BACHELOR THESIS

Phu Binh Dang

# Reducing Testing Costs by Applying Regression Test Selection

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Phu Binh Dang

# Reducing Testing Costs by Applying Regression Test Selection

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Olaf Zukunft
Supervisor: Prof. Dr. Stefan Sarstedt

Submitted on: 27. February 2023

**Phu Binh Dang**

**Title of Thesis**

Reducing Testing Costs by Applying Regression Test Selection

**Keywords**

Regression Test Selection, Software Testing

**Abstract**

Automated regression testing is an established and well-proven technique to ensure software quality. It consists of a set of tests that are re-run on every code change. With a large test suite, it can be very time and resource consuming. Researchers propose Regression Test Selection (RTS) tools as a solution for tackling this problem. RTS tools aim to select and run only tests that are impacted by code changes. The goal of this work is to evaluate Java-based RTS tools. The RTS tools are evaluated on a real-world Java project containing 14 000 LoC. The evaluation is based on four metrics: end-to-end time reduction, safety and precision violation, and fault detection ability. During implementation of this thesis, three out of five tools turned out to be incompatible with software under test or had technical problems, for instance, due to discontinued maintenance of the tools. Because of these issues, only STARTS and OpenClover are evaluated. The findings show STARTS saves 40.5% of the testing time compared to rerunning all tests on average. This time saving is achieved mainly from the integration tests. Conversely, OpenClover cannot save any time in any revision. It needs even 7% more time than rerunning all tests. The sets of selected tests by OpenClover are always larger than that of STARTS, thus OpenClover rarely misses a test that is selected by STARTS. Both tools are as good as rerunning all tests in detecting faults. Their average mutation coverages are 55%.

**Phu Binh Dang**

**Thema der Arbeit**

Reduzierung der Testkosten durch Anwendung der Regressionstestselektion

**Stichworte**

Regressionstestselektion, Testen von Software

**Kurzzusammenfassung**

Automatisiertes Regressionstesten ist eine etablierte und bewährte Methode zur Sicherung von Softwarequalität. Sie besteht aus einer Reihe von Tests, die bei jeder Codeänderung erneut durchgeführt werden. Bei einer großen Testreihe kann dies sehr zeit- und ressourcenaufwendig sein. Forscher schlagen Regression-Test-Selection-Tools (RTS Tools) als Lösung für dieses Problem vor. RTS-Tools zielen darauf ab, nur Tests auszuwählen und auszuführen, die von Codeänderungen betroffen sind. Das Ziel dieser Arbeit ist es, Java-basierte RTS-Tools zu evaluieren. Die RTS-Tools werden anhand eines realen Java-Projekts mit 14 000 Codezeilen evaluiert. Die Bewertung basiert auf vier Metriken: Zeitreduktion (end-to-end time reduction), Sicherheits- und Präzisionsverletzungen (safety and precision violations) sowie die Fähigkeit zur Fehlererkennung. Während der Durchführung dieser Arbeit stellte sich heraus, dass drei von fünf RTS-Tools nicht mit der zu testenden Software kompatibel waren oder technische Probleme aufwiesen, z.B. aufgrund eingestellter Wartung der Tools. Aufgrund dieser Probleme wurden nur STARTS und OpenClover evaluiert. Die Ergebnisse zeigen, dass STARTS im Vergleich zur Wiederholung aller Tests im Durchschnitt 40,5% der Testzeit reduziert. Diese Zeitreduktion wird hauptsächlich bei den Integrationstests erzielt. Im Gegensatz dazu kann OpenClover in keiner einzigen Revision Zeit reduzieren. Es benötigt sogar 7% mehr Zeit als die Wiederholung aller Tests. Die Menge der ausgewählten Tests von OpenClover ist immer größer als die von STARTS. Daher lässt OpenClover selten einen Test aus, der von STARTS ausgewählt wurde. Beide Tools sind so gut wie die Wiederholung aller Tests bei der Erkennung von Fehlern. Ihre durchschnittliche Mutationsabdeckung beträgt 55%.

Dedication

To my family.

# Contents

**List of Figures** **viii**

**List of Tables** **ix**

**1 Introduction** **1**

1.1 Problem and motivation . . . 1

1.2 Goal . . . 2

1.3 Structure . . . 2

**2 Related Work** **3**

2.1 RTS approaches . . . 3

2.1.1 Firewall approach . . . 3

2.1.2 Graph walking approach . . . 3

2.1.3 Code-base approach . . . 4

2.1.4 Model-based approach . . . 5

2.1.5 Approximation . . . 6

2.2 Comparison of RTS techniques . . . 6

**3 Analysis** **8**

3.1 Subject selection . . . 8

3.2 Evaluation metrics . . . 9

3.2.1 End-to-end time reduction . . . 9

3.2.2 Safety violation . . . 10

3.2.3 Precision violation . . . 10

3.2.4 Fault detection ability . . . 11

3.3 Reasoning about suitable RTS tools . . . 12

3.3.1 Metrics from empirical studies . . . 12

3.3.2 Compatibility and accessibility . . . 18

3.3.3 Selected RTS tools . . . 20

**4 Design** **23**
4.1 Selecting and running tests . . . 23
4.1.1 STARTS . . . 23
4.1.2 OpenClover . . . 25
4.1.3 Running all tests . . . 26
4.2 Mutation testing . . . 26
4.3 Revisions . . . 28

**5 Implementation** **29**
5.1 Experiment requirements . . . 29
5.2 Selecting and running tests . . . 29
5.2.1 STARTS . . . 30
5.2.2 OpenClover . . . 31
5.2.3 Running all tests . . . 31
5.3 Mutation testing . . . 32
5.4 Revisions . . . 33
5.5 Calculate metrics . . . 34

**6 Evaluation** **35**
6.1 End-to-end time reduction . . . 35
6.2 Safety and precision violations . . . 38
6.3 Fault detection ability . . . 39
6.4 Threats to validity . . . 39

**7 Generalization** **41**

**8 Conclusion and future work** **42**
8.1 Conclusion . . . 42
8.2 Future work . . . 43

**Bibliography** **44**
Declaration of Authorship . . . 47

# List of Figures

2.1 A edited version by Hyunsook Do [1] of the example originating from [2] . 4

3.1 Illustration of safety and precision violations using Venn diagram . . . . . 11
3.2 End-to-end time reduction from [3] . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Fault Detection Ability from [3] . . . . . . . . . . . . . . . . . . . . . . . . 14
3.4 Safety violation from [3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.5 Precision violation from [3] . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.6 A simple Java project with four classes . . . . . . . . . . . . . . . . . . . . 19

4.1 Activity diagram of STARTS . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2 Activity diagram of OpenClover . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Activity diagram of mutation testing . . . . . . . . . . . . . . . . . . . . . 27

6.1 End-to-end time reduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2 Execution time of unit and integration tests . . . . . . . . . . . . . . . . . 36
6.3 Effect of integration tests on STARTS's time reduction . . . . . . . . . . . 37
6.4 OpenClover's safety and precision violations . . . . . . . . . . . . . . . . . 38
6.5 Fault detection ability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

# List of Tables

3.1 Metrics comparison between SISI and SOSI . . . . . . . . . . . . . . . . . . 9
3.2 Tool rankings based on end-to-end time reduction . . . . . . . . . . . . . . . 13
3.3 Tool rankings based on time reduction and fault detection ability . . . . . 14
3.4 Metrics comparison between STARTS, Ekstazi, and FLiRTS 2 . . . . . . . . 17
3.5 Final tool performance rankings . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.6 Tool compatibility with JUnit 5 . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.7 Tool performance rankings, compatibility, and accessibility . . . . . . . . . 20

# 1 Introduction

## 1.1 Problem and motivation

When introducing a new feature to an existing software, it is essential that the new feature is completely compatible with the existing features. To ensure that, all the tests of the software need to be rerun. This process is known as regression testing. It is a widespread practice in the software industry [4]. It is understandable that the number of tests will accumulate over time and at some point will become very costly to execute them entirely. subshell [1] is also facing this problem, as evidenced by the testing phase in one of its projects can last up to 7 hours. For software developers, regression testing can reduce their productivity, since it prolongs the waiting time for feedback on their code changes. In case of subshell, developers can only see whether their code changes break any tests on the next day. For software companies, regression testing uses numerous computing resources and lowers their rates of software releases. Regression testing process usually involves setting up test runs, monitoring testing results, and maintaining testing resources, which also cause costs that are often overlooked [1].

To reduce the cost of regression testing by sacrificing quality without sacrificing quality too much, researchers proposed three approaches: test case prioritization (TCP), regression test selection (RTS), and test suite minimization (TSM) [1]. Rearranging the order of tests according to factors such as finding errors as soon as possible and targeting areas that are more prone to faults first, is known as TCP. TSM aims to identify the smallest subset of tests that provide the same test coverage as the original tests. The goal of RTS is to select relevant tests that are impacted by the code changes [3]. RTS tools have shown their effectiveness in decreasing testing costs in empirical studies [3], [5], [6].

[1]https://subshell.com/

## 1.2 Goal

The goal of this thesis is to evaluate the performance of recent Java-based RTS tools. The RTS tools are STARTS, Ekstazi, HyRTS, OpenClover, and FLiRTS 2. RTS tool performance is shown through four metrics: end-to-end time reduction, safety and precision violations, and fault detection ability. To have reliable data for comparing the metrics, the RTS tools will be applied to a subshell's Java project. Since the project has specific technical requirements, RTS tools need to be checked for compatibility with the project. Furthermore, the RTS tools and their dependencies should also be easily accessible. In case of positive experiment results, it will be a reasonable suggestion for subshell to adopt RTS tools, which can reduce testing costs in its development process.

## 1.3 Structure

The structure of the thesis is as follows. Chapter 2 summarizes important results from works related to RTS. Chapter 3 focuses on selecting a suitable experiment object and the three best suitable RTS tools. Chapter 4 outlines an experiment that provides reliable data for comparing the tools. Chapter 5 describes how the experiment is conducted. Chapter 6 compares the tools based on the experiment results. Chapter 7 discusses about the meaning of the findings in this thesis for software companies. Chapter 8 draws conclusion and gives directions for further research .

# 2 Related Work

Many RTS techniques or tools have evolved over the years. They are categorized by experts in approaches. Whereas the term "RTS approaches" often refer to the basic ideas of solving selecting tests regressively problem, the term "RTS techniques" or "RTS tools" are results after implementing those ideas. In many publications, RTS techniques and tools are interchangeable. In most cases, the tools are publicly available in the form of a code artifact such as a jar file or a plugin for a build system. This chapter starts with summarizing main ideas of popular tools in their category. At the end, the most notable comparison of state-of-the-art RTS tools [3] will be reviewed in brief.

