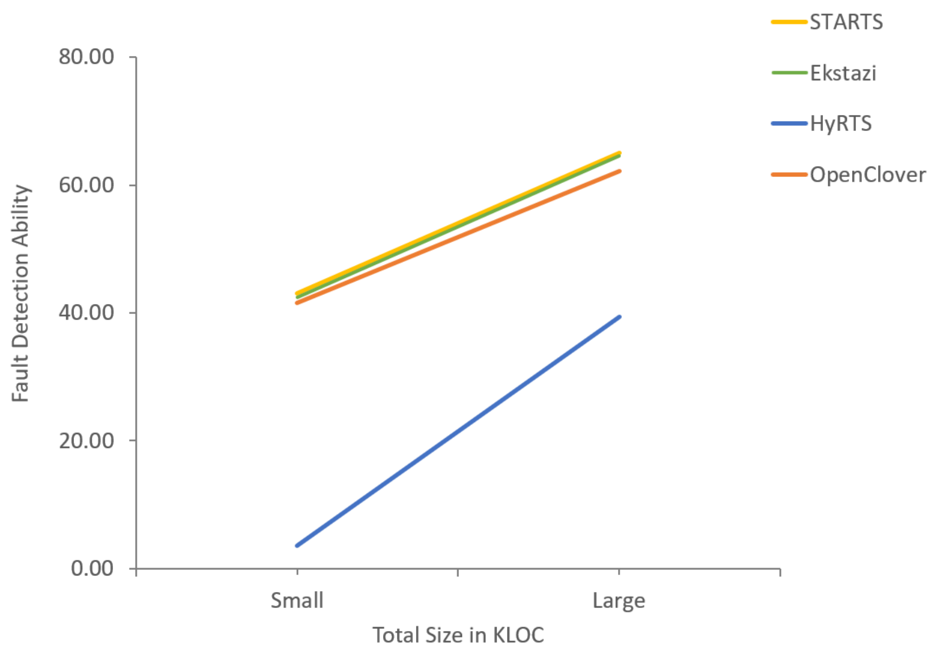

**Figure 5.15:** Two-way Interaction Between RTS Technique and Total Size in KLOC on Fault Detection Ability

Figure 5.15 depicts the pairwise comparison results with the total size in KLOC and the fault detection ability. The y-axis shows an estimated marginal means based on the fault detection abilities achieved by RTS techniques. The figure shows that the blue line that represents HyRTS is placed much lower than other lines. The significance of fault detection ability of HyRTS was 0.003 to 0.016 compared to other techniques. That means HyRTS killed statistically less mutants than STARTS, Ekstazi, and OpenClover. The yellow line for STARTS is not displayed well in Figure 5.15 because the plot values of STARS and Ekstazi are almost identical.

### 5.5.2 Total Number of Classes

**Test suite size reduction.** The non-parametric test results showed that the main effect of total number of classes on test suite size reduction was not statistically significant. The p-value for the Wald-Type statistics was 5.184339e-01. The parametric between-subjects test for difference in test suite size reduction by total number of classes was also 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).

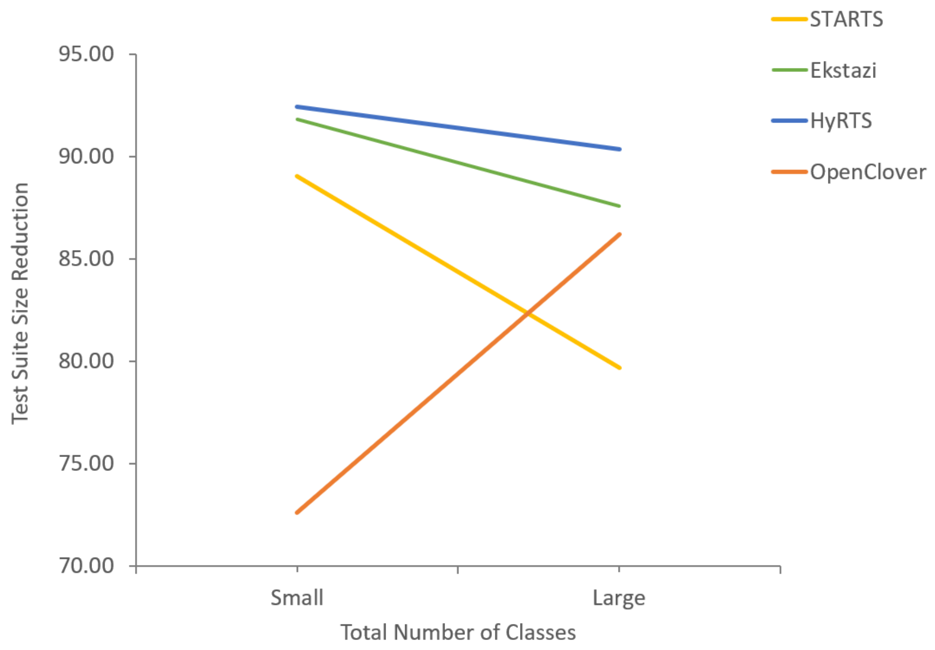

**Figure 5.16:** Two-way Interaction Between RTS Technique and Total Number of Classes on Test Suite Size Reduction

The pairwise comparison results of total number of classes and test suite size reduction is shown in Figure 5.16. The figure shows the opposite result between OpenClover and STARTS, Ekstazi, and HyRTS. OpenClover selects more test cases when the projects have a less total number of classes while the other techniques select more test cases when the projects have more classes.

