Search-based Testing in Financial Applications

by

Mohammad Moein Almasi

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
September 2017

Thesis advisor | Author

Dr. Hadi Hemmati | **Mohammad Moein Almasi**

Search-based Testing in Financial Applications

Abstract

Automated unit test generation has been extensively studied in the literature in recent years. Previous studies on open source systems have shown that test generation tools are quite effective at detecting faults, but how effective and applicable are they in an industrial application? In this thesis, this question is investigated in two phases. In the first phase, I empirically investigate the effectiveness and applicability of existing automated unit test generation tools and techniques in an industrial financial application known as *LifeCalc* which is a life insurance products calculator engine owned by *SEB Life & Pension Holding AB Riga Branch*.

In the second phase, I focus more on the software characteristics of financial application domain. In this domain, many legacy applications exist as a collection of formulas implemented in spreadsheets. These legacy code, at some point, will be migrated to more modern development environment. However, migration of such code to a full-fledged system is an error-prone process. While small differences in the outputs of numerical calculations produced by the two artifacts are tolerable, large discrepancies could have serious financial implications. Therefore, in this phase, I introduce a novel specialized search-based unit test generation technique that seeks to uncover the deviation failures in the migrated code automatically.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Hadi Hemmati for his continues support of my Master study, for his patience, motivation, and immense knowledge. I would also like thank my co-advisor Dr. Rasit Eskicioglu and the thesis committee: Dr. Neil Bruce and Dr. Shaahin Filizadeh, for their insightful comments and encouragement.

I also thank my fellow labmates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last two years. Last but not least, I wish to express my cordial appreciation to my family who always support me during my studies and I believe I would not be in place I am without their sincere support and patience.

Publications

Some ideas and figures in this thesis have previously appeared in the following publications by the author:

1. M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application." In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pp. 263-272. IEEE Press, 2017. (**IEEE Software Award: Software Engineering in Practice Best Paper Award**)

2. M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Phil McMinn, and Janis Benefelds. "Search-Based Detection of Deviation Failures in the Migration of Legacy Spreadsheet Applications." In *Proceedings of the 40th International Conference on Software Engineering*. IEEE Press, 2017. (**Submitted**)

To my mom who has been my constant source of inspiration.

Chapter 1

Introduction

Software testing is an essential part of software development processes to assure the quality of software systems. Developers typically underestimate the required testing effort, and therefore developer-written tests are generally not comprehensive [6]. To overcome the challenges of manual test generation, automatic techniques and tools based on different approaches have been introduced (e.g., [18, 22, 41]). Techniques such as random testing [22, 41, 42], search-based testing [1, 15, 26, 37], or symbolic testing [7, 20, 21, 54] can effectively generate test inputs automatically. For instance, search-based approaches [38] use techniques such as genetic algorithms to transform software testing problems into optimization problems, where the objective of the test generation process is implemented by a fitness function that guides the search.

In order for developers to adopt these tools, it is important to provide an understanding of their capabilities and the quality of the tests they produce. A common way in the literature to evaluate generated test suites is by measuring their code coverage (e.g., [16]). However, code coverage as a measure of test effectiveness has been challenged by several recent studies (e.g., [23, 27]). Alternatives are to evaluate the ability of tools to detect real faults (e.g., [17, 46]) or artificial faults (e.g., mutation analysis [19]). Another shortcoming of empirical studies on automated test generation tools is that they only focus on open source projects. Evaluations on industrial systems (e.g., [16]) are still rare, possibly because they require a level of maturity of the underlying tools that is difficult to achieve for research prototypes. Consequently, there is a need to provide further evidence of the capabilities of automated unit test generation on industrial systems.

This thesis consists of two main phases: 1) The aim of the first phase of this thesis is to evaluate the existing automated test generation techniques on a relatively complex, production ready financial application known as *LifeCalc*, owned and developed internally by *SEB Life & Pension Holding AB Riga Branch*. I selected a search-based test generation tool (EvoSuite [18]) and a random test generation tool (Randoop [41]) for experimentation, and delivered the following concrete contributions in the first phase of this thesis:

- I describe the results of an experiment using 25 real faults of the *LifeCalc* industrial application to assess the effectiveness of automatically generated test suites in terms of detecting real-world faults.
- I analyze the undetected faults in *LifeCalc* in order to guide future research on automated test generation tools.
- I provide general lessons learned from the application of the automated unit test generation to industrial software.

The experiments with the *LifeCalc* application reveal that EvoSuite detected 56.40% of the faults, and Randoop 38.00%. A closer investigation of the undetected faults shows that 97.62% of them depend on either "specific primitive values" (50.00%) or the construction of "complex state configuration of objects" (47.62%), and this can guide future research towards improved techniques for detecting such defects. My interactions with the developers of *LifeCalc* reveal several aspects of the test generation research prototypes that inhibit a successful technology transfer from academia to industry, which can guide testing researchers to achieving impact in practice.

2) In the second phase of thesis, I focus on special characteristics of the application under test. In this phase of this study, I target the financial applications, in particular, which are mostly programs to calculate different financial formulas. Spreadsheets were traditionally used in financial corporations to perform these complex calculations. For many reasons such as demands on performance, precision, support for automation, conformance with other in-house tools and frameworks, changes to a corporation's IT's strategic plans, etc. a corporation may be forced to migrate from spreadsheets to more generic solutions such as web-based services or applications. The re-implementation of the legacy spreadsheet applications may, however, introduce faults. If such faults cause the outputs of the calculations performed by the re-implementation to deviate from the outputs of the calculations of the original spreadsheet — i.e., a *deviation failure* occurs — then this could have serious financial implications.

Testing [45] is an important approach to identify such deviation failures, but finding specific inputs that reveal deviation failures is a challenging task. In particular, in financial calculations there can be a small acceptable and inconsequential deviation in the outputs known as *tolerable threshold*. The challenge thus lies in finding inputs that produce deviations that are larger than a specified tolerable threshold. As discussed in the first phase, automation is often used to support testers in challenging testing activities.

However, a general limitation of automated test generation is that it relies on the existence of a test oracle [5] that can decide whether the system is behaving correctly for an automatically generated test input. This is particularly challenging for complex applications such as financial applications, where it is difficult to determine what the correct output should be for a specific input. However, in the case of re-implementations of legacy spreadsheet programs, the legacy spreadsheets specify the expected behavior, and can thus serve as test oracles. Therefore, in the second phase of this thesis, I introduce a new automated search-based test generation approach that aims to find tests that maximize the deviation failures between a given spreadsheet and its re-implementation.

The main fitness function in my scenario is the output deviation of the root (top level) formula's return values in Excel compared to its Java re-implementation. I refer to this approach as the Output-based Search Technique (OST). To improve fault finding and to help localize faults, I also propose a novel spreadsheet-based technique, referred to as the Spreadsheet-based Search (SST), which concentrates the search on lower-level sub-formulas. The original motivation for my research came from the company reporting a substantial amount of manual effort that had to be invested to identify deviation failures in this Java reimplementation — since deviation failures were typically missed by the developer-written unit test cases, and could only be detected by a series of expensive manual acceptance tests conducted by business analysts.

I evaluate the proposed techniques using a total of 40 formulas with previously known and unknown faults from two products developed by *SEB Life & Pension Holding AB Riga Branch*. The first product, a financial calculator engine known as *LifeCalc*, is a newly implemented life insurance and pension products calculator engine written in approximately 80,000 lines of Java code. The second product is called *PensionPlanner*, and is a pension funds calculator [29] developed using Java technology stack with around 170,000 lines of code. These products are produced in order to replace the legacy Excel spreadsheets calculators (see Section 4.3.2 for more details about the subjects of study).

The empirical experiments with *LifeCalc* show that the top-level search-based approach (OST) correctly revealed 46.67% of the deviation failures, which represents a solid 21.67% improvement over a baseline of random testing and 10.01% over test cases optimized for branch coverage. Using the advanced SST method, a further 23.33% of the deviation failures are detected. Finally, relaxing time constraints and increasing the search budget led to 90% deviation failure detection. Applying this approach on *PensionPlanner*, I also managed to detect three new deviation failures that were not detected by the manual test cases.

The contributions of the second phase of this thesis are, therefore, as follows:

- A new search-based test generation approach, the Output-based Search Technique (OST), that aims to detect deviation failures.
- A modified fitness function that utilizes spreadsheets' nested structure of formu-

las, implemented into an approach called the Spreadsheet-based Search Technique (SST), to improve fault finding and their subsequent localization.

- The evaluation of the proposed OST and SST approaches using 40 formulas with real deviation failures (known and unknown), in two commercial financial applications. As part of this evaluation I compare the success rates and actual deviation values of my approaches with automatically generated test cases for branch coverage of the Java code, randomly generated test cases, as well as manual, developer-written test cases.

This thesis is organized as follows: Chapter 2 provides necessary background information. Chapter 3 presents empirical investigation of the effectiveness and applicability of existing automated unit test generation techniques which is the first phase of this thesis. Chapter 4 discusses the second phase of this thesis which focuses on the characteristics of financial applications. Finally, Chapter 5 concludes this thesis and provides future directions.

Chapter 2

Background

As matter of fact, there are two major approaches in order to test a software system. The first approach is the conventional manual testing in which it is not only laborious but also time consuming and also it requires a lot of effort in order to test a software thoroughly. Moreover, it is difficult in order to detect a fault in this approach. The second approach is automated testing, this approach is not only requires less resources but also it facilitates comprehensive testing of a software with minimal effort. In this study, I focus on automated test generation techniques such as Random and Search-based testing techniques in which I have provided the necessary background information in this chapter.

2.1 Automated Unit Test Generation

In object-oriented programming, a unit test is a small, executable piece of code that exercises a functionality of a class under test. While there is a wide range of techniques to automatically generate tests in general, the specific case of unit test generation is mainly addressed by approaches based on random generation of call sequences, search-based optimization of call sequences, and symbolic approaches.

2.1.1 Random Testing

Random testing [3] is perhaps the most basic and straight forward form of test generation [12], as it consists of invocation of functions with random inputs. Guided random testing is a refined approach that starts with random input data and then uses some form of extra knowledge to produce further input data. One of the main examples of this category is feedback-directed random testing [42]. Feedback-directed random test generation enhances random test generation by incorporating feedback collected from executing test inputs that is used to avoid generating duplicate and illegal input data. This technique takes a set of classes as input and creates method sequences incrementally. It iteratively executes new sequences and checks them against general contracts. Sequences that violate the contract are considered as failing tests, and sequences that pose normal behavior are treated as regression tests.

2.1.2 Search-based Testing

Search Based Software Engineering is an approach that transforms software engineering problems into optimization problems [37], where the objective of the test generation is implemented by a fitness function that guides the search. Among the many meta-heuristic search techniques used for test generation, Genetic Algorithms are perhaps the most common [37]. In a Genetic Algorithm, randomly selected candidate solutions are evolved by applying evolutionary operators, such as mutation and crossover, resulting in new offspring individuals, with better fitness values. An example objective function for unit test generation is the whole test suite's code coverage [15].

2.1.3 Symbolic Testing

Symbolic approaches represent execution paths through a program as constraints on the input values. A common approach is *Dynamic Symbolic Execution* (e.g., [22]), where the paths of a program are systematically explored by iteratively negating one branch condition in a path constraint at a time, and using a constraint solver to generate a new test input for that path. Most approaches of this kind target generation of specific input data and require manual construction of test drivers. The PEX [49] tool expects these drivers in the form of parameterized unit tests, and there are related approaches that aim to explore possible sequences of method calls (e.g., [48, 52]). These approaches are promising in detecting faults, Shamshiri *et al.* [46] has evaluated the effectiveness of these approaches in detecting fault in open source applications. Moreover, Fraser and Arcuri [16] conducted a comprehensive evaluation of search-based generation technique on 110 open source programs and reported challenges they had due to practical limitations such as environmental dependencies.

2.2 Test Oracles

Most of the current test generation techniques are incapable of generating good assertions [46]. Assertion is a `boolean` expression that checks the behavior of program at runtime. An assertion requires test oracle to evaluate the program output. As a matter of fact, most test generation techniques are able to generate test data but an oracle is not always available for the generated test. An oracle [5] is being used to ensure that the generated outputs are the correct ones. Without an automated oracle the automated tests won't save much time (compared to fully manual testing) due to the human involvement it the test result evaluation phase of test execution.

A pseudo-oracle [11] is an alternative version of the program under test which is produced independently, e.g., by a different programming team or written in an entirely different programming language. Pseudo-oracle has been introduced by Davis and Weyuker to test non-testable programs [50].

For instance, in the financial ecosystem, spreadsheets tools such as Microsoft Excel are widely used and they are treated as development environments for non-professional programmers [32] In situations like our case study, where the actual oracle is not available the excel forms (pseudo-oracle) can be used.

McMinn [39] has introduced a novel pseudo-oracle by utilizing testability transformation. Testability transformation is source-to-source program transformation to make the program under test more testable [24, 25]. Basically, the original program, which has no test oracle, will be automatically transformed into another program with the same functionality. Then the test cases aim at fining differences in the outputs of the two programs.

In a recent work, Matthew Patrick *et al.* [43] utilized pseudo-oracles in their new search-based technique for testing various implementations of stochastic models with the intention of maximizing the differences between the original implementation and its respective pseudo-oracle. They have used Kolmogorov-Smirnov tests to compare the distributions of outputs from each implementation and concluded that their technique reduces the testing effort and also enables discrepancies, where they could have been overlooked.

There are other approaches than pseudo-oracle to test programs without a test oracle, such as Metamorphic testing [8, 56]. Metamorphic testing predicts future outputs using knowledge of previous outputs and sets of metamorphic relations, which are necessary properties of software under test. Metamorphic relations are properties that aim to generate new test cases from existing ones [57]. These properties are presented in a form of relations between inputs and outputs of the program, which can be used to validate the correctness of the software under test [34]. For instance, if the method under test is Sine function we know that *Sine(x+PI)=-Sine(x)*. Therefore, one metamorphic relation for Sine method can be adding PI to any test input x we already have for Sine and making sure that the above property holds.

Metamorphic testing has been extensively studied in the literature [33, 40] and several studies [28, 55] compared metamorphic testing with other approaches such as assertion checking.

Chapter 3

An Industrial Evaluation of Unit Test Generation

Automated unit test generation has been extensively studied in the literature in recent years. Previous studies on open source systems have shown that test generation tools are quite effective at detecting faults, but how effective and applicable are they in an industrial application? In the first phase of this thesis, I investigate this question using a life insurance and pension products calculator engine owned by *SEB Life & Pension Holding AB Riga Branch.*

To study fault-finding effectiveness, I extracted 25 real faults from the version history of this software project, and applied two up-to-date unit test generation tools for Java, EvoSuite and Randoop, which implement search-based and feedback-directed random test generation, respectively. Automatically generated test suites detected up to 56.40% (EvoSuite) and 38.00% (Randoop) of these faults. The analysis of results demonstrates challenges that need to be addressed in order to improve fault detection in test generation tools. In particular, classification of the undetected faults shows that 97.62% of them depend on either "*specific primitive values*" (50.00%) or the construction of "*complex state configuration of objects*" (47.62%).

To study applicability, I conducted a qualitative study to understand developer's expectations about the test generation tools and the generated test cases. This leads to insights on requirements for academic prototypes for successful technology transfer from academic research to industrial practice, such as a need to integrate with popular build tools, and to improve the readability of the generated tests.