## 2.1 RTS approaches

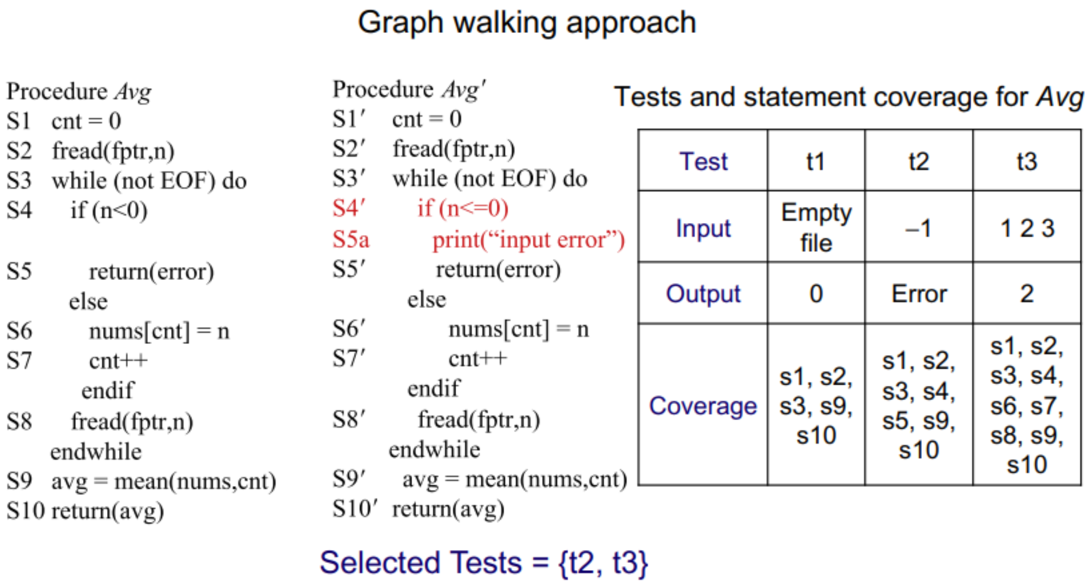
Researchers have proposed many approaches and taxonomies for them. The following categories are combined from [1], [3], [6], [7].

### 2.1.1 Firewall approach

Leung and White [8] introduced the concept of a firewall to reduce testing costs at the integration level. A firewall is used to separate modules. Inside the firewall are modules that need to be retested before integration. The firewall is built on direct interactions of unmodified modules with modified modules. The empirical study of this approach on 32 modules which contain 550 lines of Pascal code showed encouraging results.

### 2.1.2 Graph walking approach

From 1994 to 1998 Rothermel and Harrold developed a family of regression test selection which utilizes graphs. *Dejavu* [9] is a tool that traverses through control flow graphs to select tests. A control graph is a directed graph, its nodes and edges are statements and the flow control of a program, respectively. Given the set of tests $T$, the original version P of the program and the modified version $P'$. Firstly, *Dejavu* builds the control graph for $P$ and $P'$. It then performs depth-first traversals on those graphs synchronously. At each step during the traversals, it checks if the two currently visited nodes ($N$ for $P$ and $N'$ for $P'$) are lexicographically different. If yes, it will select all the tests from T that cover node N. The following example illustrates why *Dejavu* selects the test $t2, t3$.



```
Procedure Avg
S1  cnt = 0
S2  fread(fptr,n)
S3  while (not EOF) do
S4    if (n<0)

S5      return(error)
      else
S6      nums[cnt] = n
S7      cnt++
      endif
S8    fread(fptr,n)
    endwhile
S9  avg = mean(nums,cnt)
S10 return(avg)
```

```
Procedure Avg'
S1'   cnt = 0
S2'   fread(fptr,n)
S3'   while (not EOF) do
S4'     if (n<=0)
S5a       print("input error")
S5'       return(error)
        else
S6'       nums[cnt] = n
S7'       cnt++
        endif
S8'     fread(fptr,n)
      endwhile
S9'   avg = mean(nums,cnt)
S10'  return(avg)
```

Tests and statement coverage for *Avg*

| Test | t1 | t2 | t3 |
|---|---|---|---|
| Input | Empty file | –1 | 1 2 3 |
| Output | 0 | Error | 2 |
| Coverage | s1, s2, s3, s9, s10 | s1, s2, s3, s4, s5, s9, s10 | s1, s2, s3, s4, s6, s7, s8, s9, s10 |

Figure 2.1: A edited version by Hyunsook Do [1] of the example originating from [2]

*Avg'* is the modified version of *Avg*. Test coverage information is collected and analyzed before traversals. When the traversals are in the fourth step, they are visiting nodes $S4$ and $S4'$.

*Dejavu* reduced on average by 42% of the testing time, from 14 min, 27 sec to 8 min, 21 sec. The result is drawn from an experiment on seven C programs that contain 138 to 516 lines of code.

### 2.1.3 Code-base approach

Both static and dynamic RTS approaches aim to filter out irrelevant tests for a change by analyzing the source code of a program.

### Static code analysis

For the analysis, techniques in this group use the test dependencies information at compile time. Static RTS techniques are studied at different levels, from fine-granular(basic-block level) to coarse-granular (file level) [10]. Legunsen et al. [11] compared some variants of class- and method-level static RTS. The conclusion states that class-level static RTS is equally effective as class-level dynamic RTS Ekstazi while method-level static RTS performed poorly. According to [11], static RTS should be preferred over dynamic RTS for systems with long-running tests, non-determinism, or real-time constraints. STARTS[12], the state-of-the-art technique in this category, analyzes code at class-level. STARTS will be discussed in more detail in section 3.3.3.

### Dynamic code analysis

Dynamic RTS techniques require code changes between revision and test dependencies information at runtime. This means the test dependencies are collected by running the tests on the old code revision. To select a subset of tests, the techniques then analyze how the code changes affect the test dependencies [11]. One of the promising dynamic RTS tools is Ekstazi, which is adopted in real projects. While the benefit of dynamic RTS analyzing code at finer granularity is lower cost, the advantage of RTS working at coarser granularity is selecting tests more precisely. With an aim to combine those strengths, Zhang [10] created HyRTS, a hybrid RTS tool that works at multiple granularities. Another tool that utilizes dynamic code analysis is OpenClover[1]. In section 3.3.3, OpenClover will be discussed in more depth.

### 2.1.4 Model-based approach

UML (Unified Modeling Language) model-based approaches are studied in [13], [14], [15], [5]. Their data sources are structural and behavioral diagrams (e.g., activity, state, and sequence diagrams). It is practical for projects that are employing model-driven development methodologies [15]. The models represent systems at high-level abstraction, therefore, cannot fully trace the links between models and coverage-related execution traces from test cases. FLiRTS [15] improves this shortcoming by enabling the automatic refinement of abstract UML models. It utilizes fuzzy logic on UML sequence and activity

[1]https://openclover.org

diagrams. Behavioral diagrams are often insufficient or not available in many software artifacts. To tackle this problem, the authors of FLiRTS introduced FLiRTS 2 [5] which uses class diagrams instead. Because class diagrams are the most widely used UML diagrams and can be automatically generated. In terms of safety and precision, FLiRTS 2 are close to state-of-the-art RTS techniques (STARTS, Ekstazi), but need less time.

### 2.1.5 Approximation

Data-driven approaches [16], [7], [6] try to predict if a test passes or fails using a rich history of test execution and commit logs. As if the predictions are trustable, only tests that are predicted to be failed need to be run. [16], [6] use basic machine learning techniques to train a classifier that can give test verdicts. In [7], statistical models are employed to estimate the failure probability of tests. A test will be executed if its failure probability is greater than a selected threshold. The technique in Machalica et al. [6] helped Facebook reduces the total infrastructure cost by half while detecting over 99.9% of faulty changes. At Google, the proposed technique in [7] saved 15–30% of compute time while reporting 99% of buggy pull requests. At Microsoft, the evaluation of FastLane [16] on a large-scale email and collaboration platform service (O365) showed a shortening of testing time by 18,04% while predicting test outcomes with 99,99% accuracy. Such techniques can be independent of programming languages and suitable for very large software systems, but they require large test execution data and rich commit history.

## 2.2 Comparison of RTS techniques

Shin et al. [3] did a comparison study of four RTS techniques on open-source Java projects. Two of them, STARTS [12] and Ekstazi [17], are state-of-the-art techniques [18]. Two other techniques are HyRTS [10] and OpenClover. The four techniques are applied to five Java projects with sizes ranging from 16 to 204 thousand (Kilo) Lines of Code (KLOC). The results are mainly expressed and compared through five evaluation metrics: Test Suite Size Reduction, Safety Violation, Precision Violation, End-to-end Test Time Reduction, Fault Detection Ability. These metrics will be explained in more detail in section 3.2. The result shows:

- The average reduction in test suite size ranges from 86.14% to 98.13%. HyRTS achieved the greatest reduction, followed by Ekstazi, STARTS, and OpenClover.

In this respect, STARTS, Ekstazi, and HyRTS perform better on projects with over 100 KLOC.

- Four techniques reduce end-to-end time by an average of 40,49%.
- HyRTS is the least safe technique. Statistically equivalent safety violations were obtained by STARTS, Ekstazi, and OpenClover.
- To test the ability in detecting faults, artificial bugs(mutations) are first seeded in the source code by PIT[2]. Next, RTS techniques choose their tests for the modified source code and all the original tests are run. Finally, the number of bugs found by the selected tests and the original tests are compared. Those two numbers should not differ too much for an RTS technique that is good in fault detection. STARTS, Ekstazi, and OpenClover are equally good at killing mutations. Their killed mutations are the same as that of the original tests. HyRTS is the worst out of four.

In summary, Ekstazi proved to be the most efficient of the four techniques in all the measurements, especially when the program size exceeded 100 KLOC. OpenClover should be avoided if the goal is to reduce the testing time with RTS techniques.

[2]https://pitest.org

# 3 Analysis

This chapter will focus on choosing an experiment object and the three best suitable RTS tools. The experiment object is one of the projects at subshell. To determine the best three out of five RTS tools: STARTS, Ekstazi, HyRTS, OpenClover and FLiRTS 2, an analysis of their performances, compatibility, and accessibility will be conducted. The four performance metrics are end-to-end time reduction, safety and precision violations, and fault detection ability. The meaning of metrics and their correlations will be explained in section 3.2. Besides having a good performance, the selected RTS tools must also be publicly accessible, and compatible with Java 11 and JUnit 5, the testing framework utilized by SISI.

## 3.1 Subject selection

subshell's product is Sophora, a content management system that allows media organizations create, connect, curate, and publish their articles. Sophora is made up of many software components. Two of them are Sophora Server (SOSI) and Sophora Indexing Service (SISI). This work originally aims to bring a reduction of testing time for the Sophora Server, the biggest and most important component of Sophora. Though, due to the time budget of this thesis, the Sophora Indexing Service (SISI) is chosen as the experiment object. The Sophora Indexing Service is considerably smaller than Sophora Server. In both SOSI and SISI, integration tests consume the majority of the total testing time. Therefore, reducing the total testing time means smartly selecting the integration tests affected by code changes. Both projects share the same framework (Spring[1]) and programming language (Java). They also have a large overlap in their dependencies. So, if RTS tools are able to reduce the testing time for SISI, then they could also lower the testing time for SOSI.