Interestingly, even though both total size in KLOC and the total number of classes are metrics to measure the size of projects, the overall graphs of these two metrics regard to the test suite size-reduction appear the opposite. This is because more classes in projects do not mean more lines of code and vice versa.

**End-to-end time reduction.** OpenClover saves more time on the subjects that have fewer classes. As Figure 5.13 shows, on Commons Collection, which has the second most number of classes, OpenClover spends 93.63% longer time than running the original test suite. However, on Commons Net, one of the subjects with the least number of classes, OpenClover reduced 65.98% time. Even though Asterisk has the most number of classes, OpenClover achieved 1.52% time reduction on Asterisk. This is because OpenClover is affected by the combination of the number of classes and test cases.

The non-parametric test results showed that main effect of total number of classes and the interaction effect between total number of classes and RTS technique on the end-to-end time reduction was statistically significant. 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).

Figure 5.17 shows the result of pairwise comparison based on total number of classes and end-to-end time reduction of RTS techniques. The most noticeable difference between techniques is that the orange line that represents OpenClover achieved significantly lower end-to-end time reduction than the other techniques. The significance values also represent that the time reduction achieved by OpenClover is extremely different from STARTS, Ekstazi, and HyRTS in both small and large projects in terms of total number of classes. We also found that STARTS achieved statistically less time reduction than Ekstazi and HyRTS on the projects that have more total number of classes.

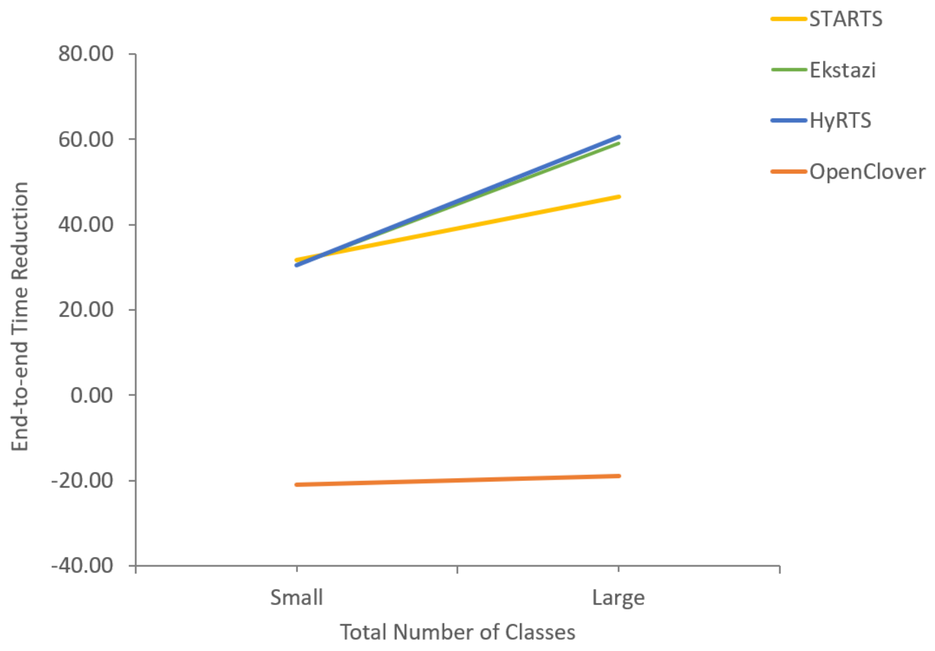

**Figure 5.17:** Two-way Interaction Between RTS Technique and Total Number of Classes on End-to-end Time Reduction

**Fault detection ability.** The non-parametric test results showed that the main effect of total number of classes was significant but the interaction effect between total number of classes and RTS technique on the fault detection ability was not statistically significant. 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.** We found that 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). Also, the interaction effect between the percentage of test classes and RTS technique on test suite size reduction was significant (F = 97.23, df = 1.37, p < 0.000).