Followings are the concrete contributions of the first phase of this thesis:

- I describe the results of an experiment using 25 real faults of the *LifeCalc* industrial application to assess the effectiveness of automatically generated test suites in terms of detecting real-world faults.
- I analyze the undetected faults in *LifeCalc* in order to guide future research on automated test generation tools.
- I provide general lessons learned from the application of the automated unit test generation to industrial software.

This chapter is organized as follows. Section 3.1 states the experiment setup. Section 3.2 presents results. Section 3.3 discusses lessons learned. Section 3.4 reviews related work. Section 3.5 concludes the first phase.

3.1 Experiment Setup

In this section, I describe the experiment methodology. The main objective of this study is to evaluate the effectiveness of existing mature test generation tools, in terms of revealing known real faults in an industrial application, and understanding the existing barriers for practitioners when adopting these tools.

3.1.1 Research Questions

To achieve the mentioned goal, I have designed an experiment to answer the following research questions:

RQ1: *How effective are automatically generated unit tests in terms of finding real faults?* This question intends to assess the capability of two test generation tools that are widely adopted in academia, in terms of revealing real faults.

RQ2: *What categories of faults are harder to detect using the current automated test generation tools?* This question aims to categorize the faults that have not been detected by any of the examined test generation tools.

RQ3: *What major barriers do developers see when adopting automatic test generation tools?* This question tries to identify some of the practitioner's requirements which are not currently supported by automatic test generation tools. (Due to resource limitations I only consider EvoSuite for *RQ3*)

3.1.2 Subject of Study

I have conducted comprehensive experiment on a life insurance and pension products calculator engine known as *LifeCalc*, which is a standalone software component owned by *SEB Life & Pension Holding AB Riga Branch*. *LifeCalc* is written using the Java technology stack. Its implementation started in early 2015 and it has been released to production in early 2016. *LifeCalc*'s development team consists of 5 developers, 2 testers and 1 business analyst/project manager. *LifeCalc* is a medium-sized application that consists of complex critical calculations with plenty of business rules that are implemented using complex conditions, which makes it challenging for test generation tools. *LifeCalc*'s stacks comprise of front-end (client) and back-end (core, services) modules with approximately 80,000 LOC.

LifeCalc is built in a nightly build using the Jenkins [30] Continuous Integration (CI) build management system, and all tests are executed during the nightly builds. Moreover, builds are also triggered on demand if there are any critical bug fixes that need to be deployed to production. In these cases, there is a restricted subset of important tests which will be executed to test the critical functionalities of the application. The company's developers provided us 25 faults from their issue tracking system. They selected these faults randomly, trying to include examples from different times throughout the life cycle of the project. I studied the commits in the version repository that contained the fix for the fault until I understood them well enough to know how to replicate them by writing manual unit tests. For each fault, I extracted the faulty and fixed program versions, such that they differ by a minimal change that demonstrates the isolated fault fix.

3.1.3 Fault Analysis

Based on the fault descriptions, I distinguish between the following two types of faults in the dataset: 1) Specification-based faults and 2) Exception-related faults. The first category requires knowledge about the expected logic (specification) vs. the existing implemented logic. This means that, in most cases, a JUnit assertion is required to detect the fault in the faulty version of the program. The second category consists of faults that can be detected without knowing the exact logic of the code. In other words, the unit test's failure is due to an unhandled exception thrown in the code under test, and not about a failing assertion in the test.

In the rest of this section I explain these categories with an anonymized code snippet example per category (the anonymization is due to the agreement with *SEB Life & Pension Holding AB Riga Branch* for any kind of publications).

Specification Faults

Among the collected 25 faults, I identified five as specification faults because the expected business logic was not implemented correctly. For instance, the code snippet from *LifeCalc-b23*, shown below, is one of the *Specification Faults*, in which the assigned developer was retrieving tariff values yearly, whereas tariff values in the properties file are defined on a monthly basis, and thus `param1` needs to be multiplied by 12. One way to detect this fault is to have an assertion in the generated tests on the fixed version to check the value of `param3`.

```
public double faultyMethod(int param1, int param2) {
  ...
  double param3 = 0.0;
  //Faulty Statement
  - param3 = Double.valueOf(PropertiesReader.getProperty("mt.m[" + param1
  + "]")) * Math.pow((1 + param2), -1);
  //Fixed Statement
  + param3 = Double.valueOf(PropertiesReader.getProperty("mt.m[" + param1
  * 12 + "]")) * Math.pow((1 + param2), -1);
  ...
  return param3;
}
```

Listing 3.1: Example of a specification fault in this study

Exception-related Faults

I identified 20 faults in this category in the pool of 25 faults. I encountered several types of common exceptions such as `NullPointerException (NPE)` (thrown when an application attempts to call methods on a `null` object instance), `ArithmeticException` (thrown when an exceptional arithmetic condition has occurred) and `NumberFormat Exception` (thrown to indicate that the application has attempted to convert a string with invalid format to one of the numeric types). However, most of the faults from this category were due to `NPE`s. The following code snippet from *LifeCalc-b18* shows one of the exception throwing faults: An exception is thrown due to an invalid parameter value (property key) of the getProperty method. Therefore, `PropertiesReader.getProperty(invalidKey)` returns a `null` value, which then causes the program to throw a `NPE` on conversion of the `String` to a `Double` object.

```java
public void faultyMethod(ObjectX objx, String param1, String param2) {
  ...
  if(param1.equalsIgnoreCase(Enum1.P_012.getValue())){
  //Faulty Statement
  Double rate = Double.valueOf(PropertiesReader.getProperty("rate_" +
  param2 + "")) * objx.getObj().getPaymentFrequency();
  }
  ...
}
```

Listing 3.2: Example of an exception-related fault in this study

3.1.4 Automated Unit Test Generation Tools

As mentioned earlier, *LifeCalc* is written in Java, and therefore I had to consider test generation tools for Java. For the Java programming language, there are mature tools that can generate JUnit test cases using random testing (e.g, Randoop) and search-based testing (e.g., EvoSuite). However, for symbolic approaches, usually research prototypes in Java only generate test data, and not JUnit test cases [10] (i.e., testers have to manually write test drivers for those symbolic tools for each single class). Therefore, I only selected tools from the categories of random unit test generation and search-based unit test generation. Randoop [41] and JCrasher [9] are instances of random unit test generation tools. I decided to use Randoop as it is one of the most used random test generation tools in academia, and it is still being actively maintained. EvoSuite [18] and TestFul [4] are representative of evolutionary test generation tools which apply search techniques in order to optimize test suites based on various coverage criteria. EvoSuite was chosen as it is actively being maintained and extended, and ranked first in recent SBST tool competitions (e.g., see [14]).

Randoop

Randoop [41] is one of the most stable random test generation tools, with easy to follow instructions to get it up and running in short time. Randoop implements feedback-directed random test generation, by generating sequences of method invocations for all the classes under test. In other words, it builds test inputs incrementally, and then the newly created test inputs extend previous ones. As soon as these test inputs are created, they will be executed and the results collected from these executions will be used to guide the generation of new ones. Randoop can be used for both fault-detection and regression testing. For regression testing, the tests contain assertions that capture the current state. For fault detection it checks various predefined or custom contracts, and the violation of any of these contracts indicates an error in the tested classes. Randoop requires the user to provide a list of classes under test. For the experiments in the first phase of this thesis, I had to manually identify a list of classes that are the dependencies of the faulty class, for each of the 25 analyzed faults. I used all the default settings, except for the stopping criterion, for which I used two configurations: The default setting of 3 minutes, and an increased duration of 15 minutes.

EvoSuite

EvoSuite [18] is a search-based unit test generation tool for Java that uses a genetic algorithm to evolve a set of test cases with the intention of maximizing code coverage. EvoSuite initially generates random sets of test cases and then uses evolutionary search operators such as selection, mutation, and crossover to improve the generated test cases. This evolution process is guided by a fitness function calculated based on various coverage criteria [44]. EvoSuite then performs optimizations with respect to the defined coverage criteria on the test suite with the highest coverage. Ultimately, it enforces sanitization checks to ensure the generated tests are valid and executable. In this experiments, I executed EvoSuite on the faulty classes with the branch coverage criterion as fitness function. I applied the same stopping criteria (i.e., 3 and 15 minutes) as I used for Randoop.

3.1.5 Test Generation Scenario

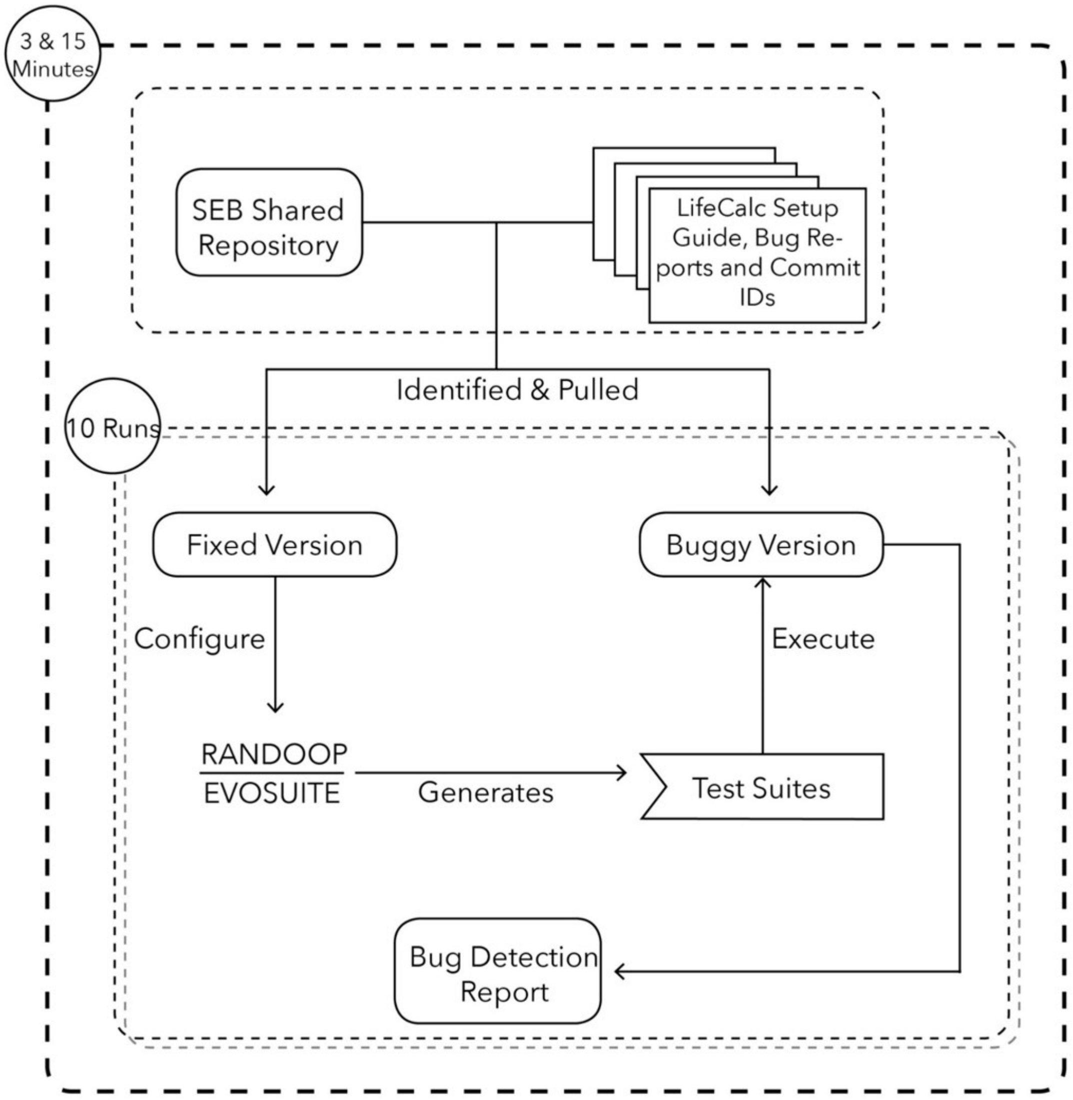
I generated the tests on the fixed versions so that the automatically generated JUnit assertions are based on the correct implementations. Generating tests on the fixed version is useful in the context of regression testing, and allows us to simulate whether the specification faults can be detected in such a testing scenario. For exception-related faults I could have also generated the tests directly on the faulty version, since throwing of exceptions can be used as an automated oracle. In other words, the current experiment design makes more sense in the context of regression testing, where one needs to create a regression test suite that passes in the current version, to be used to guard from future faulty changes. However, as the faulty and fixed versions differ only in terms of the fault fix and the interfaces are identical, I expect that results on exception-faults would be similar, if tests were generated directly on the faulty version.

3.1.6 Experiment Procedure

The overall experiment procedure is as follows (See Figure 3.1):

- First, I extracted both fixed and faulty versions of *LifeCalc* based on the identified commit IDs provided by *SEB Life & Pension Holding AB Riga Branch*.
- Then, for each fault, I generated test suites using both, EvoSuite and Randoop, on the fixed version.
- To determine whether a fault was found, I executed all generated test cases on the corresponding *LifeCalc* faulty version. These executions were done manually using the Eclipse IDE. A test case is considered to detect the fault if it fails on the faulty version. This failure can be due to an exception in the executed classes or a JUnit assertion failure in the tests.
- In order to accommodate for the randomness of the test generation tools, each tool was executed 10 times for each fault. *RQ1* uses two set ups for test generation budget (3 and 15 minutes), which will be discussed more in the next section.
- I collected all the statistics from the execution logs and manually verified the validity of the failing test cases, in order to avoid possible false positives [46].

Figure 3.1: The overall experiment procedure



The measure that I use to assess the effectiveness of the test suites is the percentages of the runs (how many out of 10) that detected the fault. I also aggregate these by averaging over several versions (faults).

3.1.7 Qualitative Study Procedure

Our participants in the conducted study were the five developers of *LifeCalc* with different level of expertise, varying from 1 to 8 years of working with Java technology stack, and familiarity with the application code base. The participants were provided with a qualitative study package (Appendix A) containing sets of tasks to perform and their respective guidelines.

Based on the analysis of the fault effectiveness experiment, I gave the participants tasks such as executing the generated tests, and trying to debug and locate a fault (I gave each developer three faults of different level of difficulty, as determined in the RQ2 analysis; see Section 3.2.2). In addition, I asked them to write a manual test that covers the same faulty code as a generated test to better understand how generated tests relate to developer preferences. They were also provided with comprehensive guidelines including all the necessary commands to run and generate tests using EvoSuite.

After performing these tasks, the developers answered a questionnaire containing seven demographic questions, four questions using Likert-scale to rate aspects of tools and generated tests, and six free-text questions. See the discussion in Section 3.2.3 for details on the questions.

3.2 Experiment Results

In this section, I discuss the results of the experiments and answer the research questions presented in Section 3.1.1.

3.2.1 Effectiveness of Automatically Generated Unit Tests

RQ1*: How effective are automatically generated unit tests in terms of finding real faults?*

The results of test suite executions are summarized in Table 3.1. I first consider the 3 min. scenario: The first observation is that both tools can find some of the faults, and are unsuccessful at detecting the others. The average fault detection rate is not particularly high (50.80% in EvoSuite vs. 36.80% in Randoop). However, the variation of effectiveness for different faults, which I can see in Table 3.1, is a more interesting observation. There are cases (like *LifeCalc-b4*, *LifeCalc-b7*, etc.) where none of the tools can detect the fault even in ten executions. On the other hand, there are cases (like *LifeCalc-b6*, *LifeCalc-b15*, *LifeCalc-b18*, etc.) where every single test suite out of the 10 per tool can detect the fault.