[1]https://spring.io/

| | | **Sophora Indexing Service** | **Sophora Server** |
|---|---|---|---|
| Functionality | | Index documents for querying | Process requests from clients |
| Code metrics | Lines of code | ~ 14k | ~ 81k |
| | Code coverage | 89,2 % | 64,1 % |
| | Maven dependencies | 1132 | 1350 |
| | Commits since last 2 years | 187 | 499 |
| Build metrics | Average testing time | 6min 30s | 1h 31min |

Retrieved on 29. Jan. 2023

Table 3.1: Metrics comparison between SISI and SOSI

The Server is almost 6 times bigger than SISI in terms of Lines of code. It has about two hundred dependencies more than SISI, which could eventually cause more compatibility problems with RTS tools. With 6 minutes 30 seconds, the testing time for SISI is significantly shorter than that of the server. Thus, it will take less time and effort to spot issues during setting up the experiment with the Server.

The Maven dependencies metric is attained by Maven goal `help:effective-pom`. This goal applies inheritance rules on the pom and translates all the transitive dependencies into directive ones. The average testing time is calculated on the last 10 successful builds by the internal build system (Jenkins).

## 3.2 Evaluation metrics

This chapter defines and explains which metrics should be used to compare RTS tools. The definitions are mainly derived from [3].

### 3.2.1 End-to-end time reduction

End-to-end time reduction is the time reduction in percentage by applying an RTS tool. The better the RTS tool is, the higher its time reduction. Given the original testing time $t$ and the testing time using the RTS technique $t'$,

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

### 3.2.2 Safety violation

This indicator is used to compare an RTS1 tool known as safe with an RTS2 tool.

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

$T1$, $T2$ are the set of tests selected by RTS1 and RTS2, respectively. Safety violation is the ratio of the tests selected **only by RTS1** and the tests in the union set from $T1$ and $T2$. This ratio varies from 0 to 1. The ratio is 0 when RTS2 chooses exactly the same test as RTS1 does. That means, T1 is the same as T2 and therefore, the numerator is 0. In any other case, the numerator is always smaller than the denominator, which results in a ratio smaller than 1. A smaller safety violation is better.

### 3.2.3 Precision violation

This metric expresses how imprecise an RTS2 tool is compared to the RTS1 known as precise. It is the ratio of the tests selected only by RTS2 to the union of both test sets. A smaller precision violation is better. Following the same reasoning as with the safety violation, the value range of the metric is between 0 and 1.

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

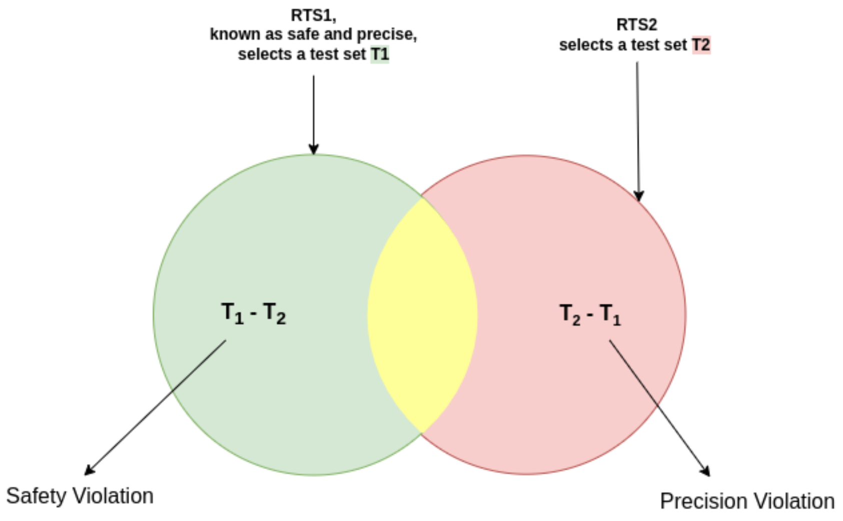


Figure 3.1: Illustration of safety and precision violations using Venn diagram

The green and red parts in the Venn diagram indicate the safety and precision violations, respectively. If there is an RTS3 tool that selects a test set T3 as the same size as T2, but T3 has more tests in common with T1, which means the yellow part is bigger, then RTS3 will have smaller safety and precision violations. As a consequence, RTS3 is better than RTS2.

### 3.2.4 Fault detection ability

This metric is the ratio of the number of killed mutations over the total mutations. The total mutations are the mutations introduced to the changed part of the program code. Each mutation modifies the program code in a predefined way. A mutation is killed if a test that is previously passed now fails. An RTS tool's selection is a subset of the original test. Mutations that are killed by an RTS tool's selection must never exceed that of the original tests. The value of this metric will never exceed 1 because the number of killed mutations cannot be bigger than the number of existing ones. The better an RTS tool is, the closer its fault detection ability to that of the original tests.

$$\text{FaultdetectionAbility} = \frac{KilledMutations}{NumberOfTotalMutations}$$

## 3.3 Reasoning about suitable RTS tools

The goal of this chapter is to determine the best three RTS tools that can be applied to the Sophora Indexing Service. Five tools: STARTS, Ekstazi, HyRTS, OpenClover and FLiRTS 2 will be examined based on their performances from empirical studies, compatibility, and accessibility. At the end of this section, the selected tools for the Sophora Indexing Service will be introduced.

### 3.3.1 Metrics from empirical studies

In this section, a ranking table of five RTS tools will be presented based on their performances. The end-to-end time reduction is considered as the most important criterion, followed by the fault detection ability, safety and precision violations.
Researchers have done studies on open-source projects to make performance comparisons of RTS tools. Considering STARTS and Ekstazi as the most effective tools, Shin et al. [3] compared them with HyRTS and OpenClover. In [5], FLiRTS 2 is introduced as a comparable RTS tool to STARTS and Ekstazi.

#### STARTS, Ekstazi, HyRTS, and OpenClover

The figures shown in this section originate from [3]. To form a ranking table of four tools: STARTS, Ekstazi, HyRTS, and OpenClover, the figures are analyzed using the suggested criteria.

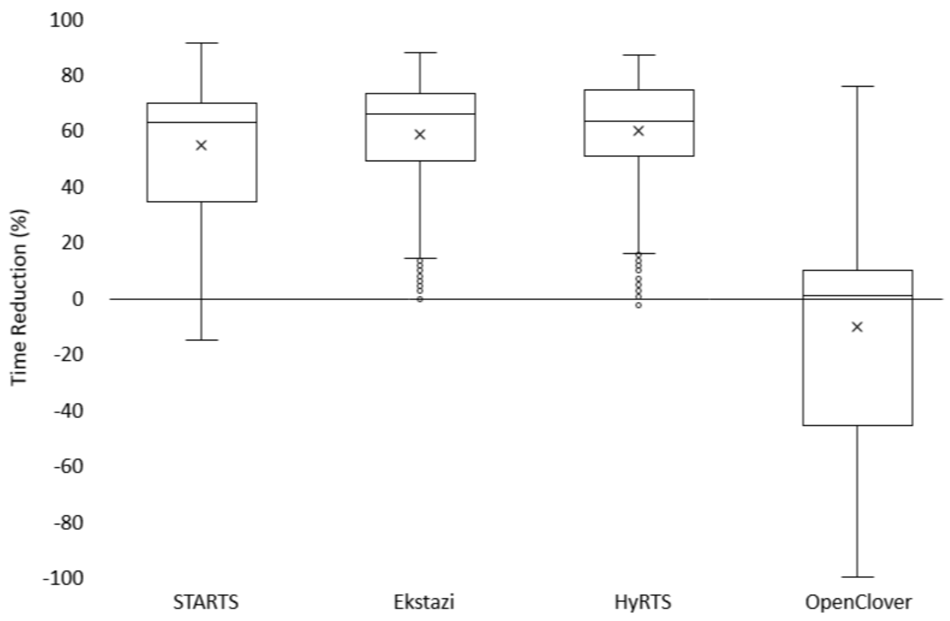


Figure 3.2: End-to-end time reduction from [3]

Figure 3.2 shows that HyRTS obtained the highest mean value, followed by Ekstazi and STARTS. OpenClover's median value is the lowest. It could reduce only 0.58% the end-to-end testing time. Shin et al. [3] conducted the parametric test and the non-parametric to check if the time reduction of four techniques is statistically significant. The conclusion is that the time reduction of Ekstazi and HyRTS are statistically similar but different from STARTS. The rankings below reflect the conclusion. Ekstazi and HyRTS are sharing the first rank. STARTS and OpenClover are in second and third place, respectively.

| | STARTS | Ekstazi | HyRTS | OpenClover |
|---|---|---|---|---|
| Ranking | 2 | 1 | 1 | 3 |

Table 3.2: Tool rankings based on end-to-end time reduction

Next, the second important criterion will be analyzed.

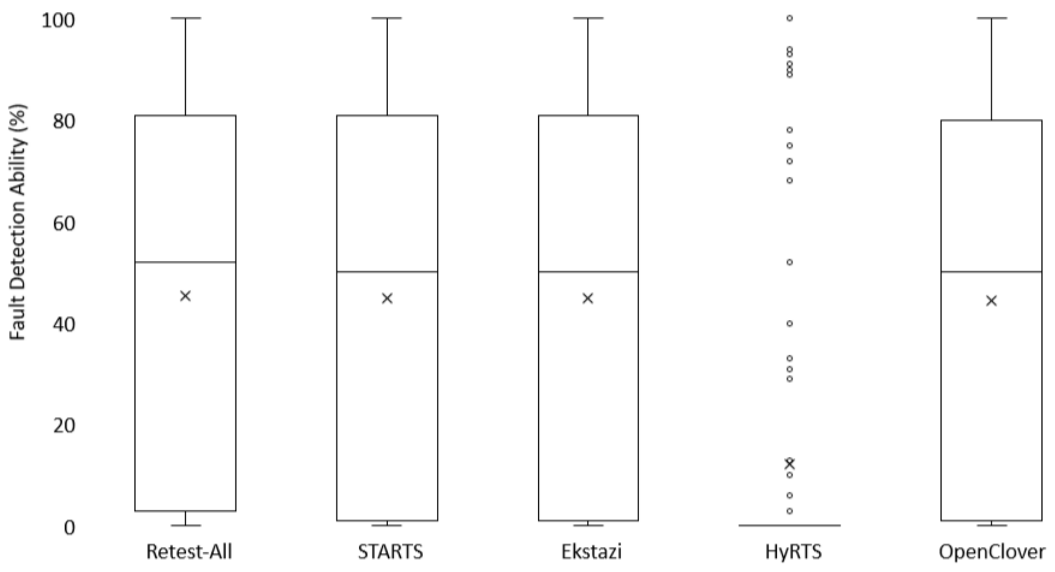


Figure 3.3: Fault Detection Ability from [3]

It is obvious, that selection of START, Ekstazi and OpenClover are almost as good as the original test suite at detecting fault. Their mean values of fault detection ability are 0.43%, 0,10%, 0.47% less than then original tests. By this tiny difference, these three tools can be seen as equally effective. On the contrary, HyRTS killed only 12.25% of the total mutations. The majority of revisions (80.82%) showed that HyRTS did not kill any mutations. This result of HyRTS could be a consequence of HyRTS misidentifying changed files or finding test dependencies. At the time, the source code of HyRTS was not publicly available, so there was no attempt to investigate this problem.

| | STARTS | Ekstazi | HyRTS | OpenClover |
|---|---|---|---|---|
| Ranking | 1 | 1 | 2 | 3 |

Table 3.3: Tool rankings based on time reduction and fault detection ability

Based on its effectiveness, Ekstazi is the best tool until now. STARTS and HyRTS have exchanged their ranks, because STARTS is significantly better than HyRTS at detecting faults. Since OpenClover could hardly save any testing time, it remains at the third place despite its good performance in killing mutations.