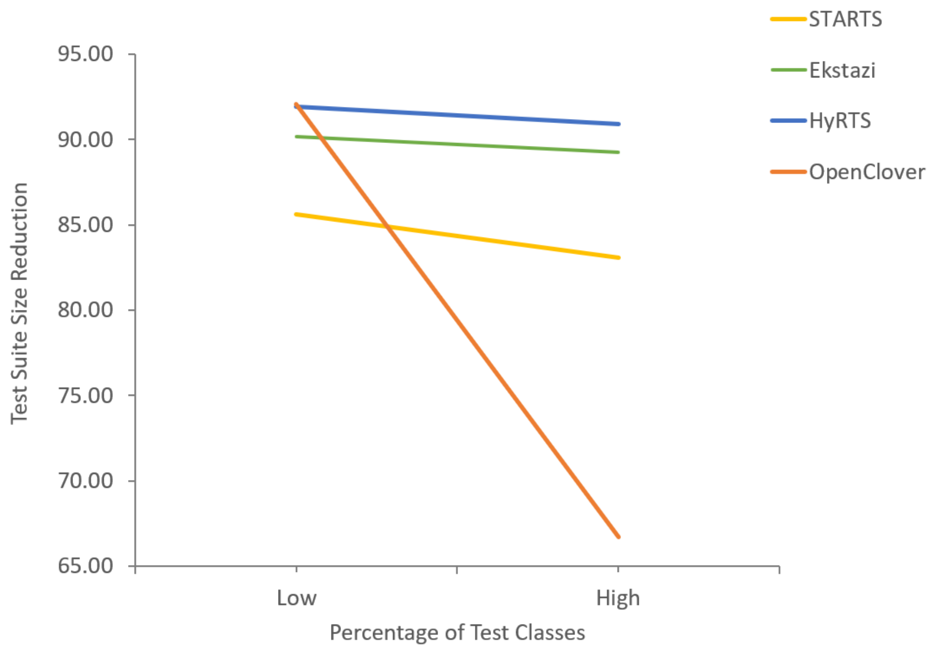

**Figure 5.18:** Two-way Interaction Between RTS Technique and Percentage of Test Classes on Test Suite Size Reduction

The parametric test result is an agreement with the non-parametric test result. The pairwise comparison based on the percentage of test classes and test suite size reduction is presented in Figure 5.18. The figure shows that RTS techniques tend to achieve higher test suite size reduction on the projects that have a lower percentage of test classes in the total number of classes. Especially, OpenClover has the most dramatic difference in test suite size reduction in the projects that have a higher percentage of test classes and the lower percentage of test classes.

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

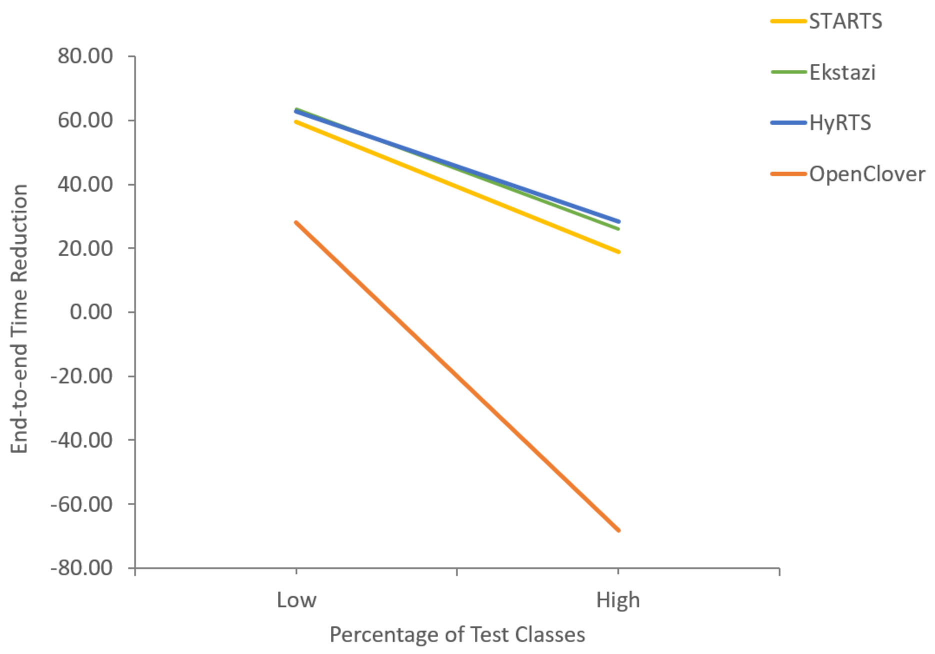

**Figure 5.19:** Two-way Interaction Between RTS Technique and Percentage of Test Classes on End-to-end Time Reduction

Figure 5.19 shows the result of pairwise comparison between the percentage of test classes in the total number of classes and the end-to-end time reduction of RTS techniques. The overall trend is similar across four RTS techniques that higher time reduction in the projects that have a lower percentage of test cases than the projects that have a higher percentage of test cases. This pattern can be explained with Figure 5.18 that RTS techniques select more test cases in the projects that has higher percentage of test classes.

**Fault detection ability.** We found that both the main effect of the percentage of test classes and the interaction effect between the percentage of test classes and RTS technique on the 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).