Given the results, one follow up question is "Can I improve the effectiveness of the tools by allocating more search budget to them?". To answer this question, I also ran the experiment of *RQ1* with the longer stopping criterion of 15 minutes. Note that this experiment is not meant to be a thorough study on the correlation between testing budget and effectiveness. The goal is just to have an idea on what can one expect by adding extra resources to these tools. A more thorough study would be an interesting future work. In total, 19 out of 25 faults were found. However, as the tools are both based on randomized algorithms, these faults were not found in all of the 10 runs. On average, Randoop can find faults in 36.8% of the runs, whereas EvoSuite in 50.8% of the runs. While increasing the search budget is useful for EvoSuite (+5.6%), it had only a moderate effect on Randoop (+1.2%).

Table 3.1: Fault detection rate of EvoSuite and Randoop for each fault over 10 executions per fault

Fault	**EvoSuite** (%)		**Randoop** (%)	
	3 min	15 min	3 min	15 min
LifeCalc-b1	40	+0	0	+0
LifeCalc-b2	30	+10	10	+0
LifeCalc-b3	60	+0	30	-10
LifeCalc-b4	0	+0	0	+0
LifeCalc-b5	100	+0	90	+10
LifeCalc-b6	100	+0	100	+0
LifeCalc-b7	0	+0	0	+0
LifeCalc-b8	20	+10	0	+0
LifeCalc-b9	80	+0	40	+10
LifeCalc-b10	0	+0	0	+0
LifeCalc-b11	60	+0	10	-10
LifeCalc-b12	0	+0	0	+0
LifeCalc-b13	70	+30	90	+10
LifeCalc-b14	50	+10	0	+0
LifeCalc-b15	100	+0	100	+0
LifeCalc-b16	30	+20	0	+0
LifeCalc-b17	80	+20	90	+10
LifeCalc-b18	100	+0	100	+0
LifeCalc-b19	60	+20	70	+10
LifeCalc-b20	60	+0	20	-10
LifeCalc-b21	0	+0	0	+0
LifeCalc-b22	0	+0	0	+0
LifeCalc-b23	100	+0	70	+10
LifeCalc-b24	100	+0	100	+0
LifeCalc-b25	30	+20	0	+0
Average	50.80%	+5.60%	36.80%	+1.20%

RQ1: Existing tools can potentially detect most of the faults (19 out of 25 were detected at least once). But there are also some faults (6 out of 25) that are never found within the explored search budgets.

Given the variations in the results per fault, it would be interesting to see what kind of faults are easier to detect and on which faults the tools have difficulties. Therefore, in *RQ2* I explore this question.

Table 3.2: Percentage of detected faults per category over all executions using EvoSuite and Randoop, + indicates the fault detection rate is increased

Fault Category	**EvoSuite** (%)		**Randoop** (%)	
	3 min	15 min	3 min	15 min
Specification Faults	46	+4.00	34.00	+2.00
Exception-related Faults	52	+6.00	37.50	+1.00

3.2.2 Analysis of Faults Not Detected

RQ2: *What categories of faults are harder to detect using the current automated test generation tools?*

As discussed in detail in the fault analysis section (Section 3.1.3), I categorized the faults into two types: 1) Specification faults and 2) Exception-related faults. Table 3.2 shows the percentages of faults that are detected in each category. The results show that, as expected, the detection rate is higher with 15 minutes search budget. However, looking into the *Exception-related Faults* I can see that there is quite a variation between the effectiveness of test suites on different faults (Detailed summary of all execution shows the fault category for each faulty version of *LifeCalc* presented by Table 3.4). Therefore, I next try to characterize the faults in the exception-related faults category.

I look at the faults in this category in three subclasses: 1) Easy Faults (that are detected by both tools in at least 80% of times, with 3 and/or 15 minutes stopping criterion), 2) Hard Faults (that are detected at least once by one tool and are not "easy faults"), and 3) Challenging Faults (that are never detected by either tools). The next subsections describe these faults with anonymized code examples from the industrial system.

Easy Faults

In the *Easy Fault* category, the faulty statement does not require satisfying a complex condition prior to its execution. In other words, those are faults that do not require specific input data and they do not depend on complex conditions. For instance, the following code snippet represents a simple `NullPointerException` `(NPE)` in *LifeCalc-b5*:

```
public void faultyMethod(ObjectX objx) {
  if(!StringUtils.isEmpty(objx.getObj().getLocale)){
  //Faulty Statement
  Double interest =
  Double.valueOf(PropertiesReader.getProperty("interest.rate.A_" +
  objx.getObj().getLocale() + ""));
  }
}
```

Listing 3.3: An Easy Fault

Detecting this fault requires the invocation of `faultyMethod` with an `ObjectX` instance that has a non-empty locale attribute `String` with invalid value. As the invalid property key `interest.rate.A_InvalidKey` would be missing in the property file, there will be a `NPE` when converting a `null` `String` to `Double` object. Table 3.3 shows the same results of Table 3.1 (for the exception-related faults only) grouped by the fault difficulty categories. Of the Easy Faults 87.14% were detected by EvoSuite and 91.43% by Randoop. Overall, in this category, EvoSuite found the fault in 61 out of 70 cases, while Randoop performed slightly better by detecting faults in 64 out 70 times. The following is a test case generated by EvoSuite for an easy fault:

```
public void test17()  throws Throwable  {
  FaultyClass faultyClass0 = new FaultyClass();
```

Table 3.3: Fault detection (%) variation over 3 and 15 minutes search budget in different fault categories, + indicates that fault detection rate is increased while - indicates the fault detection rate is decreased

Category	**EvoSuite** (%)		**Randoop** (%)		Faults(#)
	3 min	15 min	3 min	15 min	
Easy	87.14	+10.00	91.43	+5.71	7
Hard	47.78	+5.55	12.22	-2.22	9
Challenging	0.00	0.00	0.00	0.00	4

```
    ObjectX objx = new ObjectX();
    Object obj0 = new Object();
    obj0.setProdCode("012_200");
    obj0.setCoverSumLife(12);
    obj0.setLocale("012_200");
    objx.setObj(obj0);
    faultyClass0.faultyMethod(objx);
}
```

Listing 3.4: Generated Test Case for An Easy Fault by EvoSuite

In this specific example, the cases where the fault had not been detected was due to the fact that a *locale* object was not initialized. Since there is a simple `null` check condition prior to the faulty statement, the faulty statement was not covered. Randoop generated, on average, around 14,500 test methods on each run for this particular fault (*LifeCalc-b5*) while EvoSuite generated 18 test methods. Although most of the methods generated by Randoop do not cover this fault, there is at least one method that covers this fault. The following is a sample test generated by Randoop that detected the fault:

```java
public void test4() throws Throwable {
  FaultyClass var0 = new FaultyClass();
  Objectx var1 = new Objectx();
  Object var2 = new Object();
  var2.setLocale("hi");
  var1.setObj(var2);
  var0.faultyMethod(var1);
}
```

Listing 3.5: Generated Test Case for An Easy Fault by RANDOOP

Hard Faults

This category describes faults that are either surrounded by conditions which require specific primitive values, or the faulty statement itself requires specific input data. The mentioned primitive values are not only explicit inputs of the tests, but also attributes of objects passed to the tests. The following code snippet shows a fault from the *Hard Fault* category.

```java
public List<Double> faultyMethod(int param1, int param2, String param3,
String param4) {
  List<Double> list = new ArrayList<Double>();
  if (param3.equalsIgnoreCase(Enum1.POSITIVE.getValue())) {
  for (int i = 1; i <= param1 * 12; i++) {
  if (param4.equalsIgnoreCase(Enum2.LOW.getValue())) {
  // Faulty statement
  list.add(i,Math.pow((1+Double.valueOf(PropertyReader.getProperty("inv_min"))),(
  Double.valueOf(1)/Double.valueOf(12)))-1);
  }
  }
  }
}
```

Listing 3.6: Example of a Hard Fault

The faulty method contains an `NPE` in *LifeCalc-b8*. To detect the fault using the generated tests, it needs to satisfy two conditions (lines #4 and #6) that require a specific string input. Note that the two nested if conditions require test inputs that can be extracted from `enum` values. Line #8 is the faulty statement in which the property value of `inv_min` is an invalid key in the properties file. This type of faults may occur due to two reasons: 1. the program tries to get a value of an invalid key in the properties file (*the code is faulty*). 2. The valid property key is missing in the properties file (*fault in the properties file*). The second case happens because the business analysts are able to change the property file values directly, and previously there was no sanitization check in place to validate the properties files. Therefore, due to human error, there could be properties with no or wrong values.

The *Hard Faults* category is where search-based approaches have the most advantages (47.78% for EvoSuite vs. 12.23% of Randoop), because they can focus on generating corner cases. EvoSuite found *Hard Faults* in 43 out of 90 executions while Randoop only detected such faults in 11 out of 90 executions. The following is an example of a test method generated by EvoSuite for *LifeCalc-b8* which is a *Hard Fault*. The generated input data is able to satisfy both conditions and detects the fault. In the process of test data generation, EvoSuite has extracted the required input data to satisfy both conditions in line #4 and #6 in the above listing from the values existed in the `enum`.

```java
public void test2()  throws Throwable  {
  ...
  FaultyClass faultyClass0 = new FaultyClass();
  faultyClass0.faultyMethod(10,4,"positive","low");
  ...
}
```

Listing 3.7: Generated Test Case for An Hard Fault by EvoSuite

On the other hand, Randoop generated approximately 12,500 test methods on average per execution and none of the generated test methods were able to satisfy the conditions, mainly because they did not provide the required test data. The following is an example of a test generated by Randoop:

```java
public void test352() throws Throwable {
  ...
  FaultyClass var0 = new FaultyClass();
  java.util.List var1 =
  var0.faultyMethod(1,2147483,"20160419_7232","c$$u#");
  ...
}
```

Listing 3.8: Generated Test Case for An Hard Fault by Randoop

Challenging Faults

The third category of faults I call *Challenging Faults*. In this type of fault, the faulty statement is usually surrounded by complex conditions that requires constructing complex objects that are populated with specific values. For instance, in the following example, line #11 is the faulty statement in which `getPaymentFrequency()` can have the potential value of 0 which causes an arithmetic fault of a division by zero. The root-cause of this fault has to do with a new scheme for payment frequency that was denoted by value zero.

```java
public List<DateTime> faultyMethod(ObjectX objectx, DateTime a, DateTime b)
{
  ...
  List<DateTime> list = new ArrayList<DateTime>();
  Double a = 0.0;
  list.add(0, new DateTime());
  list.add(1, a);
  for (int i = 2; i <= Months.monthsBetween(a,b).getMonths(); i++) {
    if(objectx.getCalcCoverList().get("CoverCode").getPromilUWDate().isAfter(b)
     && objectx.getProdCode().equalsIgnoreCase("ProductCode") && i==2){
      list.add(i, list.get(i - 1).plusMonths(1));
      // Faulty statement
      a = (Days.daysBetween(a, list.get(i)).getDays() + 1) /
      objectx.getPaymentFrequency();
    }
  }
  ...
}
```

Listing 3.9: A Challenging Fault

Basically, the execution of the faulty statement requires satisfying a complex condition (line #8), which in turn requires specific data. It is considered complex since a randomly initialized `map` is unlikely to contain such specific keys and values. Once the outer condition has been satisfied, the faulty statement also demands a specific data which in this case, `getPaymentFrequency()`, is zero.

Based on the definition, the *Challenging Faults* are difficult for both tools, and were not detected at all. The following is a sample test case generated by EvoSuite for *LifeCalc-b4*. It constructed an empty object and set some of the basic attributes (i.e., `setProdCode()`) properly, but it failed to initialize and set a relatively complex `List`, `getCalcCoverList()`, which is a `map` containing object `Cover`:

```java
public void test10()  throws Throwable  {
  ...
  FaultyClass faultyClass0 = new FaultyClass();
  LocalTime localTime0 = LocalTime.now();
  DateTime dateTime0 = localTime0.toDateTimeToday();
  LocalTime localTime1 = localTime0.plusHours(27);
  DateTime dateTime1 = localTime1.toDateTimeToday();
  ObjectX objx = new ObjectX();
  objx.setProdCode("017_200");
  faultyClass0.faultyMethod(objx, dateTime0, dateTime1);
  ...
}
```

Listing 3.10: Generated Test Case for An Challenging Fault by EvoSuite

Randoop also partially constructed the input data, but because `var1` creates a new `ObjectX`, both `var1.getCoverEndDate()` and `var1.getBirthDateInsured()` return `null`. Given that `var5` and `var6` are *null*, the fault is not covered. The following is an example of a test generated by Randoop for *LifeCalc-b4*:

```java
public void test474() throws Throwable {
  FaultyClass var0 = new FaultyClass();
  ObjectX var1 = new ObjectX();
  DateTime var5 = var1.getCoverEndDate();
  DateTime var6 = var1.getBirthDateInsured();
  var1.setDiscount((java.lang.Double)10.0d);
  var0.faultyMethod(var1, var5, var6);
  ...
}
```

Listing 3.11: Generated Test Case for An Challenging Fault by RANDOOP

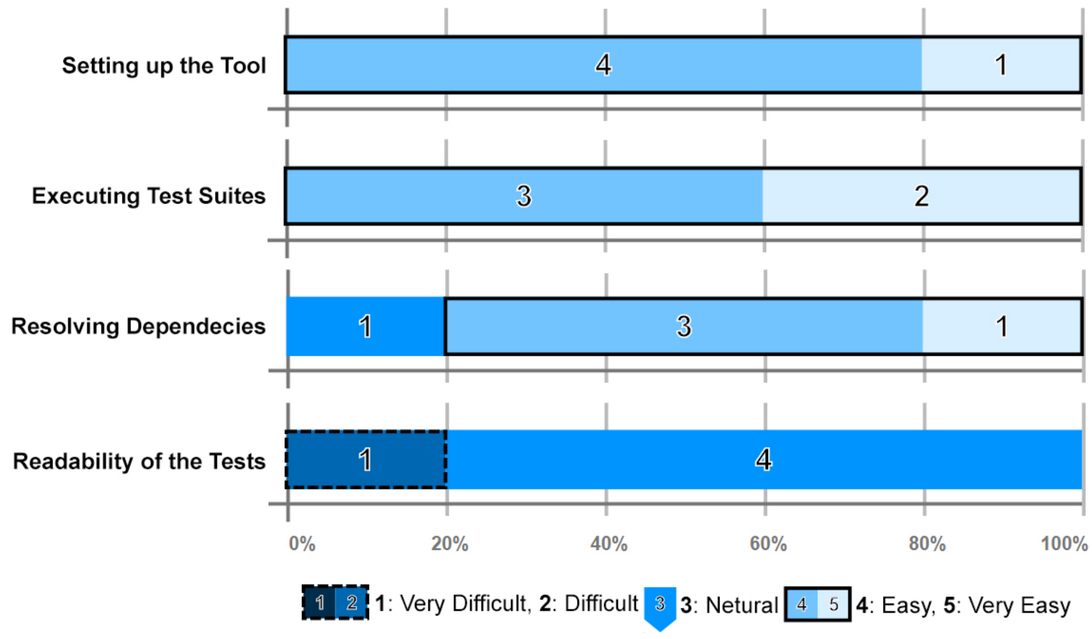
In order to detect faults from the mentioned categories, tools are required to improve their coverage and propagation, which means not only the faulty code needs to be covered, but also it has to be executed with a set of specific values in order to fail. I will discuss this in more detail in Section 3.3.