The last criteria of the ranking scheme to analyze are safety and precision violations. It is only reasonable to compare safety and precision violations of HyRTS and OpenClover because both STARTS and Ekstazi are used as baselines to produce those violation values. The rank of HyRTS and OpenClover will not alter, because time savings is the factor that weighs the most and in this aspect, HyRTS is far superior to OpenClover. Nevertheless, it could provide more information to better understand the time savings and fault detection ability.

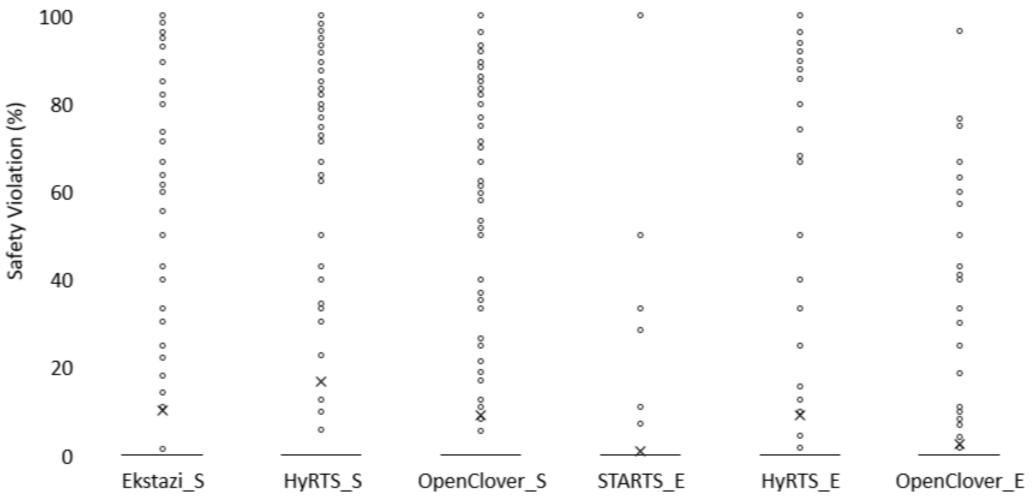


Figure 3.4: Safety violation from [3]

The suffix _*S* and _*E* denote the values calculated with respect to STARTS and EKSTAZI. OpenClover has an average safety violation rate of 9.01% wrt. STARTS and 2.61% wrt. Ekstazi. That rate of HyRTS is higher in both cases. That means, HyRTS misses more relevant tests that should be selected according to STARTS and Ekstazi. These missing tests could be directly associated with why HyRTS is inferior to OpenClover in detecting faults.

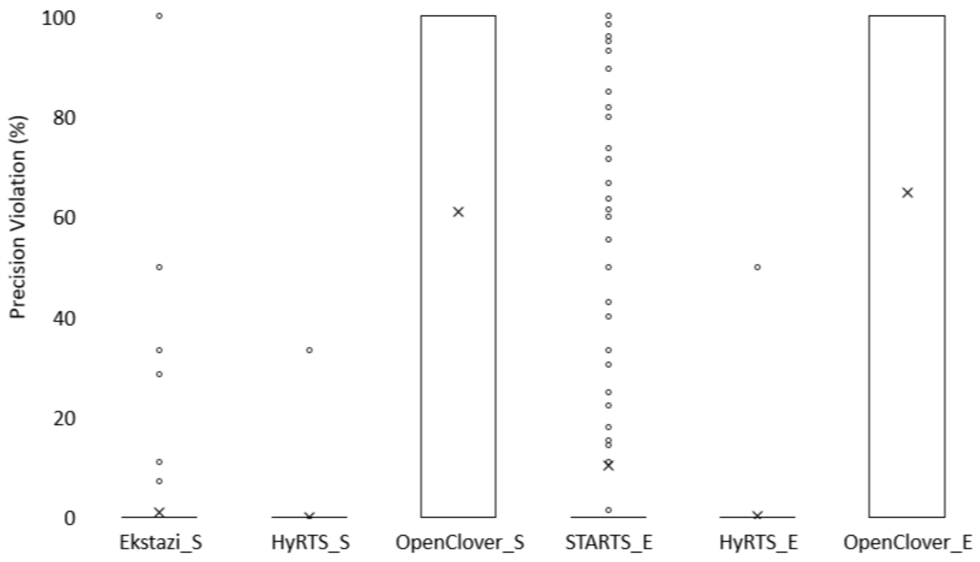


Figure 3.5: Precision violation from [3]

Whereas HyRTS's average precision violation in both respect to STARTS and Ekstazi tend towards zero, OpenClover has the highest precision violation ( 60%) among all the tools. The rate implies that OpenClover's selection contains an exessive number of tests more than the selection of STARTS or Ekstazi. This can explain why OpenClover has the lowest time savings.

**FLiRTS 2**

In this section, the ranking table above will be complemented by the fifth RTS tool, FLiRTS 2. The source information for the classification of FLiRTS 2 originates from [5], which compares state-of-the-art tools (STARTS and Ekstazi) with FLiRTS 2. The experiment in [5] includes 21 open-source Java projects with more than 8000 revisions, whereas Shin et al. [3] experimented 4 subjects, each of them has 117 revisions.

| Metrics | STARTS | Ekstazi | FLiRTS 2 |
|---|---|---|---|
| Selecting time | 102.58 s | 13.96 s | 11.83 s |
| Selecting and running tests time | 351.47 s | 238.53 s | 253.80 s |
| Safety violation of FLiRTS 2 wrt. | 16.53 % | 18.88 % | |
| Precision violation of FLiRTS 2 wrt. | 9.01 % | 13.27 % | |
| Fault detection ability of FLiRTS 2 wrt. | 95.33 % | 95.63 % | |

wrt. : with respect to

Table 3.4: Metrics comparison between STARTS, Ekstazi, and FLiRTS 2

In terms of total time savings, the third row of the table shows that Ekstazi is in first place, followed by FLiRTS 2 and STARTS. Whereas Ekstazi (238 s) is slightly better than FLiRTS 2 (253 s), there is a big gap between STARTS (351 s) and FLiRTS 2 (253 s). If only considering the time to select tests, FLiRTS 2 is the fastest tool with 11.83 seconds.
The safety violation of FLiRTS 2 with respect to STARTS and Ekstazi are both under 20%. On average, FLiRTS 2 misses about 18% of the tests that are chosen by other tools. This can be explained by the fact that FLiRTS 2's input is UML class diagrams. This type of diagram does not contain information about exceptions or Java reflection.
Approximately, FLiRTS 2 has a precision violation of 11%. That means FLiRTS 2 chooses 11% extra tests that are not chosen by both tools. The precision violation of FLiRTS 2 wrt. Ekstazi is higher than that wrt. to STARTS because Ekstazi collects test dependencies on runtime, so it can select tests more precisely.
Fault detection ability of FLiRTS 2 is around 95% compared to STARTS or Ekstazi. In comparison with the original test suite, FLiRTS 2 has a fault detection ability of 94.48% on average. These numbers mean that if 100 mutations can be found by the original test suite, then STARTS and Ekstazi would miss roughly 1 of them, and FLiRTS 2 would leave about 5.5 of them unkilled.

| | STARTS | Ekstazi | HyRTS | OpenClover | FLiRTS 2 |
|---|---|---|---|---|---|
| Performance ranking | 1 | 1 | 2 | 3 | 1 |

Table 3.5: Final tool performance rankings

Although FLiRTS 2 misses 18% of the tests that are selected by state-of-the-art tools on average, it still has a high ability in fault detection, about 94,5 out of 100 mutations. Moreover, FLiRTS 2 is in the second place in time savings, only Ekstazi is able to save slightly more time than FLiRTS 2. Therefore, FLiRTS 2 is put in the same class of STARTS and Ekstazi.

### 3.3.2 Compatibility and accessibility

In addition to good performance, RTS tools should be compatible with SISI. Moreover, RTS tools and their dependencies should be publicly accessible so that they can be applied on SISI.

#### STARTS, Ekstazi, HyRTS, and OpenClover

Four tools: STARTS, Ekstazi, HyRTS, and OpenClover, are publicly available as Maven plugins. Since the tests in SISI are written using JUnit 5 and run with Java 11, the plugins must work properly with JUnit 5 with Java 11 For the compatibility check, the latest release version of Ekstazi, HyRTS, and OpenClover Maven Plugin is used. STARTS needs a local build from public source code, which supports Java 11. Therefore, the STARTS version is the 1.4-SNAPSHOT.

To test if the RTS tools work correctly with SISI, a simple code change is made by adding a log message in a Java method. The expected result is that RTS tools should only select test classes that call the method. STARTS and OpenClover meet this expectation. On the contrary, Ekstazi and HyRTS select considerably more than the expected test classes. To ensure this problem is caused solely by JUnit 5 and not by other dependencies of SISI, Ekstazi and HyRTS are experimented on a simple Maven project that uses JUnit 5. The structure of the project is presented in fig. 3.6.

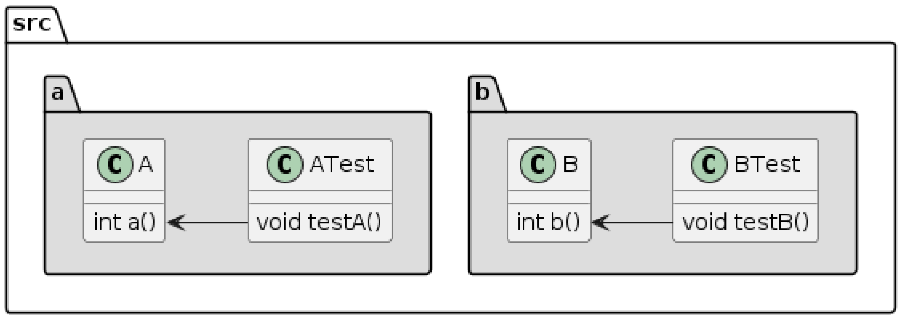


Figure 3.6: A simple Java project with four classes

There are two packages: a and b and four classes: `A`, `ATest`, `B`, and `BTest`. The arrows indicate that each test class depends on its respective source class. If the method `B.b()`, Ekstazi and HyRTS should select only `BTest`. Ekstazi, in the latest version (5.3.0), selects both `ATest` and `BTest`. This behavior of Ekstazi is independent of Java versions (1.8 or 11).
The latest HyRTS version (1.0.1) also overselects tests when using it with Java 11. When running with Java 8, HyRTS cannot select any test, Because it results in a RunTimeException related to computing the diff between versions. Table 3.6 below summarizes the compatibility of RTS tools with JUnit 5.

| | STARTS 1.4-SNAPSHOT | Ekstazi 5.3.0 | HyRTS 1.0.1 | OpenClover 4.4.1 |
|---|---|---|---|---|
| Compatible with JUnit 5 | yes | no | no | yes |

Table 3.6: Tool compatibility with JUnit 5

### FLiRTS 2

FLiRTS 2 [5] is a Java software that is available as a jar file. Its installation and run guide can be found on its official website [2]. In each revision, FLiRTS 2 requires the model of the source code. The model contains UML class diagrams achieved by using the Java to UML transformation plugin of the Rational Software Architect (RSA) framework [19]. Unfortunately, there is no open access to the RSA framework.