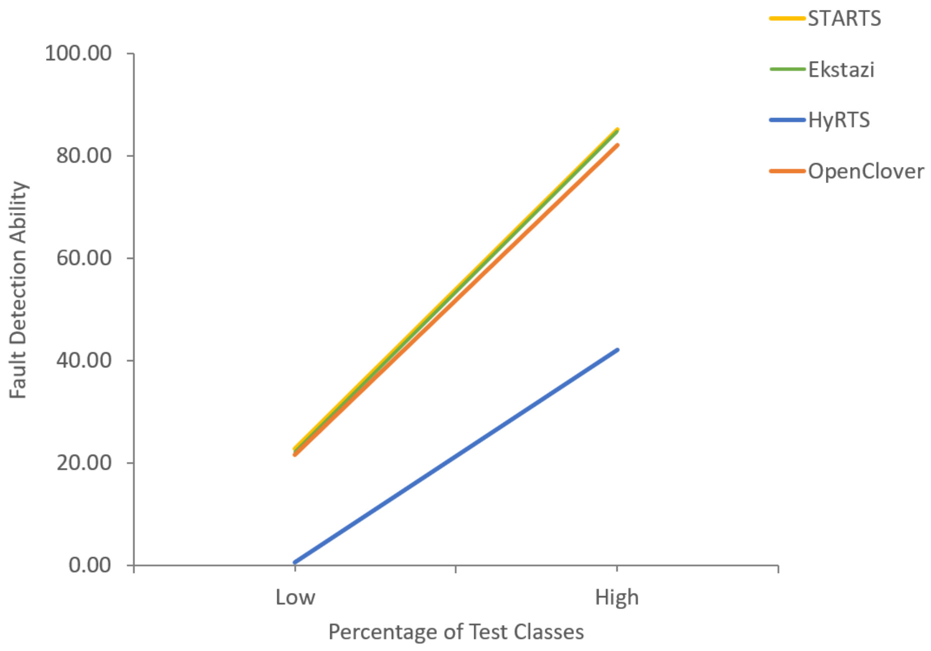

**Figure 5.20:** Two-way Interaction Between RTS Technique and Percentage of Test Classes on Fault Detection Ability

However, the parametric test indicates a statistical significance between RTS techniques where projects have a high percentage of test classes versus a low percentage of test classes. Figure 5.20 depicts the result of the pairwise comparisons between the percentage of test classes and fault detection ability. The figure shows a similar pattern between RTS techniques that higher fault detection ability on the projects that have higher test classes than lower test classes. The significance values show that HyRTS achieved statistically lower fault detection ability than other techniques in the overall projects regardless of the percentage of test classes in the project.

### 5.5.4 Percentage of Changed Classes between Revisions.

Figure 5.21 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.

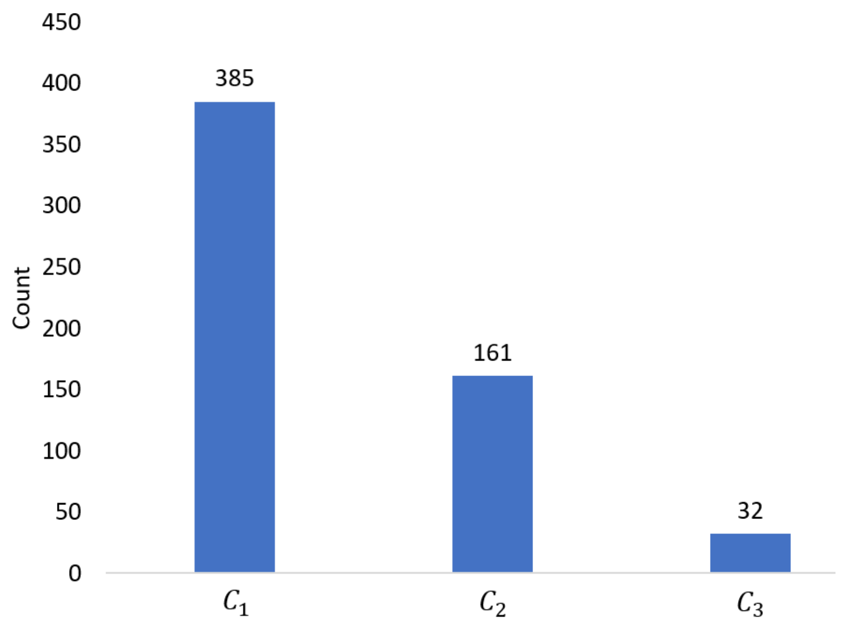

**Figure 5.21:** Distribution of Changes in Subjects

First, we considered the 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 that have multiple changed files are grouped in one category called $C_3$. We did not divide the revisions 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.** Figure 5.22 shows the relationship between the test suite reduction and the percentage of changes over the total number of files. The category $C_1$ includes the revisions that have 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. That 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 the category $C_1$, OpenClover selected and ran 10.63% more test cases that were ignored by STARTS, Ekstazi, and HyRTS. We observed that the 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 is increased. That means new test cases were added to those revisions. We confirmed that all four RTS techniques selected test cases on those revisions.

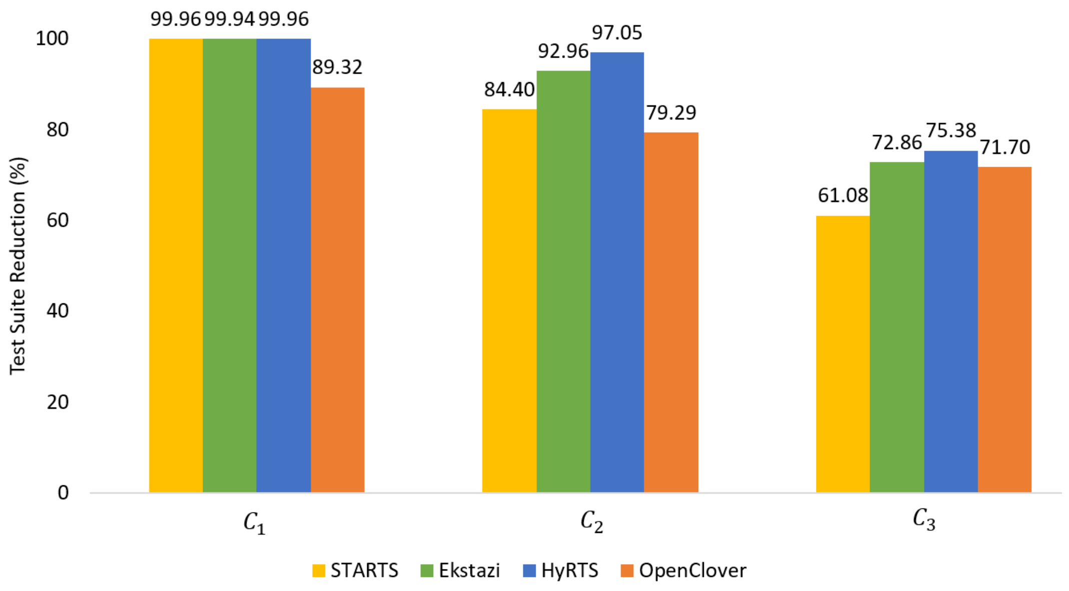

**Figure 5.22:** Number of Changed Files and Test Suite Reduction

Overall, RTS techniques run more test cases as 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, on the other hand, reduces more test cases than other RTS techniques in the revisions that have more changed files. In the category $C_3$ where there are multiple changed files, OpenClover reduces from 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. That 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 non-parametric test results showed that both the main effect of the percentage of changed classes and the interaction effect between the percentage of changed classes and RTS technique on test suite size reduction were statistically significant. 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).