Finally, I look at the results per category when I add extra test generation budget to the tools. Table 3.3 shows the new results in +/- percentages. Increasing the search budget did not facilitate the detection of *Challenging Faults*. On the other hand, detection of *Easy Faults* increased by at least 5.71%, which can be due to the fact that *Easy Faults* are detected if the faulty statement is covered, even without requiring any specific input data or construction of complex object.

RQ2: Faults whose triggering requires generating input object data with complex states are hard to detect.

Up to 56% of the faults have been detected by the test generation tools, but those are mostly *Easy Faults*. I would like to understand the developer's point of view about the generated tests, specifically about the tests which failed to detect any fault. Therefore, to answer *RQ3* I conducted a qualitative study.

Figure 3.2: Likert chart that demonstrates the difficulty of the tasks.



3.2.3 Understanding Pitfalls of Test Generation Tools

***RQ3**: What major barriers do developers see when adopting automatic test generation tools?*

The main goal of qualitative study is to assess the applicability of test generation tools and automatically generated unit tests. I gave the participants certain tasks and asked them to answer the following questions (see Section 3.1.7 for more details):

1) *How difficult was it for you to set up* EvoSuite, *execute the test suite, resolve dependencies and read the generated tests?* They rated all of above tasks relatively easy except for the readability of the generated tests. In terms of building and resolving dependencies, EvoSuite uses Maven [35]) which they found really useful by saying

"... maven is definitely a plus point". Figure 3.2 reveals some concerns about the readability of the generated tests. As one of the developers said *"... it is hard to follow some of the generated tests"*.

The developers next had to read the bug report and re-execute the unit tests again, and try to debug and locate the bug using the given set of unit tests. Then, I asked them to answer the following question:

2) *How can the generated tests be improved?* In general, the developers did not like the generated assertions, as they said *"... poor assertions, sometimes there is an assertion and sometimes there is not? The assertions are mostly checking for simple stuff like list size and so on."*. In addition, they also did not like the generated input data. They described the generated data as *" ... set of extreme value test data with a correct datatype. They are not meaningful but complying with the method signature datatype"*. Some of the generated tests are readable but they are covering easy faults based on the fault categorization. On scale of 1 to 5, 1 being not helpful at all and 5 being very helpful, the developers rated the generated tests as sightly (4 developers) to moderately (1 developer) helpful.

I then asked them to manually write unit tests that cover the same code as the given generated tests (the rationale is that the following responses would not be purely subjective, and also this would help us to have a proper understanding of the tests) and answer the following questions:

3) *Describe what you like better about manually written tests than generated tests?* Developers prefer the test data (e.g., primitive values) used in the manually written tests. In addition, in manually written tests, the assertions are meaningful and useful

unlike the generated ones.

4) *Would you keep the generated unit tests?* They mostly answered no to this question. They specified that generated tests are not as good as manually written tests in terms of test data. In addition, either there is no assertion or if there is, it is not validating useful data. However, it can easily be modified to become a useful assertion. Finally, I wanted to find out their overall opinion about the automatic test generation and identify the advantages and the pitfalls from their point of view by answering the below questions:

5) *Given your current infrastructure setup, how would you like to have automated unit test generation framework integrated?* They emphasized that it is important to integrate these tools into their development and continuous integration (CI) environments. They said *"… supporting Jenkins is a must"*. In this case, EvoSuite provides plugins for both CI tools (Jenkins) and development tools (IntelliJ and Eclipse).

6) *What are the major barriers from your point of view in adopting automatic test generation tools?* Aside from assertions and test data that I mentioned earlier on, they also reported several other issues, like inability of test generation tools to support well-known, widely-used Java framework (e.g., Spring) and core components such as dependency injection. This highlights the importance of supporting major tools and frameworks.

RQ3: Assertions and readability of generated tests need to be improved. To be embraced by developers, test generation tools need to support the major development frameworks.

Table 3.4: Summary of fault detection over 10 executions of EvoSuite and Randoop.
■ fault is detected, □ fault is not covered, ☒ fault is covered but not detected.

Fault	EvoSuite	Randoop	Fault Classification
`LifeCalc-b1 (03 Minutes)`	■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b1 (15 Minutes)`	■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b2 (03 Minutes)`	■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒	■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b2 (15 Minutes)`	■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒	■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b3 (03 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b3 (15 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b4 (03 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	Hard
`LifeCalc-b4 (15 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	
`LifeCalc-b5 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ □	Easy
`LifeCalc-b5 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b6 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	Easy
`LifeCalc-b6 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b7 (03 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	Hard
`LifeCalc-b7 (15 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	
`LifeCalc-b8 (03 Minutes)`	■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b8 (15 Minutes)`	■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b9 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ☒ ☒	■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b9 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ☒ ☒	■ ■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b10 (03 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	Hard
`LifeCalc-b10 (15 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	
`LifeCalc-b11 (03 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b11 (15 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b12 (03 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	Hard
`LifeCalc-b12 (15 Minutes)`	□ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □	
`LifeCalc-b13 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ □ □ □	■ ■ ■ ■ ■ ■ ■ ■ ■ □	Easy
`LifeCalc-b13 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b14 (03 Minutes)`	■ ■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b14 (15 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b15 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	Easy
`LifeCalc-b15 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b16 (03 Minutes)`	■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b16 (15 Minutes)`	■ ■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b17 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ □ □	■ ■ ■ ■ ■ ■ ■ ■ ■ □	Easy
`LifeCalc-b17 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b18 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	Easy
`LifeCalc-b18 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b19 (03 Minutes)`	■ ■ ■ ■ ■ ■ □ □ □ □	■ ■ ■ ■ ■ ■ ■ ☒ ☒ ☒	Easy
`LifeCalc-b19 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ □ □	■ ■ ■ ■ ■ ■ ■ ■ ☒ ☒	
`LifeCalc-b20 (03 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Average
`LifeCalc-b20 (15 Minutes)`	■ ■ ■ ■ ■ ■ ☒ ☒ ☒ ☒	■ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b21 (03 Minutes)`	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Specification
`LifeCalc-b21 (15 Minutes)`	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b22 (03 Minutes)`	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Specification
`LifeCalc-b22 (15 Minutes)`	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	
`LifeCalc-b23 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ☒ ☒ ☒	Specification
`LifeCalc-b23 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ☒ ☒	
`LifeCalc-b24 (03 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	Specification
`LifeCalc-b24 (15 Minutes)`	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	■ ■ ■ ■ ■ ■ ■ ■ ■ ■	
`LifeCalc-b25 (03 Minutes)`	■ ■ ■ ☒ ☒ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	Specification
`LifeCalc-b25 (15 Minutes)`	■ ■ ■ ■ ■ ☒ ☒ ☒ ☒ ☒	☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒	

3.3 Lessons Learned

Throughout this section, I discuss the challenges I faced in terms of tools setup and interaction with the developers, and suggest technical improvements that can be addressed by the test generation tools.

3.3.1 Tool Set up Challenges

One of the major challenges I faced during this experiment was setting up each *LifeCalc* version with its required dependencies and ensuring its successful compilation. I had to make sure all environmental dependencies, some of which contained sensitive information, are mocked and available to *LifeCalc*.

As for experiments I used the command line versions of Randoop and EvoSuite, I had to derive the right classpaths of the system under test. Given that one application might have dependencies to many third party libraries, the manual resolving of dependencies was an error prone task. Basically I had to check the tools' logs to determine which library is still missing. Then I would add those libraries to the classpath to get a proper execution. For instance *LifeCalc-b3* was dependent on *Apache Commons Lang*, and when this library was missing on classpath, EvoSuite would silently generate empty test suites. After disabling minimization (*-Dminimize=false*) and enabling the debug (*-Dlog.level=debug*) mode in EvoSuite, I managed to identify the missing libraries and re-executed all the experiments for this particular fault.

However, in retrospective, as the target application was built with Maven, I could have used some of its EvoSuite's plugins to derive the right classpaths and properly setup all the needed dependencies. Note that EvoSuite does have a plugin for Maven, but it is not suitable for the type of experiments I ran in the first phase of this thesis. Generally, the documentation of these test data generation tools could be extended to explain how to use the command line versions on existing projects compiled with build tools like Maven, Ant and Gradle.

Insight 1: The use of unit test generation tools on the command line requires detailed understanding of the build infrastructure, and tool documentations are currently not helpful in achieving a correct setup.

3.3.2 Suggested Improvements

Based on the results of the conducted experiment, the following are potential improvements for the test generation tools:

Construction of Complex Objects

Since for most of the *Challenging Faults* the generated test cases failed to satisfy the outer condition, due to incapability in constructing and populating complex objects, one priority for tool builders should be to improve the construction and population of complex objects.

Insight 2: 100% of the challenging faults remained undetected as none of the tools were able to construct and populate objects with complex structure. More research on how to solve this problem is required.

Generating Specific Input

In most of the *Hard Faults* the faulty statements are not covered as there is a conditional statement prior to them that requires specific primitive data. There are some cases where the faulty statement was covered, but only a very specific set of primitive data would trigger the failure. For example, in the code snippet highlighted in the Listing 3.9 line #11, the faulty statement should not only be covered, but `getPaymentFrequency()` also needs to be 0 in order to throw a `java.lang.ArithmeticException`. One good example of improving value generation is implemented within EvoSuite by extracting `enum` values as a set of required input data.

Insight 3: Only 47.78% (EvoSuite) and 12.22% (Randoop) of hard faults which require specific primitive values have been detected, even if the faulty statements are executed. Covering code is not enough: further criteria to optimize should be designed to help these tools in generating this kind of input values.

Extension of Assertions

I encountered cases such as *LifeCalc-b21*, specified below, where the fault could have been detected by the generated test case with a better assertion.

```
...
for (int i = 2; i <= Months.monthsBetween(param1, param2).getMonths(); i++)
{
// Faulty Statement
list.add(i, list.get(i - 1));
// Fixed Statement
list.add(i, list.get(i - 1).plusMonths(1));
}
...
```

To detect this fault, tests need to check the content of the list. However, generated assertions tend to only consider direct observer methods of the objects in the test (e.g., `isEmpty()`, `size()`), and thus only check for the list size.

Insight 4: At least 50% (EvoSuite) and 64% (Randoop) of the specification faults could have been detected with more appropriate assertions. More research in effective assertion generation would hence be useful.

3.3.3 Developer Feedback

I demonstrated the tools and the results of this study to the *LifeCalc* developers. They were interested about the possibility of integrating automated test generation tools with continuous integration tools such as Travis and Jenkins. This is currently in its early implementation stage, as tools like EvoSuite have provided beta versions of a Jenkins plugin.

Insight 5: Developers in industry expect automated test generation tools to integrate with standard continuous integration tools. For an effective technology transfer from academic research to industrial practice, building plugins for these tools would be useful.

The other comments of the developers were related to the readability of the generated tests, and difficulties in navigating through the generated test suites.

Insight 6: Developers in industry are concerned about the readability of generated unit tests, the generated input data, and the generated assertions. These are topics that would warrant further research.

Given that test readability is a concern for developers, smaller automatically generated test suites may be more preferred to read and analyze. Table 3.5 reports the size of test suites based on the total number of generated test methods. As expected from a random testing-based tool, Randoop has generated up to approximately 53,000 test methods (*LifeCalc-b1*) while EvoSuite's test suite size did not exceed 32 test methods. EvoSuite performs a minimization to ensure that redundant tests are excluded. It would be more practical if test generation tools filter all redundant tests throughout the process, and it would be beneficial to prioritize the generated tests in a way to detect faults earlier, specially in cases where the generated test suite is huge. This way, the test execution could have stopped sooner, for cases where the fault is detected, and the test execution resources would be optimized. This point may be more important for RANDOOP that tends to generated larger test suites. Moreover, recent extensions of EVOSUITE and RANDOOP to support popular build management frameworks, such as Maven and Gradle, suggest that tool development is heading in the right direction.

3.3.4 Threats to Validity

In this experiment, I only considered two major test generation tools representing search-based and random testing approaches. However, tools based on dynamic symbolic execution or any other approaches might be more suitable in the case where there is a complex condition need to be satisfied. For each of the selected tools, EVOSUITE and RANDOOP, I have used their latest version with default settings. The tools might perform better if some of the settings are fine tuned. Since I had a limited number of known faults, in order to mitigate internal validity threat, I managed to analyze all the fault detection results to ensure that they are failed with the same reason as the manually written test cases, by going through the produced error logs. However, given the limited number of faults in the experiments, I might have some external validity threats, which I tried to mitigate by asking developers to provide the faults rather than choosing the faults ourselves.

Table 3.5: Total generated test methods for all faults per execution setup (3 and 15 minutes) for both EvoSuite and Randoop

Fault	**EvoSuite**		**Randoop**	
	3 min	15 min	3 min	15 min
LifeCalc-b1	2	3	31371	52636
LifeCalc-b2	4	4	1294	3579
LifeCalc-b3	4	4	2977	3965
LifeCalc-b4	15	15	20019	27150
LifeCalc-b5	18	32	14502	17899
LifeCalc-b6	4	4	13893	17643
LifeCalc-b7	6	6	3122	5992
LifeCalc-b8	7	14	12630	15767
LifeCalc-b9	4	4	6893	9092
LifeCalc-b10	2	2	18488	23119
LifeCalc-b11	8	9	2121	6655
LifeCalc-b12	12	12	3022	6987
LifeCalc-b13	21	22	9876	12876
LifeCalc-b14	3	3	5433	7652
LifeCalc-b15	8	10	1232	3989
LifeCalc-b16	2	2	2457	4998
LifeCalc-b17	5	7	11542	13432
LifeCalc-b18	10	17	15432	17878
LifeCalc-b19	3	3	5679	8553
LifeCalc-b20	14	14	4390	7658
LifeCalc-b21	8	14	8992	11234
LifeCalc-b22	7	10	11675	21245
LifeCalc-b23	16	24	1959	3009
LifeCalc-b24	13	19	2832	6721
LifeCalc-b25	6	11	9772	14289

The categorization of the undetected defects was manual, by taking into account the executions with certain level of subjectivity in the process. I reduced that by having multiple people with different level of expertise going through them independently to mitigate this risk. There were only five participants in the qualitative study but all of them are professional developers with certain level of familiarity with system under test. The participants didn't have prior knowledge with automated unit test generation but they were provided with a guideline on how to generate tests for

EvoSuite.

3.4 Related Work

Recently, researchers have shown an increased interest in automated software testing. There are a number of studies in which tools and techniques are being evaluated in terms of coverage. For example, Fraser and Arcuri [16] evaluated EvoSuite on 110 open-source projects. Besides decent levels of achieved coverage, they reported challenges they had due to practical limitations such as environmental dependencies, which Arcuri *et al.* [2] later addressed. This work was relevant in the experiments as I had a set of faulty classes with environmental dependencies, such as the file system. Xiao *et al.* [51] identified complex domain object creation as one of the main challenges in unit test generation, which I also encountered throughout the experiment (see discussion on "challenging faults" in Section 3.2.2).