[2]https://cazzola.di.unimi.it/flirts2.html

#### Final tool rankings with compatibility and accessibility

Even though FLiRTS 2 is ranked to have the same quality as state-of-the-art tools, STARTS and Ekstazi. It cannot be applied on SISI because of no open access to one of its dependencies (Rational Software Architect framework). Ekstazi, in the latest version(5.3.0), is not working with tests written using JUnit 5. The same reason goes for HyRTS. Therefore, Ekstazi and HyRTS are not included in the experiment of this work. The two remaining tools that support JUnit5 are STARTS and OpenClover. Rankings based on performance, compatibility and accessibility of five RTS tools are recapped in table 3.7.

| | STARTS | Ekstazi | HyRTS | OpenClover | FLiRTS 2 |
|---|---|---|---|---|---|
| Performance ranking | 1 | 1 | 2 | 3 | 1 |
| Compatible with JUnit 5 | Yes | No | No | Yes | Unchecked |
| Publicly accessible | Yes | Yes | Yes | Yes | No |

Table 3.7: Tool performance rankings, compatibility, and accessibility

### 3.3.3 Selected RTS tools

Table 3.7 indicates that STARTS, Ekstazi, and FLiRTS 2 are the top three RTS tools based on their performance. Unfortunately, both Ekstazi and FLiRTS 2 cannot be used due to compatibility and accessibility issues. Additionally, HyRTS is not compatible with JUnit 5, which SISI utilizes. Therefore, the two final tools chosen for SISI are STARTS and OpenClover. This chapter explains the selection process of both STARTS and OpenClover.

#### STARTS

STARTS[12] is a static RTS tool for Java that works at class-level. STARTS is installable as a Maven plugin, and its source code is available on Github[3]. The goal *starts* : *starts* of the STARTS plugin will perform five following steps:

- **Finding Dependencies Among Type**: First, both source code and test code of a program need to be compiled. Its output is classfiles, each of them contain a constant pool. Then, *jdeps*[4] reads the constant pool of every classfile (e.g. of a type A) to determine the types that the type A depends on.

[3]https://github.com/TestingResearchIllinois/starts
[4]https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html

- **Constructing the Dependency Graph**: The output of *jdeps* from the previous step is used to build a *yasgl* [5] graph, in which an edge connects one type to each of its dependencies. The *yasgl* graph allows a quick trainsive clousure computation that is needed for finding all the types each tests depends on.
- **Finding Changed Types**: STARTS computes checksum of a classfile to identify if the type in that classfile is modified since the last run. The checksums are stored in a file for future runs. Note STARTS computes checksum of classfiles, because it is more precise than checksum of source file and more reliable than timestamp-based solution.
- **Computing and Storing Checksums**: STARTS maintains a single file that contains a *type-to-tests* mapping for every type. This mapping shows which tests are dependent on a type.
- **Selecting Impacted Tests**: Based on the *type-to-tests* file and the changed types, STARTS finds the set of **not** impacted tests. The impacted tests are the difference of the set of tests in the current revision and the set of non-impacted tests. Thus, it always contains the newly added tests.
- **Running Impacted Tests**: In this step, the non-impacted tests are added to the set of tests that are marked as not run by Surefire[6] plugin.

At the time of this writing, the latest release version of the STARTS Maven Plugin is 1.3, which does not support Java 11. To support Java 11, a local build of the plugin is required.

### OpenClover

OpenClover[7] is available as a Maven plugin that provides the test optimization feature[8]. The plugin source code is accessible via Github [9]. This plugin performs three Maven goals (`setup, optimize, snapshot`) to select and run the selected tests. Unlike STARTS, for a continuous test optimization over several revisions, selecting and running selected tests are inseparable. Because as a dynamic RTS tool OpenClover updates its test dependencies with the test execution runtime information.

[5]https://github.com/TestingResearchIllinois/yasgl
[6]https://maven.apache.org/surefire/maven-surefire-plugin/
[7]https://openclover.org
[8]http://openclover.org/doc/manual/latest/general–what-is-test-optimization.html
[9]https://github.com/openclover/clover-maven-plugin

- **setup**: Instrument all source files and update its registry. It is configurable to reinstrument only the modified files.
- **optimize**: Based on information from the registry and the Clover snapshot, the tests that need to be run are selected. After this step, the instrumented files are compiled and the selected tests are run.
- **snapshot**: The snapshot holds the information about which tests hit which source files. It is the so-called test dependencies. During the tests execution, the snapshot is refreshed.

The plugin documentation page[10] provides descriptions of the plugin goals together with their configuration parameters.

[10]https://docs.atlassian.com/clover-maven-plugin/4.1.2/plugin-info.html

# 4 Design

After RTS tools are analyzed in Chapter 3, STARTS and OpenClover are selected to experiment on the Sophora Indexing Service. This chapter gives an outline for a two-phase experiment. The experiment will produce reliable data to compare STARTS and OpenClover. The comparison of RTS tools requires four metrics: end-to-end time reduction, safety violation, precision violation, and fault detection ability. The first phase of the experiment, which is *Selecting and running tests*, is responsible for generating data that will allow for an investigation of the first three metrics. The output from the second phase, which is *Mutation testing*, will be used to compare RTS tool's abilities in detecting fault. At the end of this chapter, the number of code revisions that RTS tools should repeat will be discussed, since it is vital for having dependable data.

## 4.1 Selecting and running tests

This is the first phase of the experiment, in which data for comparing three metrics: end-to-end time reduction, safety violation, and precision violation will be created. In this phase, STARTS and OpenClover select tests and the selected tests are subsequently executed. The selected tests will be useful for comparing safety and precision violations. Along with the runtime of selected tests, the time of RTS's selecting process will be recorded. These two times combined will create the RTS tool's execution time. The end-to-end time reduction metric is calculated on the basis of RTS tool's execution time and the runtime of all tests.

### 4.1.1 STARTS

In order to create dependable data, STARTS and Clover need to select tests on multiple source code changes. A source code change (diff) is retrieved between two code revisions. Iterating over a set of chronologically sorted revisions helps to create multiple diffs by determining a diff between the current and the last revision.
The diagram below shows that a set of code revisions is iterated through. On each revision STARTS calculates the diff and then generates a set of affected tests. This set of

tests will be stored for evaluating the safety and precision violations. The affected tests are run to measure time. The time for generating affected tests is also recorded.
The *STARTS diff* in fig. 4.1, is different from the diff between revisions of version control. It contains only source code changes, in other words, Java classes, whereas the diff of version control could include changes of any resources or configuration files. The *STARTS diff* is not employed to evaluate any metrics, but rather utilized as an input for the subsequent phase of the experiment, which is *Mutation testing*.
After checking out a revision, STARTS compiles the source and test code. Based on the compiled code, it creates test dependencies of the revision. In order to select the tests, STARTS compares the test dependencies of the revision to that of the last revision. After selecting tests, STARTS updates its test dependencies with the new ones.

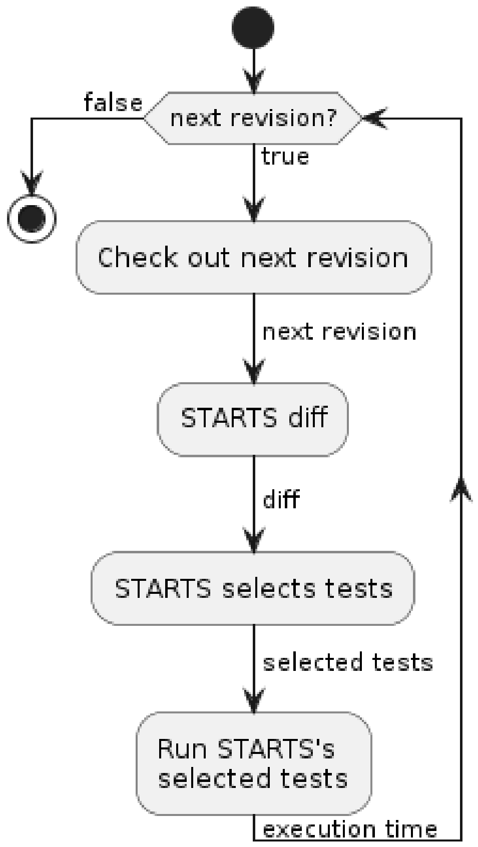


Figure 4.1: Activity diagram of STARTS

### 4.1.2 OpenClover

The activity diagram of OpenClover (fig. 4.2) is quite similar to that of STARTS (fig. 4.1), except for two things. This first thing is visible in the last step of the diagram, which is *Update the Clover snapshot*. As Clover utilizes dynamic code analysis, it needs to update its test dependencies (the Clover snapshot) based on runtime information.
The second thing is a technical difference that leaves out of the diagrams for the sake of simplicity. Still, it is worth mentioning that STARTS selects tests after the source files are compiled and OpenClover does that before the compiling. To generate a selection, OpenClover does not compile the source files, but instruments the files and updates its registry. Employing the registry and the Clover snapshot, OpenClover finds out which tests should be rerun. After the affected tests are identified, the source and test code are compiled, followed by running the affected tests. During the test execution, the snapshot is updated and will be used for the next selection.
To serve the goals of this phase, which is collecting data for the metrics, OpenClover's selection is logged and execution time is measured on every revision.

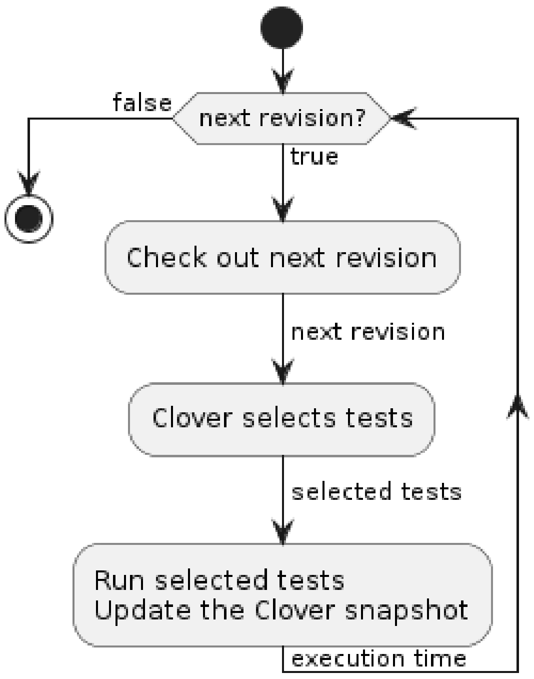


Figure 4.2: Activity diagram of OpenClover

### 4.1.3 Running all tests

Executing STARTS and OpenClover are the first and second parts of the first phase. The third and last part of this phase is running all tests. This part is done after STARTS and Clover have finished selecting and running selected tests for all revisions. Time measurement of this part is used as a baseline for calculating the end-to-end reduction metric. Besides that, this part can find out if a revision has a green test suite, meaning all tests are passed, which is needed for the *Mutation testing* phase. This part can be done simply by iterating over revisions and running a Maven test command on each revision.