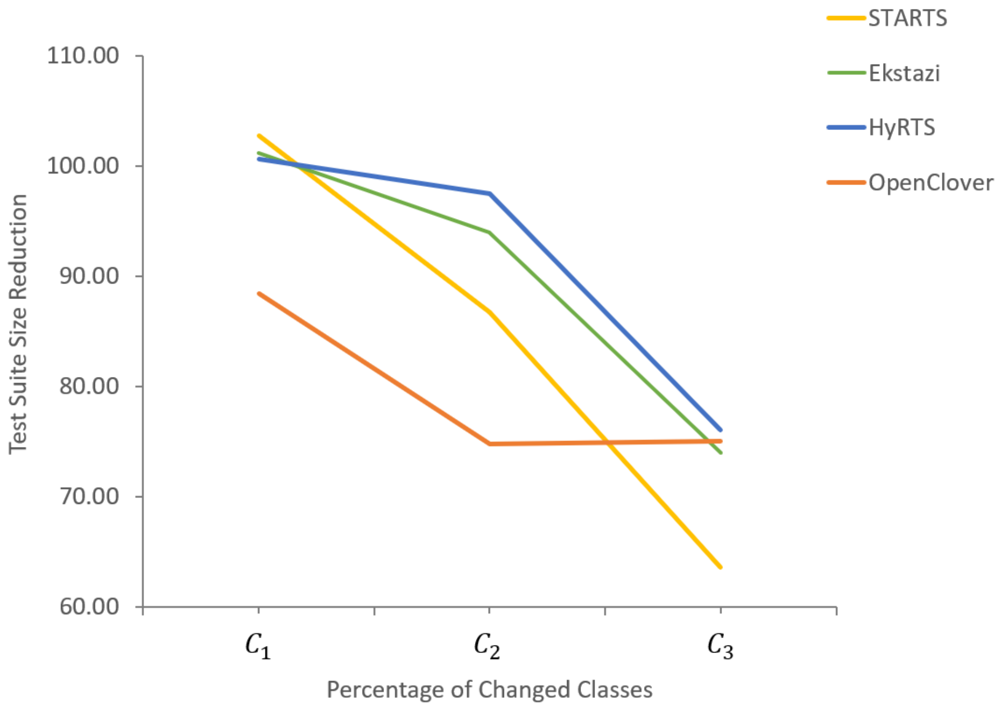

**Figure 5.23:** Two-way Interaction Between RTS Technique and Percentage of Changed Classes on Test Suite Reduction

The result of parametric test based on the percentage of changed files between revisions and the test suite size reduction is shown in Figure 5.23. The trend of the figure is similar to the bar chart in Figure 5.22 that RTS techniques select more test cases as there are more changed class files. The test suite size reductions achieved by STARTS, Ekstazi, and HyRTS are statistically not different while OpenClover achieved lower test suite size reduction than other techniques on the revisions that have no changed files. OpenClover, however, achieved higher test suite size reduction than STARTS and statistically similar test suite size reduction to Ekstazi and HyRTS in the category $C_3$.