The work of Shamshiri *et al.* [46] is perhaps the most related one to what is presented in this phase. Shamshiri *et al.* evaluated the effectiveness of automated test generation tools (e.g., Randoop and EvoSuite), but in contrast to this work they used open source projects (the Defect4J benchmark). Quantitatively, the number of faults detected on these open source software is comparable with what is reported in this phase for the industrial case study: Shamshiri *et al.* report a fault detection rate of 55.7%, which is comparable to the conducted experiment, in which the test generation tools found up to 56.4% of the real faults. In addition, this work also has an additional qualitative study.

3.5 Conclusions

In the first phase of this thesis, I performed a systematic study to determine the effectiveness of automatically generated test suites in terms of revealing real industrial faults. I used two of the most common test generation tools in academia, EvoSuite and Randoop. I evaluated them on a life insurance and pension products calculator engine developed by *SEB Life & Pension Holding AB Riga Branch*. Our experiment results demonstrate that test generation tools detected up to 56.40% (EvoSuite) and 38.00% (Randoop) of faults in all executions.

Our fault categorization shows that at least 41% of the undetected faults are on *Hard Faults* (they remained undetected due to the tests not being able to satisfy specific primitive values required by the faulty methods) and *Challenging Faults* (test generation tools were not able to detect the faults due to the incapability of constructing complex objects as input data). Increasing the search budget had minimal impact on fault detection rate in case of *Hard Faults*, but it increased the fault detection rate by at least 5.71% for *Easy Faults*.

I have investigated the challenges that need to be addressed by test generation tools in order to be adapted by practitioners by conducting a qualitative study. Based on the result and the analysis, the tools are not yet there to be used by industry but certainly they are on the right track. I analyzed the undetected faults in order to find out the areas which test generation tools can be improved and hopefully these concrete insights will lead the future research to address these challenges.

Chapter 4

Search-Based Detection of Deviation Failures

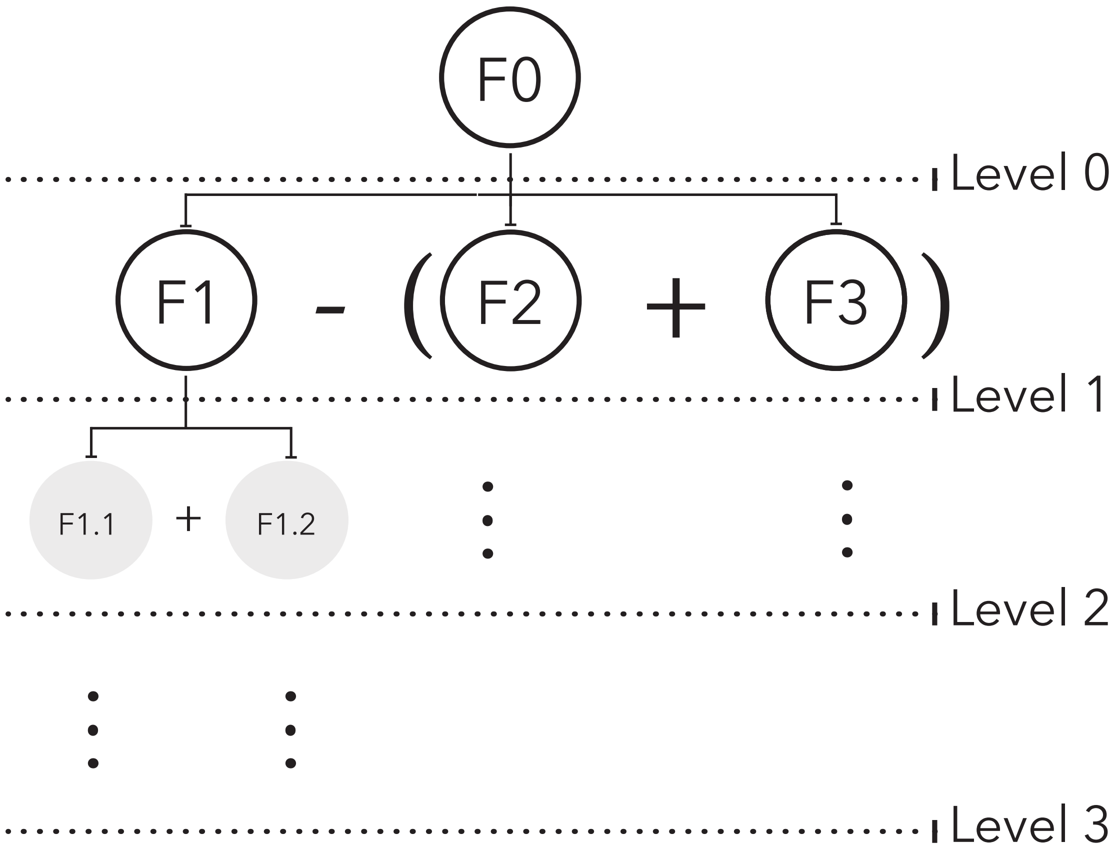
Many legacy financial applications exist as a collection of formulas implemented in spreadsheets. Migration of such code to a full-fledged system, written in a language such as Java, is an error-prone process. While small differences in the outputs of numerical calculations produced by the two artifacts are tolerable, large discrepancies could have serious financial implications. Such discrepancies are likely due to faults in the migrated implementation, manifesting themselves in what I refer to as *deviation failures*. In the second phase, I introduce a search-based technique that seeks to uncover deviation failures in migrated code automatically, by maximizing fitness functions that measure the numerical difference in the output of an original spreadsheet compared to its migration. I evaluate my proposed techniques on two financial applications, produced by *SEB Life & Pension Holding AB Riga Branch* singular, who migrated their system from a Microsoft Excel spreadsheet to a Java application. The evaluation involves 40 formulas with known and previously unknown faults in the Java code, which were accidentally introduced by developers during the migration process. While random and branch coverage-based test generation techniques were only respectively able to detect approximately 25% and 31.66% of the faults in the migrated code, the search-based approaches proposed in this phase detected up to 70% of faults with the same test generation budget. Without restriction of the search budget, up to 90% of the known deviation failures were detected. In addition, three previously unknown faults were detected by this approach which were confirmed by the *SEB Life & Pension Holding AB Riga Branch* experts.

In detail, the contributions of second phase of this thesis are as follows:

- A new search-based test generation approach, the Output-based Search Technique (OST), that aims to detect deviation failures.
- A modified fitness function that utilizes spreadsheets' nested structure of formulas, implemented into an approach called the Spreadsheet-based Search Technique (SST), to improve fault finding and their subsequent localization.
- The evaluation of the proposed OST and SST approaches using 40 formulas with real deviation failures (known and unknown), in two commercial financial applications. As part of this evaluation I compare the success rates and actual deviation values of the approaches with automatically generated test cases for branch coverage of the Java code, randomly generated test cases, as well as manual, developer-written test cases.

The second phase of this thesis is organized as follows: Section 4.1 presents a

Figure 4.1: *Invested Premium* Formula Hierarchy



motivational example. Section 4.2 describes the proposed search-based approaches. Section 4.3 describes the experiment setup and research questions and presents the results of the empirical evaluation, Section 4.4 reviews related works, and Section 4.5 concludes this phase.

4.1 Motivational Example

To provide a better understanding of the specifics of spreadsheet-based financial applications and the challenges faced during their re-implementation, I present an anonymized example. Figure 4.1 shows the hierarchy of the following legacy Excel formula to calculate "Invested Premium" in my case study (*LifeCalc*-f1), anonymized for confidentiality reasons.

$$P = MROUND(I - (A + R), 100) \tag{F0}$$

In F0, P is the Invested Premium, I is the Paid Insurance, A are the Administrative Fees, and R are the Risk Fees. The formula contains rounding to the nearest multiple of 100. Listing 4.1 presents the re-implementation of the formula.

$$I = M + S \tag{F1}$$

In this formula, F1, I is the Paid Insurance, M is the Monthly Future Contribution, and S is the Current Assets. F1 computes M as a subformlua of F0. Listing 4.2 shows its re-implementation in Java.

The next formula, F1.1, calculates the value of an asset at specific future date based on its growth interest rate over a period of time.

$$M = FV(rate, nper, pmt, pv, type) \tag{F1.1}$$

The "*future value*" (FV) represents monthly future contributions, while *rate* is the interest rate per period, *nper* is the total number of payment periods, *pmt* is

```java
public Double investedPremium(AppObj obj, Double targetAmount, String strategy){}
  premium = Util.round(paidInsurance(obj) - (adminFees(strategy) +
  riskFees(targetAmount)), 100);
  return premium;
}
```

Listing 4.1: Java Re-Implementation of Invested Premium (F0)

```java
public Double paidInsurance(AppObj obj) {
  Double investmentReturnRate =
  Double.valueOf(rateProperties("lv.return.rate")); // 0.06
  paidInsurance = futureMonthlyContrib(obj.getCurrentAmount(),
  obj.getMonthlyPayments(), investmentReturnRate,
  obj.getRetirementTime(), obj.getSavingTime(),
  obj.getFutureAssets()) + currentAssets(obj.getPensionAssets());
  return paidInsurance;
}
```

Listing 4.2: Java Re-Implementation of Paid Insurance Formula (F1)

```java
public Double futureMonthlyContrib(Double currentAmount,
Double monthlyPayments, Double investmentReturnRate, Integer
retirementTime, Integer savingTime, Double futureAssets) {
  monthlyContrib = (currenAmount + monthlyPayment) * (((Math.pow
  ((1 + (investmentReturnRate / 12)), (retirementTime -
  savingTime))) - 1)) / (investmentReturnRate / 12)(*\bfseries + futureAssets*)
  * Math.pow((1 +(investmentReturnRate / 12)), (retirementTime
  - savingTime));
  return monthlyContrib;
}
```

Listing 4.3: Java Re-Implementation of Monthly Contribution (F1.1)

the payment made in each period, *pv* is the present value, and *type* indicates due payments. Listing 4.3 presents the re-implementation of this formula in Java. In the re-implemented code, *rate* is $investmentReturnRate/12$, which is the monthly investment return rate, *nper* is $retirementTime - savingTime$, which is the remaining months till the retirements, *pmt* is $currentAmount + monthlyPayment$, which is the current contributed amount plus the upcoming monthly contribution and *pv* is *futureAssets*, which is based on the value of future assets.

The Java re-implementation of this formula contains a fault — the reference to the variable `futureAssets` (shown in bold in Listing 4.3) is is missing in the faulty Java re-implementation. This causes it to produce a different output compared to its Excel implementation. Thorough testing may reveal this problem. However, finding a specific input that reveals the failure is non-trivial. It is not enough to just execute the faulty code, since the effect of the fault may not propagate to the output and produce a failure.

For instance, Listing 4.4 presents the test case generated automatically as part of a branch-coverage optimized test suite by the EvoSuite tool [18]. This test case executes (covers) the fault, but fails to reveal a failure. The calculated *Invested Premium* in both programs is 1000, and therefore, based on the generated test data, the difference between the output of the legacy Excel formula and its Java re-implementation of *Invested Premium (P)* is 0.

Listing 4.5 shows a test case generated by my approach. This test case results in a difference in the output of the *Invested Premium (P)* in the Excel and Java programs as presented in Table 4.1 ($\Delta = |1100 - 1200| = 100$). The difference results from the the

```
@Test(timeout = 4000)
public void test1()  throws Throwable  {
  Pension pension0 = new Pension();
  AppObj appObj0 = new AppObj();
  Double double0 = new Double(30.0);
  appObj0.setCurrentAmount(double0);
  appObj0.setPensionAssets(double0);
  Double double1 = pension0.investedPremium(appObj0, 0.0, "S1");
}
```

Listing 4.4: Sample Test Case Generated by Branch Coverage

```
@Test(timeout = 4000)
public void test3()  throws Throwable  {
  Pension pension0 = new Pension();
  AppObj appObj0 = new AppObj();
  Double double0 = new Double(10.0);
  appObj0.setCurrentAmount(double0);
  Double double1 = new Double(25.0);
  appObj0.setMonthlyPayments(double1);
  Double double2 = new Double(11.0);
  appObj0.setSavingTime(double2);
  Double double3 = new Double(100.0);
  appObj0.setFutureAssets(double3);
  Double double4 = new Double(40.0);
  appObj0.setRetirementTime(double4);
  Double double5 = pension0.investedPremium(appObj0, 838.0, "S1");
}
```

Listing 4.5: Sample Test Case Generated Proposed Search Technique

faulty re-implementation of *Invested Premium (P)* in one of the sub-formulas (Monthly Contribution Formula F1.1) and is ($\Delta = |Excel - Java| = |1204 - 1258| = 54$).

In the second phase of this thesis, I aim to automatically generate test cases that

Table 4.1: Invested Premium calculation using generated tests case in Listing 4.5.

Variables	**P**	**I**	**A**	**R**	**S**	**M**
Excel	1100	1604.80	500	0	400	1204.90
Java	1200	1658.80	500	0	400	1258.80

detect such differences (which I refer to as *deviations*). I propose a generic search-based approach with two fitness function implementations. The baseline fitness function in this example would be the delta between the root formulas (*Invested Premium (P)*) in Excel and Java. The other proposed fitness functions use the delta between lower level sub-formulas. The next section, introduces the proposed approach and different fitness functions in detail.

4.2 Search-based Approach for Deviation Detection

In this section, I describe a search-based approach (with two alternative fitness functions) that aims to generate tests that maximize the deviation between an Excel formula and its Java re-implementation. Note that at each given time, I only focus on one Excel formula and its Java implementation, and the tests are generated as JUnit tests for the Java implementation.

4.2.1 Deviation failures

Deviation failures are failures where two alternative implementations produce different outputs. In financial applications, like many other domains, not all deviations are immediately considered as a failure, that is, some degree of variation in the outputs is tolerable. These tolerable variations are specified in form of threshold limits defined for each formula and its sub-formulas. Consequently, a deviation failure is deemed to have occurred when the output of two programs differ by more than a specified threshold limit:

Definition 1 (Deviation Failure). A deviation failure occurs when the numerical output o_1 from a program p_1 differs from the output o_2 for a similar program p_2 by a tolerable threshold t. That is, $|o_1 - o_2| > t$.

For ease of understanding, we normalize deviation failures in this phase, such that violations are represented by numbers greater than 1.0, regardless of the formula-specific tolerable threshold t:

Definition 2 (Normalized Deviation Failure). A normalized deviation failure occurs when the numerical output o_1 from a program p_1, and the output o_2 from a similar program p_2, divided by a tolerable threshold t, exceeds 1.0. That is, $\frac{|o_1 - o_2|}{t} > 1.0$.

4.2.2 The Output-based Search Technique (OST)

Search-based test generation techniques, in general, formulate the test objective as an optimization problem and apply meta-heuristic search algorithms, such as genetic algorithms, to find an optimal solution [38]. Typically, the fitness function drives the search to generate test cases that cover as much of the code as possible. In this study, however, I focus on the detection of deviation failures. I therefore first propose a search-based method that guides the search for test cases towards deviations between program versions that exceed defined tolerable thresholds.

I refer to this method as the Output-based Search Technique (OST). Assume the Excel formula under test (F0) is re-implemented by a method called M0, in Java. Let M0 and F0's input parameters be $x_0, \ldots, x_n$. Each test case may therefore be represented by the vector $< x_0, \ldots, x_n >$, and a fitness function can be defined as:

$$FF\_OST = |\mathrm{F0} - \mathrm{M0}|$$

To implement OST, I used the search-based test generation tool EvoSuite [18], and extended it with my own fitness function. OST's stopping criterion can be any typical criterion such as time-outs, maximum number of generations or fitness evaluations, etc.