## 4.2 Mutation testing

This is the second phase of the experiment. This phase will provide data for comparing STARTS's and OpenClover's fault detection ability to that of all tests. The results of the first phase: STARTS's diff, STARTS's selection, and OpenClover's selection are input for this phase.
Mutation testing is regarded as a better method to measure code quality than traditional test coverage. Because it not only calculates line coverage but also checks if the tests can detect faults. In mutation testing, faults (mutations) are seeded into the code and then the tests are executed. The more mutations are killed, the better the tests are. A mutation is killed if a test is passed and now fails. PIT provides an easy-to-adopt functionality for mutation testing and is frequently used in research [3], [18], [5]. Given a set of target classes and a set of target tests, PIT will use the specified mutation operators to inject faults into the target classes, and then rerun a subset of target tests. The subset of tests is determined by the line coverage analysis that is done by PIT beforehand.

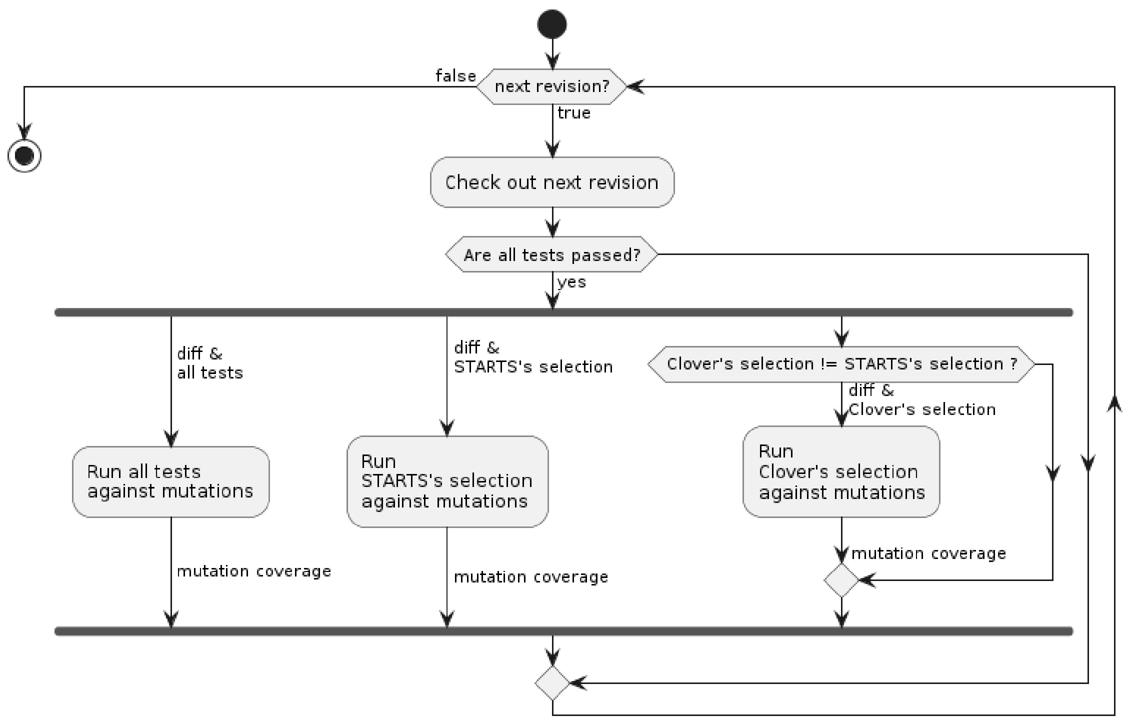


Figure 4.3: Activity diagram of mutation testing

In the fig. 4.3, after a revision is checked out, all tests in the revision are checked if they are passed. Instead of running all tests, the test report from section 4.1.3 can be used to do this check. The reason for this check is PIT requires a whole green test suite. If a test turns from passed to failed, then that must be caused by the mutation. But if a test is failed at first and then fails again after mutating, then it is not entirely sure that the cause for the test failing the second time is the mutation.
The *STARTS diff* in fig. 4.1 is passed to PIT as target classes. In each revision, there are three test sets in total: STARTS's, OpenClover's selection, and all tests. Those sets are provided to PIT as target tests. There are three sets of input for PIT that share the same target classes (the diff) but differ from each other by the test classes (the selections). As the processes do not depend on each other, they are designed to run in parallel to save time. These three parallel processes are visualized between the two thick horizontal lines in fig. 4.3. If STARTS and OpenClover have the same selection, then PIT needs only to be run once. That is why OpenClover's selection is compared with that of STARTS before the third process (rightmost) starts. Each process produces a mutation coverage report that is used to calculate fault detection ability.

## 4.3 Revisions

Shin et al. [3] experimented on 4 subjects, on average each subject has 117 revisions. Cazzola et al. [5] designed their research in a bigger scale that includes 21 subjects with more than 8000 revisions. Although their experiments include a large number of revisions, the tests are mostly unit tests. In terms of quantity, unit tests are the majority in SISI. Though, integration tests take up most of the testing time. Due to the time budget of this thesis, a range of 50 to 100 code revisions is considered, the exact number will be decided in section 5.4.

# 5 Implementation

This chapter describes the implementation of the two-phase experiment proposed in Chapter 4. In the first phase, the implementation follows the procedure suggested in fig. 4.1 and fig. 4.2. The implementation of this phase includes running STARTS, Clover, and all tests sequentially, collecting their selections, and measuring the execution time. The collected data of this phase help to provide a comparison of these two RTS tools in terms of time reduction, safety, and precision violation. The implementation of the second phase is based on fig. 4.3, which uses the selections in the first phase as input to execute PIT. PIT executions produce mutation coverage reports containing information about fault detection ability. The content of this chapter will cover the detailed procedures for running STARTS, Clover, and PIT, as well as the technical issues that have arisen.

## 5.1 Experiment requirements

Following version information about tools and libraries that are utilized to conduct the experiment: Apache Maven 3.6.3, STARTS 1.4-SNAPSHOT, Clover 4.4.2-SNAPSHOT, PIT 1.9.8, JUnit 5.7.2, Maven Surefire Plugin 3.0.0-M5. The tools and libraries are run with Java 11.0.6. The Python scripts are run with Python 3.8.

## 5.2 Selecting and running tests

This phase of the experiment is implemented using a Bash script. The Bash script runs an RTS tool (STARTS or OpenClover) that is specified via its argument. This script iterates over 40 revisions. For each revision, it follows this procedure.

1. checking out a revision
2. copying the project content to a working directory
3. adding the configuration of the respective Maven Plugin in the project configuration file (POM)

4. running the respective Maven Plugin (STARTS or OpenClover)

5. cleaning up the working directory

Steps (1) and (4) are the main steps, they are outlined in fig. 4.1 and fig. 4.2 from Chapter 4 Design. Steps (2), (3), and (5) are extra steps, they do preparatory work and reduce unwanted side effects between revisions. These three steps are not included in the design.
Steps (1), (2), and (5) are accomplished quickly via `cp`, `git checkout`, and `rm` command. In (5), everything inside the working directory will be removed but the RTS tool's directory (`.starts` or `.clover`). (3) is done by a Python script that is executed within the Bash script. The most important step is (4) which selects tests and runs the selected tests. Step 4 is done differently depending on the specified RTS tool, so they will be discussed in depth in section 5.2.1 and section 5.2.2.
To have a reliable time reduction metric, this phase of the experiment is repeated four times. Only the results of the last there runs are used for time reduction calculation. Since the first run could take longer to complete, because of downloading Maven artifacts or no memory cache.

### 5.2.1 STARTS

This section describes how the design in fig. 4.1 is implemented.
STARTS provides its functionalities via a Maven plugin. Since SISI is a Maven project, the STARTS plugin can be simply integrated by adding its configuration to the project configuration file (POM). The use of this plugin is done mainly through 3 goals: `diff`, `select`, `starts`. The purpose of `diff` is to identify changes since the last time STARTS was executed. The `select` goal selects affected tests by the most recent changes. The goal `diff` and `select` execute the step "STARTS diff" and "STARTS selects tests" in fig. 4.1, respectively. To select tests and run selected tests in one command, the `starts` goal can be used. STARTS's execution time is the execution time of this goal. Except for `starts`, every other goal does not automatically update and save the checksums of the files in the latest version by default. In the experiment, flags (`updateDiffChecksums` and `updateSelectChecksums`) are set to update the checksums, which is needed when running STARTS on multiple revisions.

**Technical issues**

The STARTS release version at the time of this writing is 1.3, which does not support Java 11. However, STARTS is compatible with Java 11 in the unreleased version. Therefore, the STARTS project [1] is cloned and built locally, which creates the STARTS 1.4-SNAPSHOT version. In the snapshot version, the `starts` goal always selects and run all tests instead of the affected ones. As a workaround for this problem, tests are selected by the `select` goal and run separately by the Maven Surefire plugin. A file containing selected tests is parsed from the build log file of the `select` goal. The path of that file is then passed to Surefire plugin as a command line argument.

### 5.2.2 OpenClover

This section explains the implementation of the design shown in fig. 4.2.
OpenClover is available as a Maven plugin. according to OpenClover's quick start guide[2], steps "Clover selects tests" and "Run selected tests ..." in fig. 4.2 should be achieved by running a single command. The command is `mvn clover:setup clover:optimize test clover:snapshot`. The command generates a Maven build log. The duration of the build is OpenClover's execution time. Due to the use of Lombok's annotations in SISI, the command does not give the desired result. Lombok's annotations can help to avoid boilerplate code. The annotations must be transformed into Java code. After the transformation, there is a new source directory. OpenClover should instrument the code in the new source directory, but there is no option to specify the path to a custom source directory from the project root. As a solution for this limitation, the transformed Java code is copied to the default (old) source directory, so that OpenClover's instrumenting can work correctly. This is done by running a simple script using the Exec Maven Plugin[3]. The public source code of the Clover Maven Plugin is cloned and modified so that a file path can be passed in as a configuration parameter. The file path is where the selected tests should be stored.

### 5.2.3 Running all tests

This part of the first phase is carried out using another Bash script. The script checks out every revision and runs all tests in a revision using the command `mvn test`. The execution time of this command is used as a baseline for the time reduction metric.

[1] https://github.com/TestingResearchIllinois/starts
[2] https://openclover.org/doc/manual/latest/maven–using-test-optimization.html
[3] https://www.mojohaus.org/exec-maven-plugin/

## 5.3 Mutation testing

This section discusses how the second phase of the experiment is implemented. This phase provides data for the fault detection ability metric. To do that, a Bash script is written based on the proposed design in fig. 4.3. The script iterates over revisions. For each revision, the results from the first phase are passed to the PIT Maven plugin as input, and the plugin is executed to create mutation coverage reports.