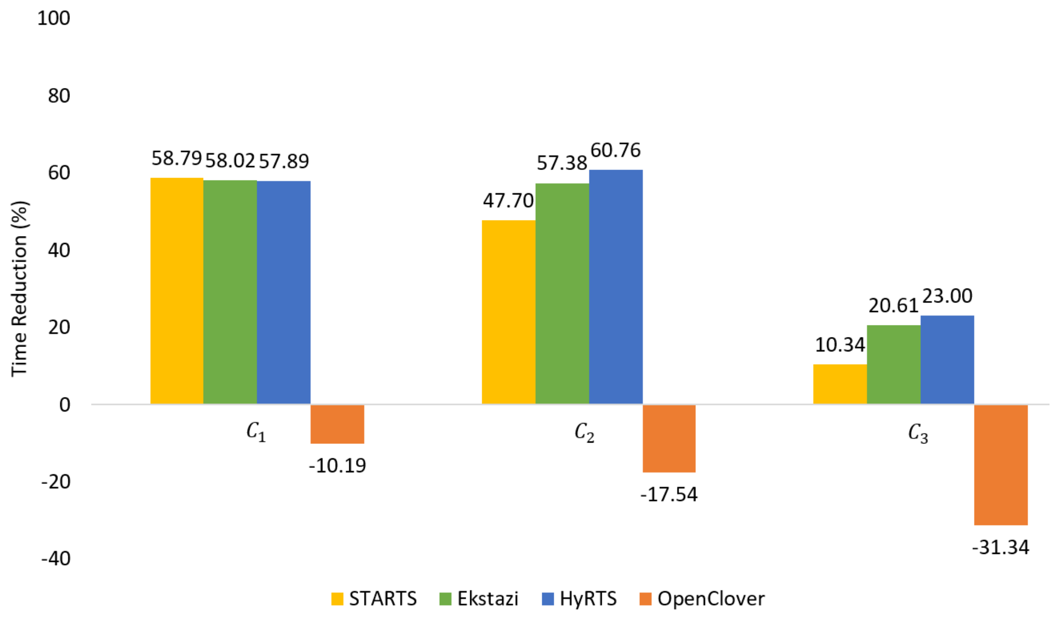

**Figure 5.24:** Number of Changed Files and Time Reduction

**End-to-end time reduction.** In Figure 5.24, we show how much time was reduced compared to running the original test suite when there are a different number of changes. Figure 5.24 shows that the result is similar to the cumulative results from the four RTS techniques (Figure 5.3) that HyRTS and OpenClover achieved the highest and lowest time reduction, 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 non-parametric test results showed that both the main effect of the percentage of changed classes and the interaction effect between the percentage of changed classes and RTS technique on the end-to-end time reduction were statistically significant. 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).

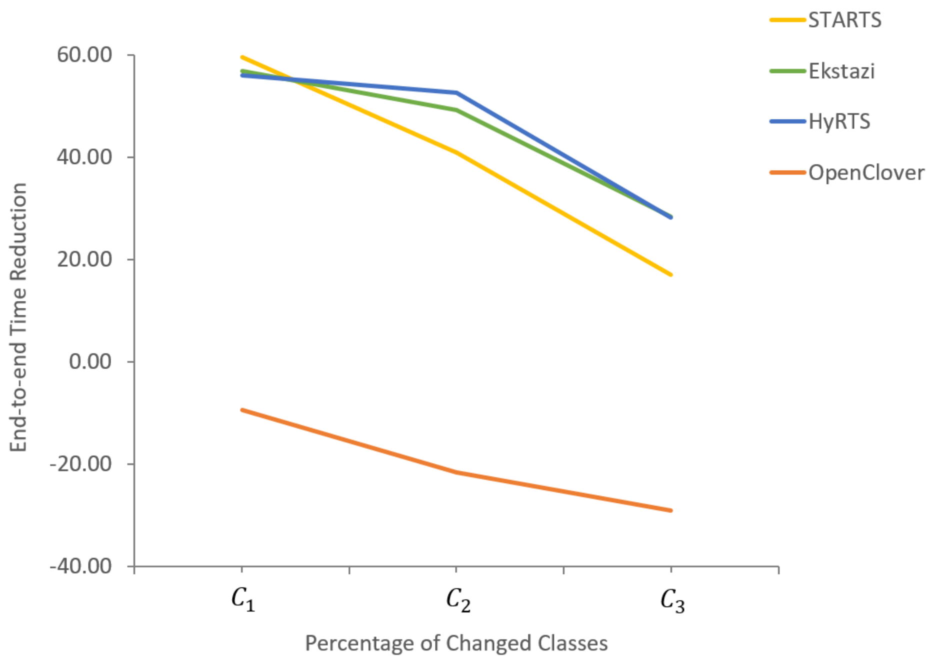

**Figure 5.25:** Two-way Interaction Between RTS Technique and Percentage of Changed Classes in Revisions on End-to-end Time Reduction

The parametric test result shows the details of the differences in time reduction between techniques. Figure 5.25 represents the result of pairwise comparison for the interaction between percentage of changed classes and the end-to-end time reduction achieved by the four RTS techniques. The figure depicts that the overall trend is similar to the test suite size reduction as Figure 5.25 shows. The figure also shows that OpenClover achieved significantly lower time reduction than other techniques in any category.

**Fault detection ability.** The non-parametric test results showed that the main effect of the percentage of changed classes was statistically significant, while the interaction effect between the percentage of changed classes and RTS technique on the fault detection ability was not statistically significant. 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). The interaction effect between the percentage of changed classes and RTS technique on the fault detection ability was not significant (F = 1.54, df = 1.09, p = 0.22).

## 5.6 Discussion

Our empirical study results are in agreement with results from the studies conducted by other researchers [10, 18]. Zhang [10] shows that 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. [18] demonstrates that not every RTS technique saves time. Running OpenClover sometimes took a longer time than running the original test suite [18]. Our empirical study results demonstrate similar results. Overall, HyRTS selects the least number of test cases among the four techniques (Section 5.1), and the highest mean value of the end-to-end time reduction is achieved by HyRTS at 59.40% (Section 5.2). OpenClover spent 93.63% longer time than running the original test suite of the Commons Collection subject (Section 5.5.2).

## 5.7 Threats to Validity

**Internal Validity**

These threats are related to the implementations used in the study. We followed the manuals that the 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 rest of 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). We assumed that STARTS, Ekstazi, and HyRTS compute smart checksums in the same way. STARTS reuses smart checksum computation code from Ekstazi. Second, OpenClover has a known bug [18] that selects any test cases with certain annotations. Furthermore, Zhu et al. [18] show that OpenClover has several safety issues on finding test dependencies.

### External Validity

We used five publicly available open source projects as subjects, which can lead to a lack of generalizability to other programs. To reduce the threat, we used different programs in terms of the number of test cases and the number of lines of code. We selected different revisions than 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 [48] have shown the usefulness of mutation faults in software engineering experiments.