The other search operators such as crossover, mutation, selection strategy, etc. are the same as default settings in EvoSuite (i.e., The crossover operation combines different parents (*P1* and *P2*) to generate new offspring (*O1* and *O2*) which is then mutated by adding, modifying or deleting statements. Once the reproduction is over, there will be new parents ready for selection).

It is also worth mentioning that identifying the corresponding Java method (M0) per Excel formula (F0) was done manually in this study. It can be potentially automated for instance using keyword matching, but it may become very imprecise. Therefore, I did not try to automate this step.

4.2.3 Spreadsheet-based Search Technique (SST)

OST explores the top level formula of a spreadsheet only. The Spreadsheet-based Search Technique (SST) delves into sub-formulas. Assume F0 from the previous section consists of two sub-formulas F1-1 and F1-2, which are implemented in M1-1 and M1-2, respectively. M1-1 and M1-2 are called in M0 and have input parameters as $< a_1, \ldots, a_i >$ and $< b_1, \ldots, b_j >$. Now assume the defect that results in deviation failure is in M1-2 and it will be detected only with a specific value sets for $< b_1, \ldots, b_j >$. Following OST, the search algorithm's inputs are $< x_1, \ldots, x_n >$ thus there is no specific guidance towards values for $< b_1, \ldots, b_j >$ during the search. The FF_OST fitness function is detached from the sub-formula inputs, and only looks at the final output.

Therefore, to overcome this problem I need to localize the underlying deviated sub-formula at each level, starting from root formula all the way to its leaves. Spreadsheet-based Search Technique (SST), implements this idea by starting from root-level deviations but narrowing it down to the defective method over time.

In high-level, the proposed SST algorithm is a recursive version of OST, as follows:

1. It applies an OST on a level i formula Fi-x (where i is the level starting from 0 – the root level – and x is the sequential ID for each sub-formula in the given level); See Figure 4.1.

2. The OST searches through the input space of Mi-x parameters. Mi-x is a Java method that corresponds to the Fi-x formula.

3. The search continues until a deviation between the outputs of Fi-x and Mi-x is

detected.

4. After finding a deviation, the search continues for t seconds, while still in level i, to potentially increase the already detected deviation.

5. If the algorithm manages to increase the deviation in t seconds, step 4 is repeated.

6. The algorithm stops exploring level i and moves to level $i + 1$, when the best deviation could not be improved in t seconds.

7. To move to level $j = i + 1$, the Fi-x's sub-formulas (F$j - 1$, …, Fj-y) are analyzed to find out which one is the "most contributing" sub-formula. The "most contributing" sub-formula is the one with the highest local deviation, where a local deviation is the delta between the sub-formula's output and its corresponding Java method's output, in level j.

8. The algorithm repeats steps 1-7, until it reaches a global stopping criterion (e.g., time-out).

Variations of SST

In this phase, the SST algorithm comes in three variations (configurations). These configurations vary in terms of a) the global stopping criteria and b) the time periods spent to explore the input space in each level. In the first two variations (SST1 and SST2), the global stopping criterion is $n = 5$ minutes. In SST1, the algorithm spends multiples (m) of 60 seconds on the same formula level, after the first deviation is found ($t = 60$). As discussed in the algorithm, the exact value of m depends on whether there is still a progress in the fitness value for each given level or not.

In SST2, n is still equal to 5 minutes, but the algorithm spends multiples (m) of 30 seconds on the same formula level. The main difference between SS1 and SS2 is that SST2 spends minimal time in each level, thus has more time to explore lower levels, with the same total budget. SST3 is the final variation of SST with $t = 60$ seconds, but no time limit for global stopping criterion. The SST3's global stopping criterion is "Stop when all levels of formula are explored". The objective of SST3 is to explore the maximum power of the SST approaches, regardless of the underlying cost.

In the empirical study section (Section 4.3), I will compare SST1, SST2, SST3, and OST with each other and with some baselines (random search, coverage-based testing, and manual testing).

Preparation Step

There are at least three preparation steps that are required before running any of the SST techniques, as follows:

- **Identifying hierarchy of sub-formulas per Excel formula**. I have automatically collected this information and record it in a matrix using a custom Excel macro.

- **Mapping excel formulas and Java methods**. As mentioned before, this part has been done manually, I mapped all identified Java and Excels formulas, manually. Note that the thresholds per sub-formula was already available to us as "error margins" in the applications.

- **Dealing with external dependencies**. In order to avoid external dependency issues, such as access to properties files, I have replaced all the variables retrieved from property files with the actual values, in the Java code.

4.3 Empirical Study

In order to evaluate the proposed search-based test generation techniques for detecting deviation failures, I conducted an empirical study on two real financial applications. This section describes the details and results of this study.

4.3.1 Research Questions

The objective of the study has been broken into four research questions as follows:

RQ1 *How effective is the Output-based Search Technique (OST) compared to baseline test generation at detecting deviation failures?*

The aim of this research question is to evaluate how a focused search-based approach for deviation detection compares to common existing test generation techniques at defecting defects.

RQ2 *Is spreadsheet-based search (SST) more effective than Output-based Search (OST) at detecting deviation failures?*

The aim of this research question is to evaluate the improvement of the optimized Spreadsheet-based search (SST) over the basic Output-based (OST) approach, given the same search budget.

RQ3 *What is the deviation failure finding potential of the Spreadsheet-based Search?*

The aim of this research question is to assess the overall fault finding ability of the

Spreadsheet-based Search Technique (SST), without tight constraints on the search budget.

RQ4 *Can Spreadsheet-based Search detect new and unknown deviation failures?*

The aim of this research question is to evaluate whether results on SST generalize and can detect previously unknown deviation failures.

4.3.2 Subjects of Study

The experiments are based on two real financial applications. The first application is a life insurance and pension products calculator engine known as *LifeCalc*. It is a medium-sized standalone software component with approximately 80,000 LOC, and consists of complex critical pension products calculations with many business rules. Its implementation started in early 2015, and it has been released to production in early 2016. The second application is *PensionPlanner*, a pension funds calculator engine [29] with at about 170,000 LOC. Both of these applications are developed and owned by *SEB Life & Pension Holding AB Riga Branch*.

Initially, *LifeCalc* and *PensionPlanner* were implemented using Excel sheets and have been used by the company for several years. For many internal strategic reasons, including better automation support and technology compatibility with their other products, these products been newly implemented using the Java technology stack. Throughout the implementation, the original Excel sheets have been used as the specification for the implementation of the new applications.

For RQ1–3, the company's developers provided us with 20 spreadsheet formulas from *LifeCalc*. For each of these formulas they further provided us with a deviation failure report of the Java-based reimplementation, which had been resolved in their issue tracking system. They selected these formulas and failures arbitrarily, trying to include examples from different times throughout the life cycle of the project. For each spreadsheet formula, I extracted the corresponding Java program version along side with any manual tests that had been used in order to detect and fix the particular fault.

For RQ4 (detection of unknown faults) I could not use the same formulas, since I already know that each of them contains contains a fault. Therefore, I further randomly selected 20 spreadsheet formulas from *LifeCalc* and *PensionPlanner* (10 each), without knowing whether they contain any deviation failures. I did not have access to any manually written test cases for these formulas.

Table 4.2 reports characteristics of the studied 40 formulas in terms of some Excel and Java metrics. The "Levels" (number of nested levels in the hierarchy of the formula starting from 0 for root) and "Sub-formulas" (total number of sub-formulas in all levels) are extracted from the Excel formula directly. However, the "Variables" (the total number of unique variables in corresponding Java methods for all sub-formulas in all levels) and "LOC" (the total number of lines of Java code for all sub-formula of the root) are extracted from the Java code[1]. These statistics show that the studied formulas cover a wide range of "difficulty", with "Variables" ranging from 3 to 36, and "Sub-formulas" ranging from 1 to 16.

Note that in all the cases, the deviation failure has been revealed using the manual test cases written by the developers *after* the bug had been reported by business analysts. In other words, the initial test cases written by the developers were not able

[1]http://metrics.sourceforge.net

Table 4.2: Characteristics of 40 formulas in terms of #Unique variables (V), #Nested levels in the formula hierarchy (L), #Total sub-formulas within all levels (S) and total #Lines of Java code (C).

(a) 20 known faulty formulas

Formula	**V**	**L**	**S**	**C**
***LifeCalc*-f1**	8	2	3	41
***LifeCalc*-f2**	15	3	9	165
***LifeCalc*-f3**	12	4	7	109
***LifeCalc*-f4**	6	2	3	56
***LifeCalc*-f5**	24	3	12	270
***LifeCalc*-f6**	19	6	10	307
***LifeCalc*-f7**	26	2	8	120
***LifeCalc*-f8**	21	3	11	235
***LifeCalc*-f9**	12	2	8	154
***LifeCalc*-f10**	5	1	4	89
***LifeCalc*-f11**	31	6	16	534
***LifeCalc*-f12**	18	3	9	192
***LifeCalc*-f13**	36	5	13	389
***LifeCalc*-f14**	7	1	5	104
***LifeCalc*-f15**	3	1	1	13
***LifeCalc*-f16**	10	1	2	29
***LifeCalc*-f17**	8	4	5	95
***LifeCalc*-f18**	7	2	3	45
***LifeCalc*-f19**	27	5	10	216
***LifeCalc*-f20**	15	6	9	182

(b) 20 NEW formulas

Formula	**V**	**L**	**S**	**C**
***LifeCalc*-f21**	23	4	9	233
***LifeCalc*-f22**	13	3	7	112
***LifeCalc*-f23**	11	3	5	77
***LifeCalc*-f24**	9	3	6	117
***LifeCalc*-f25**	14	4	9	203
***LifeCalc*-f26**	33	7	14	309
***LifeCalc*-f27**	25	5	13	230
***LifeCalc*-f28**	17	2	4	78
***LifeCalc*-f29**	11	4	7	145
***LifeCalc*-f30**	18	3	6	97
***PensionPlanner*-f1**	29	3	7	142
***PensionPlanner*-f2**	15	3	7	134
***PensionPlanner*-f3**	11	4	9	234
***PensionPlanner*-f4**	7	1	3	80
***PensionPlanner*-f5**	13	2	5	71
***PensionPlanner*-f6**	19	4	8	173
***PensionPlanner*-f7**	5	2	4	101
***PensionPlanner*-f8**	8	1	3	42
***PensionPlanner*-f9**	22	4	10	217
***PensionPlanner*-f10**	11	3	7	88

to detect these faults. The business analysts' manual acceptance testing is the last resort before releasing the product. Therefore, such late detection of deviation failure is expensive and risky. Hence, the goal is to provide an automated test generation approach that can detect such faults with smaller budget and earlier in the development phase.

4.3.3 Experiment Design

In this section, I discuss the design of my experiment to answer each research question.

Normalization

For each execution of a technique in my experiments, I record the maximum deviation detected in the root-level. Remember that not all deviations are considered a failure unless they are above certain tolerable threshold associated with the formula. Thus to help with readability of the tables, the deviations are normalized by dividing them by the actual thresholds.

Therefore, zero means no deviation, values in (0, 1) are tolerable deviations, and any value greater than or equal to 1.0 is considered a detected deviation failure.

My main goal is just to detect any deviation equal to or greater than 1.0. However, the higher the deviation the better, assuming that higher values of deviations may correspond to more serious effects. For example, suppose the system's threshold for an annual pension value is \$1.0, and so a \$1.5 deviation is a non-tolerable failure but may not affect the customer satisfaction by large if not detected and fixed right away. However, a deviation of \$100 will create a much bigger impact and insatisfaction among clients.

False positives

Automated test generation techniques in general may create false positives. That is, there may be cases where a generated test exceeds the tolerable threshold, but may violate domain and business related preconditions that the test generator is not aware of. For example, in *LifeCalc*-f16 the `variable` value must not be more than 100000.0, and a generated test with value 184414.16 is not acceptable based on business assumptions. Unfortunately, the only reliable way to validate the results is to ask domain experts. For all results in my experiments I therefore validated the solutions (test cases) with *SEB Life & Pension Holding AB Riga Branch*'s domain experts.

Metrics

The metrics used in the results are as follows:

- **Median Root Deviation:** The median of normalized deviations in the root-level, over 30 runs, per technique per failure.
- **Median Sub-formula Deviation:** The median of normalized deviations in a specific sub-formula, over 30 runs, per technique per failure. Note that this metric will only be used for SST techniques in RQ3.
- **Success Rate:** The ratio of detecting a deviation ≥ 1.0, over all 30 runs per formula per technique.
- **Validated Success Rate:** The ratio of detecting a "valid" deviation ≥ 1.0, over all 30 runs per formula per technique (validated by company's experts).
- **False Positives:** The ratio of "invalid" reported deviations over total reported deviations, in 30 runs per formula (validated by company's experts).

Statistical Tests

Whenever I compare deviation values directly, I not only look at the median values, but also run a non-parametric statistical significance test (U-Test) to make sure the differences between two techniques are not due to chance. I also report the median Success Rates and median Validated Success Rates, over all formulas under study.

RQ1 Methodology

This research question compares OST, my basic search-based technique for deviation detection, against two baseline test generation approaches:

- **Random testing:** Random search is used as a sanity check. If OST is not better than random testing, then there would be no need for a search-based approach, because the search problem is either very simple, or too challenging. I used EvoSuite's random test generation to do a random search with the same search budget as OST (with the minimization option disabled).
- **Branch coverage testing:** Code coverage criteria such as statement and branch coverage are quite common in industry, and they are commonly targeted by search-based test generation. I compared my approach with the coverage-based test generation implemented in EvoSuite. Since coverage based test generation is also search-based, this allows us to focus only on the different fitness functions (from maximizing code coverage to deviation values).

As all studied test generation approaches, in the second phase of this thesis, are implemented in EvoSuite, the comparisons have less confounding factors related to the implementation and optimization.

In this RQ, I use the 20 known faulty formulas of *LifeCalc* (explained above) as my targets. Each technique (R: Random Technique, BR: Branch Coverage Technique, and OST: Output-based Search Technique) was executed 30 times per faulty formula, resulting in a total of 600 executions. As each formula is implemented in a single Java class, each execution consisted of applying EvoSuite on the corresponding class with either random test generation, branch coverage optimization, or the OST fitness function. I set a global stopping criterion for all three algorithm of 5 minutes.

RQ2 Methodology

This RQ investigates the effectiveness of SST. SST1 repeats 30 seconds of search, after each maximum deviation detection, to improve in the same level (*t=30 seconds*), whereas SST2 spends 60 seconds for the same reason. The global stopping criteria for SST1, SST2, and OST are all the same (5 minutes). I have evaluated these approaches on the same sets of failures as RQ1 by executing each approach 30 times per faulty version. Since the deviations reported by SST are at the level of sub-formulas, they are not directly comparable with OST's deviations, which are in the root-level. Therefore, I use Validated Success Rates as the main comparison metric.

RQ3 Methodology

SST3 uses the same time interval as SST2 ($t = 60$ seconds) at each level, but it does not impose a global timeout (e.g., the 5 minutes used in RQ1 and RQ2). Instead, the algorithm searches until there are no sub-formulas left. Obviously, this algorithm is going to be more expensive than SST1, SST2, and OST, which is why RQ3 does not compare SST3 with these algorithms. However, in RQ3 I am interested to know the ultimate effectiveness of an SST approach.