### The PIT Maven plugin

This section introduces the PIT Maven plugin and its configuration options.
As mentioned in section 4.2, PIT Maven plugin[4] is a suitable tool for mutation testing. For the PIT Maven plugin to work on JUnit 5 tests, it requires the JUnit 5 plugin for pitest[5]. PIT seeds mutations directly into byte code and runs the compiled tests classes, so the source code needs to be compiled via command `mvn test-compile` beforehand.
A list of Java classes that PIT should inject mutations into is specified through the configuration option `targetClasses`. The Java test classes that should be run against the mutations can be configured by `targetTests` option. If nothing is specified, PIT will try to mutate all source classes and run all test classes.
As default, a mutation will be classified as timed out if its corresponding test run takes longer than *1.25* times its original execution time. To give test runs more time, the *timeoutFactor* is set to *2*.
The goal `mutationCoverage` of PIT plugin can be run with an option for exporting the coverage report as *html*. Besides the number of killed over total mutations, the report shows the location of the seeded mutations and a list of the examined tests.

### PIT Execution

One each revision, the Bash script of this phase follows these steps:

1. checking if all tests in the revision are passed
2. extending the PIT's base configuration by `targetClasses` and `targetTests`
3. adding the extended configuration of PIT Maven plugin to the POM file

[4]https://github.com/hcoles/pitest
[5]https://github.com/pitest/pitest-junit5-plugin

4. running command `mvn test-compile pitest:mutationCoverage -Dfeatures=+EXPORT`

In the activity diagram of fig. 4.3, the first action relates to step (1), and each action noted with "Run [...] against mutations" refers to step (4). Steps (2) and (3) process the input denoted by "diff & [...] tests/selections". They are done by a Python script that adds PIT's configuration and the content of the "STARTS diff" file as well as the selection file in the POM file.
As designed, the command in step (4) can start a process. In each revision, there are three parallel processes at most, two for STARTS's and OpenClover's selection, and one for all tests. In case STARTS's and OpenClover's selection are the same, only two processes are started. Every process of the next revision will wait to start until all processes of the current revision are finished.
Even though the SmokeIntergrationTest is passed in the normal test run, PIT often reports that it is not passed during its line coverage analysis without mutation. As a result, PIT does not start, because it requires a green test suite. That is why the SmokeIntergrationTest is excluded from target tests.
Time measurement in this phase of the experiment is not the main focus and PIT will always create the same mutations on the same input, so the script only needs to be run once.

## 5.4 Revisions

The experiment is initially designed to run with 50 to 100 revisions of SISI. Because of SISI compile issues, it was conducted on 40 revisions. The revisions are chronologically ordered from the first release (4.0.0 on December 2021) to the 4.3.6-SNAPSHOT version (January 2023). The number 4 stands for the 4th product version. The newest product version that has not been released yet is 5th. Firstly, the development of SISI is taken place on a Git branch that is compatible with the 5th Sophora product line. To have a SISI version that is compliant with Sophora 4, the branch was taken over to a new branch. At that time, the new branch needed a few commits that fixed compile issues. The issues are rooted in the API between the product lines. SISI in all revisions prior to those fixing commits had the issues. That is why the revisions before those commits cannot be taken into account.

## 5.5 Calculate metrics

The output of every step in the experiment is stored as raw text, *html* as well as *xml* files. The important information of the output files is then extracted, mostly by parsing, to *csv* files. As mentioned in section 5.2, the phase "Selecting and running tests" are repeated four times and results of the last three runs are saved for time reduction calculation. Thus, the time reduction metric is the average result of the three runs. A Python library, panda [6], helps to transform data in csv to analysis-ready data. The visualization is done by matplotlib [7].

[6]https://pandas.pydata.org/docs/
[7]https://matplotlib.org/stable/index.html

# 6 Evaluation

This chapter evaluates STARTS's and Clover's performance based on the metrics that are introduced in Section 3.2. The results of the experiment are compared with the results in [3]. At the end of the chapter, some options are discussed to make the fault detection ability metric more reliable.

## 6.1 End-to-end time reduction

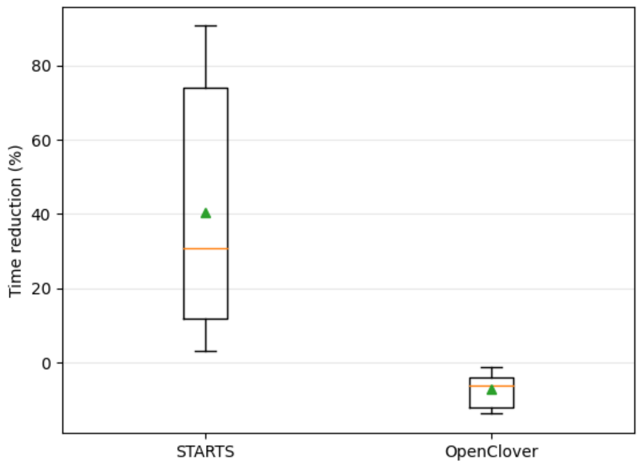


Figure 6.1: End-to-end time reduction

The boxplots in fig. 6.1 show the end-to-end time reduction achieved by STARTS and OpenClover compared to rerun all tests. The end-to-end time reduction of an RTS tool is the difference between the time execution of the RTS tool and the time it takes to run all tests. The execution time of an RTS tool is the sum of the time for the selection process and the runtime of selected tests. On average, STARTS can save 40.5% of the testing time. The median value of STARTS's time reduction is 30%. In contrast, OpenClover cannot save any time in any revision. The average OpenClover's execution time is even

]

7.28% more than rerun all tests. This result is relatively consistent with the result in [3] where the mean value of the STARTS's and Clover's time reduction is around 50% and -10%, respectively. In 2019, Shin et al. [3] found an explanation for the poor performance of OpenClover on its official website. According to the explanation, OpenClover must create and update a large number of per-test coverage files. The number is proportional to the number of test classes and the number of test cases. The explanation is no longer found on the website at the time of this writing. As OpenClover cannot improve testing time, it will be left out of further evaluation that relates to time reduction. The next two figures explain why STARTS can save 40.5% of the testing time and why its time reduction values span a wide range.

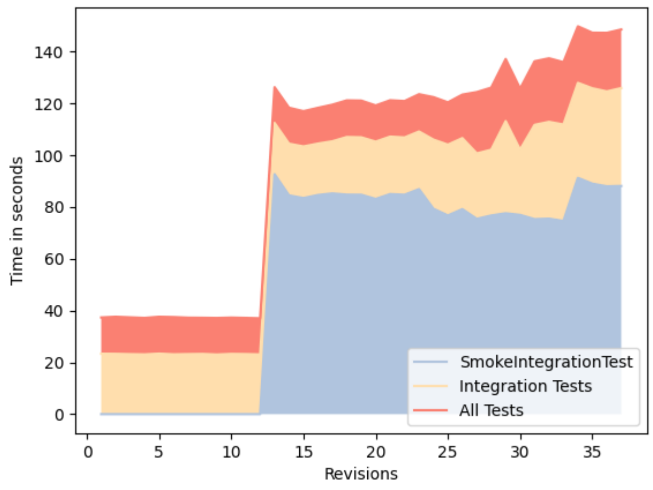


Figure 6.2: Execution time of unit and integration tests

There are two types of tests in SISI: unit and integration tests. Line chart 6.2 helps illustrate the ratio of testing time between them. In the first 13 revisions, integration tests take about twice as long as unit tests. From the 14th revision, this ratio grows from twice to five times because of the SmokeIntegrationTest. This test starts up all the components in separate Docker containers as in operating conditions and thus is the most time-consuming test.

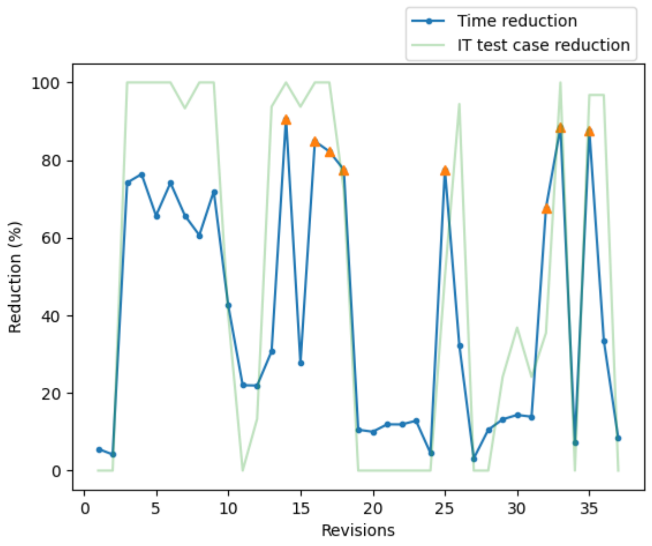


Figure 6.3: Effect of integration tests on STARTS's time reduction

Figure 6.3 visualizes the correlation between IT test case reduction and the total time reduction. The blue line depicts the time reduction of STARTS with respect to all tests. It goes through a wide range of values from around 5% to 90%. The orange triangles note that the SmokeIntegrationTest is not included in the selection. From the 14th revision, the peaks of the blue line are consistently bound to an orange triangle. This is logical, because as shown in fig. 6.2, if STARTS does not select the SmokeIntegrationTest, then the second third of the full testing time can be saved at the minimum. The green line demonstrates IT test case reduction with respect to all IT test cases. The number of test cases in the SISI test classes varies significantly. Omitting two different IT tests could lead to considerable differences in time reduction, although they result in the same test reduction. The IT test case reduction can have a stronger correlation with the time reduction, since the more test cases a test contains, the more likely it is that test will take longer to finish. That is the reason why the IT test case reduction is used instead of IT test reduction in fig. 6.3. The blue and green lines share the same pattern when they have peaks and lows. That implies that STARTS's time reduction depends mainly on the time saved by the IT tests. This implication also applies to revisions (1 - 13) without the SmokeIntegrationTest, as the figure shows.