### Construct Validity

There are other metrics that can be used to measure time reduction, such as test case selection time, test execution time, and end-to-end test time. The use of the metric end-to-end test time can be a threat to construct validity. However, we used it because Ekstazi, HyRTS, and OpenClover do not output only the test selection time or only the test execution time. Having a gold standard is necessary for correctly using safety and precision violation formulas. If neither tool is safe (or precise), the results can be misleading.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

Regression testing is an essential but expensive software engineering activity. Researchers have developed many regression test selection techniques to reduce the regression testing time. Researchers have empirically evaluated RTS techniques, but each of these evaluations were performed under different conditions.

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, precision, and fault detection ability. We also investigated program factors that affect the performance of the RTS techniques, which can help practitioners determine the most appropriate technique for their specific requirements. To achieve this goal, we ran RTS techniques and analyzed the results to answer our research questions.

We found that the average test suite size reduction varies 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 three 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 four RTS techniques was 40.49%. HyRTS was the least safe RTS technique with respect to both STARTS and Ekstazi, while the safety violations achieved by STARTS, Ekstazi, and OpenClover were statistically not different. As OpenClover selected the largest number of test cases, it had the highest precision violation. HyRTS achieved the lowest fault detection ability while STARTS, Ekstazi, and OpenClover killed as many mutants as running the original test suite.

We used total program size in KLOC, the total number of classes, the percentage of test classes in the total number of classes, and the percentage of changed classes as program characteristics to explain which of them influence the performance of RTS techniques. We found that OpenClover has an opposite pattern from other techniques related to the test suite size reduction. For example, STARTS, EKstazi, and HyRTS achieve higher test suite size reduction in the subjects that have over 100 KLOC than less than 100 KLOC, while OpenClover selects more test cases in the subjects that have over 100 KLOC. OpenClover in general selects more test cases than other techniques but selected less test cases on the programs that have fewer test classes. HyRTS killed relatively fewer mutants than other techniques regardless of the number of changed files.

We also conducted statistical tests to analyze the empirical study results. These results often show similar results that we can expect from the plots of the metrics achieved with RTS techniques while it shows even further detailed information. For example, the Bonferroni test results show that the test suite size reduction achieved by STARTS is statistically lower than Ekstzi and HyRTS. Statistical test results also provide analyses that were not intuitively shown on the plots. Because there are many outliers in safety violations, it is not easy to compare the techniques. 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 out of the four techniques, especially when the program size is over 100 KLOC. OpenClover should be avoided if an expectation of the RTS technique is related to the end-to-end time reduction.

## 6.2 Future Work

Our empirical evaluation involves one static and three dynamic RTS techniques. 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, which emphasize the selection of fewer test cases as the main objective rather than considering the safety of test selection [8]. 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.

In our empirical study, we used five open-source Java projects as subjects. A possible extension is to use more subjects and revisions to get a better general conclusions. 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. The future work can measure the generality of RTS techniques.

We considered four factors while determining their impact on the performance of RTS techniques. There are additional factors that can be worth investigating. For example, our result( 5.5.4) shows that the type of changes affects the 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 can lead to interesting results.

# Bibliography

[1] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-Scale Continuous Testing. In *39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 233–242. IEEE, 2017.

[2] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[3] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing Test Cases for Regression Testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[4] Gregg Rothermel and Mary Jean Harrold. Empirical Studies of A Safe Regression Test Selection Technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.

[5] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *International Conference on Software Maintenance*, pages 34–43. IEEE, 1998.

[6] Owolabi Legunsen, August Shi, and Darko Marinov. STARTS: STAtic Regression Test Selection. In *32nd International Conference on Automated Software Engineering (ASE)*, pages 949–954. IEEE, 2017.

[7] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[8] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. Reflection-Aware Static Regression Test Selection. *Conference on Object-Oriented Programming, Systems, Languages, and Application*, 3:1–29, 2019.

[9] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[10] Lingming Zhang. Hybrid Regression Test Selection. In *40th International Conference on Software Engineering (ICSE)*, pages 199–209. IEEE, 2018.

[11] Openclover. https://openclover.org/ (Accessed 2019-04-19).

[12] Filippos I. Vokolos and Phyllis G. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Reliability, Quality and Safety of Software-Intensive Systems*, pages 3–21. Springer, 1997.

[13] Quinten David Soetens and Serge Demeyer. ChEOPSJ: Change-Based Test Optimization. In *Euromicro Conference on Software Maintenance and Reengineering*, pages 535–538, 2012.

[14] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing Trade-Offs in Test-Suite Reduction. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256, 2014.

[15] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 22–31, 2008.

[16] John Bible, Gregg Rothermel, and David S Rosenblum. A Comparative Study of Coarse-and Fine-Grained Safe Regression Test-Selection Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):149–183, 2001.

[17] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[18] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. A Framework for Checking Regression Test Selection Tools. In *41st International Conference on Software Engineering (ICSE)*, pages 430–441. IEEE, 2019.

[19] Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[20] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. Change-Based Test Selection: An Empirical Evaluation. *Empirical Software Engineering*, 21(5):1990–2032, 2016.

[21] Rafaqut Kazmi, Dayang N.A. Jawawi, Radziah Mohamad, and Imran Ghani. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Computing Surveys (CSUR)*, 50(2):1–32, 2017.

[22] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive Test Selection. In *41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE, 2019.