RQ4 Methodology

For RQ4, I compared SST with the baseline of manual testing. I used 20 new spreadsheet formulas, different from those used in RQ1–3. These formulas were not given to us as known faulty formulas, but I selected them randomly from two applications *LifeCalc* and *PensionPlanner* (10 formulas each). This means that at the time of experimentation 1) I did not know which formulas are faulty, if any, 2) for the detected deviations I did not know if manual testing have detected a deviation there as well or not, and 3) I did not know if SST3 has missed any deviation. Therefore, this dataset was perfect for analyzing the effectiveness of SST3 in detecting unknown deviation failures.

4.3.4 RQ1: How effective is the Output-based Search Technique (OST) compared to baseline test generation at detecting deviation failures?

Table 4.3 summarizes the Median Root Deviations and Success Rates for R, BR, and OST for the 20 known failures of *LifeCalc*. All Median Root Deviations ≥ 1.0 are highlighted as well. Overall, in 8 out of 20 cases OST's Median Deviations are above thresholds. In contrast, BR only exceeds the threshold in 7 out of 20 cases and R only in 5 out 20. Consequently, OST has a median success rate of 50%, BR has 40%, and R only 28.34%.

Looking at the actual Median Root Deviations, in most cases, BR is better than

Table 4.3: Comparing the Median Root Deviations (D) and Median Success Rate (SR) over 30 executions per 20 formulas of *LifeCalc* using R (Random), BR (Branch Coverage) and OST (Output-based Search) Techniques. Those highlighted are detected deviations above threshold per technique. ∗ and † represents cases where the OST's deviation is significantly different than R and BR (p-value ¡ 0.05)

Technique	**R**		**BR**		**OST**	
Formula	**D**	**SR**	**D**	**SR**	**D**	**SR**
***LifeCalc*-f1**	1.18	50.00	1.25	73.33	$2.62^{*\dagger}$	70.00
***LifeCalc*-f2**	0.64	23.33	0.88	46.67	$0.99^{*\dagger}$	50.00
***LifeCalc*-f3**	0.40	0.00	0.54	0.00	$0.82^{*\dagger}$	16.67
***LifeCalc*-f4**	0.78	30.00	1.15	63.33	1.63^{*}	63.33
***LifeCalc*-f5**	0.84	36.67	1.14	60.00	0.99	50.00
***LifeCalc*-f6**	0.46	36.67	0.64	30.00	$0.96^{*\dagger}$	50.00
***LifeCalc*-f7**	0.86	40.00	0.90	46.67	$1.36^{*\dagger}$	66.67
***LifeCalc*-f8**	0.73	16.67	0.57	0.00	$0.99^{*\dagger}$	50.00
***LifeCalc*-f9**	0.42	0.00	0.43	0.00	$0.57^{*\dagger}$	16.67
***LifeCalc*-f10**	3.24	86.67	4.26	100.00	$5.74^{*\dagger}$	100.00
***LifeCalc*-f11**	0.61	26.67	0.98	50.00	0.97^{*}	46.67
***LifeCalc*-f12**	0.65	3.33	0.55	0.00	$0.88^{*\dagger}$	16.67
***LifeCalc*-f13**	0.69	23.33	0.68	26.67	0.84	33.33
***LifeCalc*-f14**	1.06	50.00	1.09	60.00	1.50	70.00
***LifeCalc*-f15**	2.01	76.67	1.38	63.33	$2.30^{\dagger}$	86.67
***LifeCalc*-f16**	1.97	80.00	1.68	66.67	2.11	70.00
***LifeCalc*-f17**	0.55	0.00	0.60	0.00	$0.79^{*\dagger}$	20.00
***LifeCalc*-f18**	0.94	40.00	0.82	33.33	0.88	43.33
***LifeCalc*-f19**	0.61	0.00	0.70	0.00	$0.96^{*\dagger}$	46.67
***LifeCalc*-f20**	0.61	3.33	0.66	0.00	$1.03^{*\dagger}$	50.00
Median (%)	0.96	28.34	1.05	40.00	1.45	50.00

R (14 out of 20 cases) and OST is better than both (18 out of 20). Since the raw comparisons of medians may be misleading, I also ran a statistical significance U-Test, which shows that OST's results are significantly different (with higher medians) in 14 cases out of 20 compared to Random and 13 cases out of 20 compared to Branch.

Though having higher actual deviation values is important, these deviations may include false positives. Therefore, Table 4.4 shows the False Positives both in terms of number of invalid reported deviations per formula and the median percentages of invalid reported cases over all reported deviations, over all formulas. The effect of False Positives on the Success Rates can be seen as the Validated Success Rates. Looking at R, BR, and OST, in Table 4.4, I can see that OST's validated success rate is 46.67%, whereas, R and BR are far behind at 25% and 31.66%. Another point to note is that the False Positive percentage for OST is not very high (11.67%). Therefore, the developers' time that is spent to inspect OST's reported deviations are in most case well paid off. It is also worth noting that the number of False Positives produced by OST is in the same range as random search and branch coverage techniques.

Overall, the results suggest that neither test generation based on random search nor on branch coverage are sufficient to detect deviation failures. However, the basic search-based approach seems interesting and may have some potential. The results conform with the motivational example given in Section 4.1 and confirm that a more focused search-based approach may be performing even better, in terms of detecting deviation failures.

OST outperforms Random search and Branch coverage (46.67% vs. 25% vs. 31.66% Validated Success Rates).

Table 4.4: False Positives (FP) and Median Validated Success Rates (SR) for Random (R), Branch Coverage (BR) and Output-based (OST) Techniques for 20 known faults in the *LifeCalc*. The undetected deviations are represented by (-).

Formula	**R (%)**	**BR (%)**	**OST (%)**
***LifeCalc*-f1**	40.00	22.73	19.05
***LifeCalc*-f2**	14.29	21.43	6.67
***LifeCalc*-f3**	-	-	20.00
***LifeCalc*-f4**	22.22	31.58	15.79
***LifeCalc*-f5**	18.18	11.11	20.00
***LifeCalc*-f6**	27.27	22.23	0.00
***LifeCalc*-f7**	33.33	14.29	10.00
***LifeCalc*-f8**	40.00	-	0.00
***LifeCalc*-f9**	-	-	0.00
***LifeCalc*-f10**	46.15	40.00	33.33
***LifeCalc*-f11**	0.00	26.67	0.00
***LifeCalc*-f12**	0.00	0.00	0.00
***LifeCalc*-f13**	14.29	25.00	10.00
***LifeCalc*-f14**	6.67	27.78	19.05
***LifeCalc*-f15**	43.48	10.53	19.23
***LifeCalc*-f16**	25.00	10.00	14.29
***LifeCalc*-f17**	-	-	0.00
***LifeCalc*-f18**	25.00	20.00	23.08
***LifeCalc*-f19**	-	-	7.14
***LifeCalc*-f20**	0.00	-	13.33
Median FP (%)	16.23	12.69	11.67
Median Validated SR (%)	25.00	31.66	46.67

4.3.5 RQ2: Is Spreadsheet-based Search (SST) more effective than Output-based Search (OST) at detecting deviation failures?

As described in Section 4.2.3, the Spreadsheet-based Technique (SST) comes with different variations, based on how much time is spent on different sub-formulas. In order to allow for a fair comparison to OST, in this RQ I consider SST1 and SST2, which use the same global stopping criterion.

Table 4.5 shows the False Positives and the Validated Success Rates for SST1 and SST2, for the same 20 formulas of RQ1. Comparing Validated Success Rates of SST1 (70%) and SST2 (66.7%) with OST's (46.67%) from Table 4.4 I can already observe the amount of improvement a spreadsheet-based approach can provide over OST (extra 23.33%). The False Positives of SST1 and SST2 are also in a reasonable range (11.25% and 12.66% respectively).

The idea behind having these two configurations for SST was to evaluate the algorithm with a setting that lets more exploration (SST1) vs. more exploitation (SST2). Although I obviously see some differences between the results, the comparison remains inconclusive with respect to SST1 and SST2.

To better compare SST techniques with OST, I look at the root level deviations of OST and SSTs. This requires manual propagation of SSTs to root-level which is quite labour intensive. In my case 50 hours of manual work was required to apply it on 600 solutions of SST1 (20 formulas each 30 runs). Therefore, given the cost of this process and the fact that the differences between SST1 and SST2' results are insignificant, I only apply this on SST1 (note that the manual propagation is not part of the SST algorithm. I apply that just to calculate the extra evaluation metric).

Table 4.6 summarizes the Median Root Deviations for SST1 and OST over 30 runs, per faulty version. According to the table, OST's Median Root Deviations are always less than SST1's. Checking the statistical significance test results show that in fact in 4 out of 20 cases (*LifeCalc*-f1, f10, f15, and f16) the differences are not significant. Therefore, overall, in 16 out 20 cases SST approaches outperforms OST. Note that SST1 not only finds higher root-level deviations and detects more failures, but it also

Table 4.5: False Positives (FP) and Median Validated Success Rate (SR) using Spreadsheet-based Techniques, SST1 and SST2, for 20 known faults in the *LifeCalc*.

Formula	**SST1 (%)**	**SST2 (%)**
***LifeCalc*-f1**	17.39	23.93
***LifeCalc*-f2**	4.76	25.00
***LifeCalc*-f3**	4.55	16.67
***LifeCalc*-f4**	17.39	15.38
***LifeCalc*-f5**	17.86	19.23
***LifeCalc*-f6**	10.00	14.29
***LifeCalc*-f7**	16.67	13.33
***LifeCalc*-f8**	14.81	0.00
***LifeCalc*-f9**	0.00	0.00
***LifeCalc*-f10**	26.67	26.67
***LifeCalc*-f11**	3.85	14.29
***LifeCalc*-f12**	0.00	9.09
***LifeCalc*-f13**	18.52	0.00
***LifeCalc*-f14**	5.00	18.52
***LifeCalc*-f15**	8.70	12.00
***LifeCalc*-f16**	12.5	12.00
***LifeCalc*-f17**	6.25	11.11
***LifeCalc*-f18**	16.67	0.00
***LifeCalc*-f19**	8.00	7.41
***LifeCalc*-f20**	22.22	8.00
Median FP (%)	11.25	12.66
Median Validated SR (%)	70.00	66.70

localizes the fault by identifying an exact low-level sub-formula with high deviation. The more test budget, the better localization one can achieve. I explore this in more details in RQ3.

Spreadsheet-based Search performs better than Output-based Search (70% vs. 46.67% Validated Success Rates).

Table 4.6: Median root deviations for SST1 and OST over 30 runs per failure for 20 known failures in *LifeCalc*. Those highlighted are detected deviations above threshold. SST1 results are significantly different than OST's (p-value ¡ 0.05).

Formula	**OST**	**SST1**
***LifeCalc*-f1**	2.62	1.53
***LifeCalc*-f2**	0.99	1.72
***LifeCalc*-f3**	0.82	1.10
***LifeCalc*-f4**	1.63	1.87
***LifeCalc*-f5**	0.99	1.82
***LifeCalc*-f6**	0.96	1.50
***LifeCalc*-f7**	1.36	1.88
***LifeCalc*-f8**	0.99	1.25
***LifeCalc*-f9**	0.57	0.83
***LifeCalc*-f10**	5.74	3.88
***LifeCalc*-f11**	0.97	1.39
***LifeCalc*-f12**	0.88	1.83
***LifeCalc*-f13**	0.84	1.13
***LifeCalc*-f14**	1.5	1.75
***LifeCalc*-f15**	2.30	1.80
***LifeCalc*-f16**	2.11	1.53
***LifeCalc*-f17**	0.79	1.37
***LifeCalc*-f18**	0.88	1.96
***LifeCalc*-f19**	0.96	1.33
***LifeCalc*-f20**	1.03	1.81
Median Validated SR (%)	46.67	70.00

4.3.6 RQ3: What is the deviation failure finding potential of the Spreadsheet-based Search (SST3)?

Since RQ2 considered SST with limited search budget, there remains the question whether more faults could have been found with higher search budget, or if the experiments have already shown the full potential of SST. Table 4.7 summarizes the "Validated Success Rates" for SST1 and SST3. I compare SST3 with the baseline of SST1, which was shown as the better of the two SST variants evaluated in RQ2. As the table shows SST3 outperforms SST1 by 20% (70% vs. 90%), over the 20 formulas under study from *LifeCalc* (same setup as RQ1 and RQ2).

This substantial improvement of SST3 over SST1 is due to the increased search budget. Table 4.8 summarizes the average cost of SST3, over 30 runs, for each of the 20 formulas. The costs are presented as minutes spent on average to finish one run of SST3. They range from 5 (e.g., *LifeCalc*-f15) to 27 minutes (*LifeCalc*-f11). Since different SSTs may go into different levels and find deviations in different sub-formulas, direct comparison of deviation values is not possible. However, it is interesting to see whether SST3 has managed to localize the faults better than SST1 or not. I can measure this by counting the cases where SST3 identifies a faulty sub-formula in a lower deeper level than SST1.

Table 4.7 also lists the "Median Sub-formula Deviations" for SST1 and SST3 techniques together with their identified methods ID and level. As I can see in 5 out of 20 cases SST3 has better localized the fault (deeper level faulty method is identified). In the other 15 cases out of 20, the two techniques identified the same method as faulty but SST3 have detected a higher deviation (all p-values are also less than 0.05). Therefore, SST3 is definitely more effective than SST1, both in terms of detecting deviation failures and higher normalized deviations.

Given enough time, SST can detected up to 90% of deviation failures.

Although the maximum (27 minutes) is almost 5.5 times that of the SST1 and OST budget, it still looks quite reasonable compared to the time that is needed for manual testing: The faults that were identified as "detected" by manual testing, were only reported when in a separate sprint(s) business analysts went through a thorough

Table 4.7: Sub-formula Deviation (D), Method ID (M) and Level (L) of detected failures using SST1 and SST3 in the 20 known faults in *LifeCalc*.

Technique	**SST1**			**SST3**		
Formula	**L**	**M**	**D**	**L**	**M**	**D**
***LifeCalc*-f1**	2	1	1.81	2	1	3.28
***LifeCalc*-f2**	2	2	1.25	2	2	2.40
***LifeCalc*-f3**	3	3	1.26	4	4	2.53
***LifeCalc*-f4**	2	5	1.61	2	5	3.83
***LifeCalc*-f5**	2	6	2.04	2	6	4.24
***LifeCalc*-f6**	4	7	1.95	4	7	6.86
***LifeCalc*-f7**	2	9	1.82	2	9	4.70
***LifeCalc*-f8**	2	10	1.97	2	10	2.36
***LifeCalc*-f9**	2	11	0.94	2	11	1.47
***LifeCalc*-f10**	1	12	4.68	1	12	10.67
***LifeCalc*-f11**	4	13	1.45	6	15	2.64
***LifeCalc*-f12**	2	16	1.16	2	16	1.91
***LifeCalc*-f13**	4	17	1.44	5	19	3.05
***LifeCalc*-f14**	1	20	1.15	1	20	2.93
***LifeCalc*-f15**	1	21	1.53	1	21	7.37
***LifeCalc*-f16**	1	22	1.22	1	22	6.64
***LifeCalc*-f17**	2	23	1.17	3	24	2.68
***LifeCalc*-f18**	2	25	1.80	2	25	2.92
***LifeCalc*-f19**	4	26	1.54	5	28	3.02
***LifeCalc*-f20**	4	29	2.10	6	31	3.81
Median Validated SR (%)	70.00			90.00		

and expensive set of manual acceptance testing. The analysts would use the domain knowledge and datasets that typical developers won't know about. Hence, in general, creating a failure deviation detecting test case by a developer required a round of unit level testing by the developer followed by at least one round of manual acceptance testing by the analysts to report the symptoms of the bug, which finally would result in a unit test case by developers to be added into regression test suite. It also worth mentioning that the required budget in SST3 varies across formulas and is highly correlated with the size and complexity metrics reported in Table 4.2.