## 6.2 Safety and precision violations

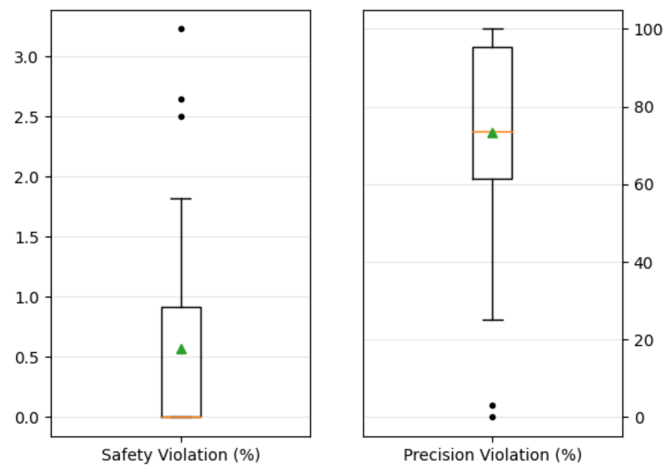


Figure 6.4: OpenClover's safety and precision violations

As discussed in section 3.2, two other metrics to compare RTS tools are safety and precision violations. Since STARTS is broadly considered a state-of-the-art RTS tool, it is set as a baseline for these two metrics. Safety violation indicates the percentage of tests that are selected by STARTS but not by OpenClover over the tests combined from both of them. Precision violation is calculated in the same way, except that the numerator is the tests that are included by OpenClover and not by STARTS. The left boxplot in fig. 6.4 shows that both mean and median values of safety violation are under 1%. That means Clover selects virtually all tests that should be selected. On the contrary, the OpenClover's average precision violation is 74%. This high number could explain why OpenClover achieves no reduction in testing time. The big gap between these two violations achieved by this work is relatively comparable to that in [3] where the average safety and precision violations are 9.01% and 60%.

## 6.3 Fault detection ability

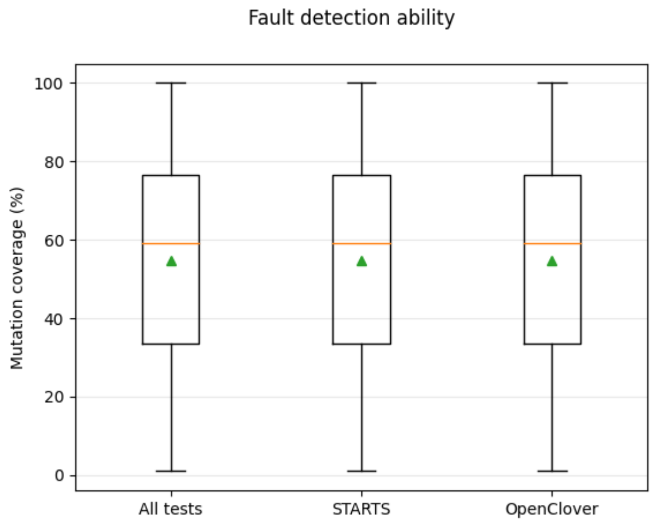


Figure 6.5: Fault detection ability

Figure 6.5 displays the percentage of the killed mutations over all mutations among STARTS's, OpenClover's selection, and all tests. Both STARTS and OpenClover achieve the same average mutation coverage as all tests, which is 55%. Their median values are also equivalent, which is 59%. A detailed observation shows that their fault detection abilities are identical to that of all tests in every revision. That means the tiny safety violation of OpenClover with respect to STARTS does not cause any negative impact on its capability in detecting fault. A large number of optional selected tests by OpenClover, which is reflected in precision violation, brings no advantage in killing mutations either.

## 6.4 Threats to validity

On the official website of PIT, applying PIT on unit tests is explicitly mentioned, but there is no information about the usage of PIT on IT tests. In this work, PIT is used on both types of tests. Even though the PIT log files show that the IT tests kill mutations as unit tests do, there is uncertainty about its correctness due to the lack of PIT documentation about IT tests. This page [1] from PIT notes that a mutation may be detected as time out on one run, but killed or surviving on another. This phenomenon

[1] https://pitest.org/faq/

of PIT is also found in the experiment. The number of killed mutations by STARTS's and Clover's selection is identical to all tests in every revision. Though, they are not the same in quality in every revision. That means, in some revisions, the killed mutations are not the same, but the survived and killed mutations compensate each other to result in the same number of killed mutations.

PIT's line coverage analysis usually reveals that the SmokeIntegrationTest fails, despite being passed by the Surefire Plugin when running all tests. Since PIT requires a green test suite, this SmokeIntegrationTest must be excluded from PIT's execution. fig. 6.5 shows that STARTS and Clover kill mutations as well as all tests in every revision. However, if the SmokeIntegrationTest could be executed during mutation testing, especially in revisions that STARTS considers it as unaffected by code changes, the result might slightly vary.

SISI does not include any tests that could give variable results without alteration, the so-called flaky tests. However, it is important to point out that flaky tests can reduce the reliability of mutation testing. If a flaky test fails against a mutation, PIT will identify this as a killed mutation, yet the cause could be the inherent non-deterministic behavior of the test.

# 7 Generalization

The findings of this work show that if the objective is to reduce testing time by using RTS tools, then STARTS is highly recommended and OpenClover should be avoided. The experiment object (SISI) provided by subshell uses common and popular technologies and popular Java libraries. It is based on Spring, built with Maven, uses JUnit 5 as testing framework, and runs on a fairly recent Java Platform (Java 11 LTS). When other firms choose a project to evaluate RTS tools, the more similar these project properties are, the more transferable the results are.
The integration of STARTS into a Maven project is relatively straightforward, since the POM file of the project only needs to be extended with the configuration of the STARTS Maven plugin. STARTS can be utilized in two forms: on developer's machines and dedicated CI/CD servers. Developers can get feedback sooner on their code changes by running only the impacted tests that are identified by STARTS. Incorporating STARTS into the development or release pipelines that operate on a CI/CD server is a decision that should be made carefully. Although STARTS proved to be effective in detecting faults in this work and other empirical studies, it cannot guarantee that every commit will be bug-free through its bug-free selected tests. A hasty testing process that results in a fault being released may cause more damage than any saved cost.
Additionally, Ekstazi, a state-of-the-art RTS tool could be useful for other projects at companies that do not rely on a particular testing framework or are using one other than JUnit 5, though it is currently not compatible with JUnit 5. FLiRTS 2 is not evaluated in this work due to a lack of access to the Java to UML transformation plugin of the Rational Software Architect (RSA) framework. Still, companies that are already using the plugin could double-check the performance of FLiRTS 2, which is proven to be effective on many open-source projects.

# 8 Conclusion and future work

In this chapter, the results of this thesis are summarized, and possible further research will be discussed.

## 8.1 Conclusion

This research delves into an investigation of 5 Regression Testing Tools: STARTS, Ekstazi, HyRTS, OpenClover, and FLiRTS 2, scrutinizing their performance, accessibility, and compatibility with JUnit 5.
The analysis of two previous empirical studies has revealed that STARTS, Ekstazi, and FLiRTS 2 are the most effective tools. In terms of compatibility, a check indicates that only STARTS and OpenClover are fully compatible with JUnit 5, whereas compatibility for FLiRTS 2 has not been checked. However, FLiRTS 2 has accessibility issues, as there is no open access to a component that is responsible for transforming source code to UML class diagrams, which are essential inputs for FLiRTS 2. As a result, only STARTS and OpenClover have been experimented with the Sophora Indexing Service (SISI) using a set of 40 code revisions from release version 4.0.0 to 4.3.6. SISI is a subshell's Java project. It is based on Spring, built with Maven, uses JUnit 5 as testing framework, and runs on Java 11 Platform.
The results of the experiment indicate that STARTS performs better than OpenClover in terms of reducing the testing time. On average, STARTS reduces the testing time by 40.5%, with a median time reduction of 30%. This reduction in time is mainly achieved through integration tests. In contrast, OpenClover fails to save any time on any revision and even increases the testing time by around 7% by selecting many irrelevant tests. The reason is that OpenClover's need to collect and update its dynamic test dependencies adds a considerable amount of time to test execution. The number of irrelevant tests selected by OpenClover reflects its overall precision violation, which is 60%. In terms of safety violation, the mean and median value of OperClover's safety violation with respect to STARTS is less than 1%. That means OpenClover rarely misses a test that is selected by STARTS. The selections of STARTS and OpenClover never miss a fault that

is detected by the original tests. That indicates they are as good as rerunning all tests in detecting faults. Their average mutation coverage is 55%.
If the objective is to reduce testing time by using RTS tools, STARTS should be employed and OpenClover should be avoided. subshell could potentially leverage STARTS as an RTS tool that aids developers in identifying and executing relevant tests on their local machines prior to committing changes that may trigger long-running pipelines.

## 8.2 Future work

The objective of the STARTS Maven Plugin is to choose and execute tests, yet the respective Maven plugin goal `starts` is currently not functioning as intended. To enable STARTS to be used in a subshell, a fix is required. In chapter 3, it was determined that Ekstazi and HyRTS are not compatible with JUnit 5. As the source code for Ekstazi and HyRTS is publicly available, improvements can be made to facilitate compatibility with JUnit 5.
By omitting the SmokeIntegrationTest, RTS tools can significantly reduce testing time, although its fault detection capability is presently being ignored. This is due to the fact that PIT reports the test as failed during its line coverage analysis. Further work can uncover the underlying cause. By enabling PIT to run on the SmokeIntegrationTest, the comparison of fault detection ability between RTS tools and the original test will be more accurate.
In [16], [7], [6], big tech companies report that RTS data-driven approaches can save up to 30% of the testing time while reporting 99% of buggy pull requests. In those approaches, large data of test execution and version control play an important role. Future studies could investigate if the applicability of those approaches can be beneficial for software enterprises.

# Bibliography

[1] Hyunsook Do. Recent advances in regression testing techniques. *Advances in computers*, 103:53–77, 2016.

[2] 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.

[3] Min Kyung Shin, Sudipto Ghosh, and Leo R Vijayasarathy. An empirical comparison of four java-based regression test selection techniques. *Journal of Systems and Software*, 186:111174, 2022.

[4] Mary Jean Harrold and Alessandro Orso. Retesting software during development and maintenance. In *2008 Frontiers of Software Maintenance*, pages 99–108. IEEE, 2008.

[5] Walter Cazzola, Sudipto Ghosh, Mohammed Al-Refai, and Gabriele Maurina. Bridging the model-to-code abstraction gap with fuzzy logic in model-based regression test selection. *Software and Systems Modeling*, 21(1):207–224, 2022.

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

[7] Sonu Mehta, Farima Farmahinifarahani, Ranjita Bhagwan, Suraj Guptha, Sina Jafari, Rahul Kumar, Vaibhav Saini, and Anirudh Santhiar. Data-driven test selection at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1225–1235, 2021.

[8] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[9] 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.

[10] Lingming Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209. IEEE, 2018.

[11] 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 *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[12] Owolabi Legunsen, August Shi, and Darko Marinov. Starts: Static regression test selection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 949–954. IEEE, 2017.

[13] Bogdan Korel, Luay Ho Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 214–223. IEEE, 2002.

[14] Nan Ye, Xin Chen, Peng Jiang, Wenxu Ding, and Xuandong Li. Automatic regression test selection based on activity diagrams. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion*, pages 166–171. IEEE, 2011.

[15] Mohammed Al-Refai, Walter Cazzola, and Sudipto Ghosh. A fuzzy logic based approach for model-based regression test selection. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 55–62. IEEE, 2017.

[16] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. Fastlane: Test minimization for rapidly deployed large-scale online services. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE, 2019.

[17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[18] Mohammed Al-Refai. *Towards model-based regression test selection.* PhD thesis, Colorado State University, 2019.

[19] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational software architect: A tool for domain-specific modeling. *IBM systems journal*, 45(3):555–568, 2006.

## Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| | | |
|---|---|---|
| Ort | Datum | Unterschrift im Original |