[23] Ahmet Celik, Young Chul Lee, and Milos Gligoric. Regression Test Selection for TizenRT. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 845–850, 2018.

[24] Mary Jean Harrold and Mary Lou Soffa. Interprocedual Data Flow Testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):158–167, 1989.

[25] Hareton KN Leung and Lee White. A Study of Integration Testing and Software Regression at the Integration Level. In *Conference on Software Maintenance*, pages 290–301. IEEE, 1990.

[26] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class Firewall, Test Order, and Regression Testing of Object-oriented Programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[27] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[28] Yih-Farn Chen, David S Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *16th International Conference on Software Engineering*, pages 211–220. IEEE, 1994.

[29] David Rosenblum and Gregg Rothermel. A Comparative Study of Regression Test Selection Techniques. In *2nd International Workshop on Empirical Studies of Software Maintenance*. IEEE Computer Society Press, 1997.

[30] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[31] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression Test Selection for Java Software. *ACM Sigplan Notices*, 36(11):312–326, 2001.

[32] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling Regression Testing to Large Software Systems. *ACM SIGSOFT Software Engineering Notes*, 29(6):241–251, 2004.

[33] Feng Lin, Michael Ruth, and Shengru Tu. Applying Safe Regression Test Selection Techniques to Java Web Services. In *International Conference on Next Generation Web Services Practices*, pages 133–142. IEEE, 2006.

[34] Guoqing Xu and Atanas Rountev. Regression Test Selection for AspectJ Software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 65–74. IEEE, 2007.

[35] Lee J White. A Firewall Approach for Regression Testing of Object-Oriented Software. *10th Annual Software Quality Week*, 1997.

[36] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *19th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.

[37] Jianjun Zhao, Tao Xie, and Nan Li. Towards Regression Test Selection for AspectJ Programs. In *2nd Workshop on Testing Aspect-Oriented Programs*, pages 21–26, 2006.

[38] Yanping Chen, Robert L Probert, and D Paul Sims. Specification-Based Regression Test Selection with Risk Analysis. In *Conference of the Centre for Advanced Studies on Collaborative Research*, page 1, 2002.

[39] Quinten David Soetens, Serge Demeyer, and Andy Zaidman. Change-Based Test Selection in the Presence of Developer Tests. In *17th European Conference on Software Maintenance and Reengineering*, pages 101–110. IEEE, 2013.

[40] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. SPIRITuS: A SimPle Information Retrieval regressIon Test Selection approach. *Information and Software Technology*, 99:62–80, 2018.

[41] Maral Azizi and Hyunsook Do. ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development. In *29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 144–154. IEEE, 2018.

[42] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating Regression Test Selection Opportunities in a Very Large Open-source Ecosystem. In *29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 112–122. IEEE, 2018.

[43] Gregg Rothermel and Mary Jean Harrold. Experience with Regression Test Selection. *Empirical Software Engineering*, 2(2):178–188, 1997.

[44] Ben Fu, Sasa Misailovic, and Milos Gligoric. Resurgence of Regression Test Selection for C++. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 323–334. IEEE, 2019.

[45] Pavan Kumar Chittimalli and Mary Jean Harrold. Regression Test Selection on System Requirements. In *1st India Software Engineering Conference*, pages 87–96, 2008.

[46] Mary Jean Harrold, James A Jones, and Gregg Rothermel. Empirical Studies of Control Dependence Graph Size for C Programs. *Empirical Software Engineering*, 3(2):203–211, 1998.

[47] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, 2001.

[48] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *27th International Conference on Software Engineering*, pages 402–411. ACM, 2005.

[49] Clover 4 Test Optimization. https://confluence.atlassian.com/clover/about-test-optimization-169119919.html/ (Accessed 2019-04-19).

[50] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[51] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An Empirical Study on Object-Oriented Metrics. In *International software metrics symposium*, pages 242–249. IEEE, 1999.

[52] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010.

[53] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? An empirical analysis of Java

mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, 2018.

[54] Mohammed Nayef Al-Refai. *Towards Model-Based Regression Test Selection*. PhD thesis, Colorado State University. Libraries, 2019.

[55] Kimihiro Noguchi, Yulia R Gel, Edgar Brunner, and Frank Konietschke. nparLD: An R Software Package for the Nonparametric Analysis of Longitudinal Data in Factorial Experiments. *Journal of Statistical Software*, 50(12), 2012.

[56] Michael G Akritas and Edgar Brunner. A Unified Approach to Rank Tests for Mixed Models. *Journal of Statistical Planning and Inference*, 61(2):249–277, 1997.

[57] Abhik Ghosh, Abhijit Mandal, Nirian Martín, and Leandro Pardo. Influence Analysis of Robust Wald-type Tests. *Journal of Multivariate Analysis*, 147:102–126, 2016.

[58] John Ludbrook. Multiple Comparison Procedures Updated. *Clinical and Experimental Pharmacology and Physiology*, 25(12):1032–1037, 1998.

[59] YH Chan. Biostatistics 102: Quantitative Data-Parametric & Non-Parametric Tests. *blood Press*, 140(24.08):79, 2003.

[60] Thom Baguley. An Introduction to Sphericity. *Retrieved September*, 1:2008, 2004.

[61] Open clover user guide. http://openclover.org/doc/manual/4.2.0/ant--coverage-recorders.html (Accessed 2019-06-03).

[62] PITest. https://pitest.org/ (Accessed 2019-07-08).