Table 4.8: Average execution time used (minutes) over 30 executions for each failure, using SST3.

Formula	Avg. Time	Formula	Avg. Time
***LifeCalc*-f1**	8	***LifeCalc*-f11**	27
***LifeCalc*-f2**	12	***LifeCalc*-f12**	15
***LifeCalc*-f3**	14	***LifeCalc*-f13**	20
***LifeCalc*-f4**	8	***LifeCalc*-f14**	7
***LifeCalc*-f5**	22	***LifeCalc*-f15**	5
***LifeCalc*-f6**	23	***LifeCalc*-f16**	5
***LifeCalc*-f7**	15	***LifeCalc*-f17**	11
***LifeCalc*-f8**	16	***LifeCalc*-f18**	7
***LifeCalc*-f9**	13	***LifeCalc*-f19**	17
***LifeCalc*-f10**	6	***LifeCalc*-f20**	19

4.3.7 RQ4: Can SST3 detect new and unknown deviation failures?

In total, deviation failures were found for 7 out of the 20 new formulas at least once. Among the 7 faulty formulas, three of *PensionPlanner* had not been detected by Manual testing but were detected by SST3. After validating the results, the *SEB Life & Pension Holding AB Riga Branch* confirmed that these faults are new and are due to their limited manual acceptance testing for the new product. Four faults were detected in *LifeCalc*, and these were confirmed as known faults. Note that *LifeCalc* is an older application with a lot of time already spent on manual acceptance testing by the business analysts. No known fault were missed by SST3.

Table 4.9 presents the percentage of false positives for the new 20 formulas of the two applications under study (*LifeCalc* and *PensionPlanner*), for SST3. It also shows which formulas were identified as faulty by manual testing. Note that the information of which formulas contain known faults was not available at the time of test generation;

I obtained this information *after* test generation, when validating the tests with the domain experts.

To summarize, Table 4.9 also shows the “Validated SR” for SST3 (86.67%) and Manual testing (57.14%) in the second 20 formulas, which were randomly selected. However, manual testing misses all three deviations that SST3 detected in *Pension-Planner*, which indicates that more manual acceptance testing is required for that application.

It also worth reminding that all failure detecting manual unit test cases were written after the faults were reported by the analysts. This means that the original manual *unit* test cases’ “Validated Success Rate” was zero. It also means that getting the reported success rates by manual testing requires the business analysts involvement in manual acceptance testing and thus is very expensive, compared to the automated SST3’s approach which can be integrated with developers unit testing framework.

SST3 detected 86.67% of deviation failures (including three unknown failures), whereas the expensive manual testing detected 57.14%, in the second set of 20 formulas.

4.3.8 Threats to Validity

The main threat to the validity of this study is the generalizability issues. Given that the evaluation is done based on a case study in one company, I can not generalize the results to other applications. However, my case study is based on two real-world industry applications with real deviation faults, which might be considered as representative. In addition, even if the applications themselves might be representative, the selection of the first 20 formulas by the company, might have been biased. However,

Table 4.9: False Positives (FP) and Sub-formula Deviation (D) using SST3 for 20 NEW formulas. (- / ✓/ ✗) represents no failure/detected/not detected using Manual techniques.

Technique	**SST3**		**Manual**
Formula	**FP**	**D**	**Detected?**
***LifeCalc*-f21**	-	-	-
***LifeCalc*-f22**	6.66	93.34	✓
***LifeCalc*-f23**	-	-	-
***LifeCalc*-f24**	13.33	86.64	✓
***LifeCalc*-f25**	6.66	93.34	✓
***LifeCalc*-f26**	-	-	-
***LifeCalc*-f27**	23.33	76.64	✓
***LifeCalc*-f28**	-	-	-
***LifeCalc*-f29**	-	-	-
***LifeCalc*-f30**	-	-	-
***PensionPlanner*-f1**	16.66	83.34	✗
***PensionPlanner*-f2**	20.00	80.00	✗
***PensionPlanner*-f3**	-	-	-
***PensionPlanner*-f4**	-	-	-
***PensionPlanner*-f5**	-	-	-
***PensionPlanner*-f6**	-	-	-
***PensionPlanner*-f7**	-	-	-
***PensionPlanner*-f8**	-	-	-
***PensionPlanner*-f9**	10.00	90.00	✗
***PensionPlanner*-f10**	-	-	-
Median Validated SR (%)	86.67		57.14

random selection of the second 20 formulas reduced that threat. In addition, I believe that the approach is quite generic for spreadsheet application migration and thus I encourage replication of this study.

In terms of conclusion validity, I have conducted each experiment 30 times and reported medians and statistical significance results. Ideally I would like to rerun the experiment with more executions to gain more confidence in the results.

I keep internal validity threats as low as possible, by using a common framework

(EvoSuite) to implement all techniques. However, evolutionary algorithms for my objective may be required and there might have been errors in the tools and scripts.

Finally, in terms of construct validity, I have reduced the threat by defining very basic and clear measures for deviations and success rates . I also validated the solutions by the company experts. However, the domain experts may have given incorrect answers.

4.4 Related Work

Differential testing [36] such as regression testing and N-version testing aims to demonstrate the behavioral differences between two versions of a program executed with the same test inputs. Evans and Savoia [13] presented differential testing with the intention of detecting more changes as compared to regression testing alone. It generates tests for both original and alternative systems and compare both versions with these two test suites. Furthermore, Tao Xie *et al.* [53] extended differential testing for object-oriented programs. They proposed a framework called *Diffut* in which it simultaneously executes methods of two versions of the program with the same inputs and compare their outputs.

Another related work in the context of automated regression testing is BERT (BEhavioral Regresstion Testing) [31]. BERT tries to provide more insight to the developers as compared to the traditional regression tests by focusing on subset of code and identifying the behavioral differences between two versions of a program. *DiffGen* is another approach presented by Taneja and Xie [47] which tries to reveal the behavioral differences of two version of Java programs by instrumenting the code and adding new branches to expose the differences between two version of class if these branches are covered by test generation tools.

McMinn [39] has introduced a novel pseudo-oracle by utilizing testability transformation. Testability transformation is source-to-source program transformation to make the program under test more testable [24, 25]. Basically, the original program, which has no test oracle, will be automatically transformed into another program with the same functionality. Then the test cases aim at fining differences in the outputs of the two programs. In a recent work, Matthew Patrick *et al.* [43] utilized pseudo-oracles in their new search-based technique for testing various implementations of stochastic models with the intention of maximizing the differences between the original implementation and its respective pseudo-oracle. They have used Kolmogorov-Smirnov tests to compare the distributions of outputs from each implementation and concluded that their technique reduces the testing effort and also enables discrepancies, where they could have been overlooked.

4.5 Conclusions

Detecting deviation failures in financial applications is difficult because of the large input domain of financial formula's outputs that need to be explored. I have introduced a new search-based approach to address this problem by generating tests to maximize the discrepancies between the newly implemented program (Java implementation) and its legacy version (Excel spreadsheets). My approach explores not only the final outputs of formulas but also the respective sub-formulas. The exploration time of each sub-formula level and the global stopping criterion can be tuned (t seconds).

I have evaluated my proposed techniques on two complex pension product calculators, *LifeCalc* and *PensionPlanner*, using real financial deviation failures. The new proposed approach outperforms random and branch coverage test generation approaches. Furthermore, I have compared both of my proposed approaches, multi-level sub-formulas search-based approach with Output-based Search approach, in which spreadsheet-based approach produced up to 23.33% better detection rate as well as better fault localization. Finally, I showed that my results is quite cost-effective in practice, given the higher validated success rates (86.67%) compared to the expensive manual testing (57.14%).

Chapter 5

Conclusions and Future Works

This thesis consists of two major phases; in the first phase I performed a detailed empirical study to evaluate the effectiveness and the applicability of automated unit test generation tools and techniques in an industrial setup. The results of fault analysis demonstrate that current techniques have difficulties with detecting faults that depends on generation of specific primitive values or construction of complex object structure. I have provided several concrete insights, such as incapability of tools in generating proper assertions or the need to integrate with well-known development frameworks, which will hopefully lead the future research in this area. A future work in this direction is evaluating other test generation techniques, for example symbolic testing.

In the second phase of this thesis, I addressed specific problem within financial applications domain. Detecting numerical discrepancies are difficult and in this thesis, I proposed two novel search-based deviation detection approaches that seek to uncover deviation failures by maximizing the differences in the output of the original program and its alternative implementation. I evaluated the proposed techniques using real faults within a complex financial application. The results indicates that the new techniques outperforms current search-based and random techniques. One future work in this direction, is to automatically map the (sub)formulas to Java methods, to fully automated the proposed test generation process. Another direction, is to investigate other methods of designing an SST e.g., by choosing sub-formula differently. I would also like to more thoroughly evaluate the fault localization aspect of SST compared to other localization techniques. Finally, a future direction that applies on both phases is conducting the evaluations on a larger scale, involving multiple applications from multiple financial corporations.

Appendix A

Qualitative Study Package

A.1 Main Study

1. How can the generated tests be improved?
2. Describe what you like better about manually written tests than generated tests?
3. Would you keep the given set of generated unit tests?
4. Given current infrastructure setup, how would you like to have automated unit test generation framework to be integrated in?
5. What are the major barriers from your point of view in adapting automatic test generation tools?

A.2 Tasks

Tasks (please follow the steps)

1. Open the qualitative study package including 3 test suites, 3 bug reports, 3 code snapshots, and the questionnaire.

2. Extract the given unit tests and follow the basic given Standalone EvoSuite guidelines to setup and execute them.

   a. Answer Questionnaire SECTION A.1: Q1

3. Read the bug report and re-execute the unit tests again and try to debug and locate the bug using the given set of unit tests. (There are both failing and passing tests in the given set of test suites).

   a. Answer Questionnaire SECTION A.1: Q2

4. Try to write a manual unit tests that covers the same code as the given generated tests.

   a. Answer Questionnaire SECTION A.1: Q3 and Q4

5. Try to identify the advantages and the pitfalls of given tests from your point of view.

   a. Answer Questionnaire SECTION A.1: Q 5 and Q 6

6. If you still have time, run this extra task:

   a. Read the instructions for test generation in the guideline.

   b. Use EvoSuite to generate test cases for the given class.

   c. What is your overall impression on using the tool?

Bibliography

[1] Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *International Conference on Software Engineering (ICSE)*, pages 263–272. ACM/IEEE, May, 2017.

[2] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated Unit Test Generation for Classes with Environment Dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 79–90. ACM, October, 2014.

[3] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random Testing: Theoretical Results and Practical Implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[4] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. Testful: an Evolutionary Test Approach for Java. In *Software Testing, Verification and Validation (ICST), 2010 third international conference on*, pages 185–194. IEEE, April, 2010.

[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo.

The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, August, 2015.

[7] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[8] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[9] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[10] Lajos Cseppento and Zoltán Micskei. Evaluating Symbolic Execution-based Test Tools. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, April, 2015.

[11] Martin D Davis and Elaine J Weyuker. Pseudo-Oracles for Non-testable Programs. In *Proceedings of the ACM'81 Conference*, pages 254–257. ACM, 1981.

[12] Joe W Duran and Simeon C Ntafos. An Evaluation of Random Testing. *Software Engineering, IEEE Transactions on*, 4:438–444, 1984.

[13] Robert B Evans and Alberto Savoia. Differential Testing: A New Approach to Change Detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM, 2007.

[14] Gordon Fraser and Andrea Arcuri. EvoSuite at the Second Unit Testing Tool Competition. In *Future Internet Testing*, pages 95–100. Springer, 2013.

[15] Gordon Fraser and Andrea Arcuri. Whole Test Suite Generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, 2013.

[16] Gordon Fraser and Andrea Arcuri. A Large-scale Evaluation of Automated Unit Test Generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[17] Gordon Fraser and Andrea Arcuri. 1600 Faults in 100 Projects: Automatically Finding Faults while Achieving High Coverage with EvoSuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[18] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419. ACM, September, 2011.

[19] Gordon Fraser and Andreas Zeller. Mutation-driven Generation of Unit Tests and Oracles. *Software Engineering, IEEE Trans. on*, 38(2):278–292, 2012.

[20] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving Search-based Test Suite Generation with Dynamic Symbolic Execution. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 360–369. IEEE, 2013.

[21] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Extending a Search-based Test Generator with Adaptive Dynamic symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 421–424. ACM, 2014.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[23] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82. ACM, July, 2014.

[24] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[25] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving Evolutionary Testing by Flag Removal. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1359–1366. Morgan Kaufmann Publishers Inc., 2002.

[26] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, Open Problems and

Challenges for Search Based Software Testing. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–12. IEEE, 2015.

[27] Hadi Hemmati. How Effective Are Code Coverage Criteria? In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 151–156. IEEE, August, 2015.

[28] Peifeng Hu, Zhenyu Zhang, WK Chan, and TH Tse. An Empirical Comparison between Direct and Indirect Test Result Checking Approaches. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, pages 6–13. ACM, 2006.

[29] Investopedia. Pension pillars, 2017. 2017-08-11.

[30] Jenkins. Jenkins. `https://jenkins.io`, 2017. Accessed: 2017-02-04.

[31] Wei Jin, Alessandro Orso, and Tao Xie. Automated Behavioral Regression Testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 137–146. IEEE, 2010.

[32] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The State of the Art in End-user Software Engineering. *ACM Computing Surveys (CSUR)*, 43(3):21, 2011.

[33] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. Testing Embedded Software by Metamorphic Testing: A Wireless Metering System Case Study. In *Local*

Computer Networks (LCN), 2011 IEEE 36th Conference on, pages 291–294. IEEE, 2011.

[34] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. How Effectively does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[35] Apache Maven. Apache maven. `https://maven.apache.org`, 2017. Accessed: 2017-02-04.

[36] William M McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[37] Phil McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[38] Phil McMinn. Search-based Software Testing: Past, Present and Future. In *Software testing, verification and validation workshops (ICSTW), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[39] Phil McMinn. Search-based Failure Discovery using Testability Transformations to Generate Pseudo-oracles. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1689–1696. ACM, July, 2009.

[40] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic System Testing of Programs without Test Oracles. In *Proceedings of the eighteenth international Symposium on Software Testing and Analysis*, pages 189–200. ACM, July, 2009.

[41] Carlos Pacheco and Michael D Ernst. Randoop: Deedback-directed Random Testing for Java. In *22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, pages 815–816, October, 2007.

[42] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 75–84. IEEE, May, 2007.

[43] Matthew Patrick, Andrew P Craig, Nik J Cunniffe, Matthew Parry, and Christopher A Gilligan. Testing Stochastic Software using Pseudo-Oracles. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 235–246. ACM, July, 2016.

[44] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Search-Based Software Engineering*, pages 93–108. Springer, 2015.

[45] Sohon Roy, Felienne Hermans, and Arie van Deursen. Spreadsheet Testing in Practice. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 338–348. IEEE, 2017.

[46] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Automated Software Engineering, 30th IEEE/ACM International Conference on*, pages 201–211, November, 2015.

[47] Kunal Taneja and Tao Xie. DiffGen: Automated Regression Unit-test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410. IEEE Computer Society, 2008.

[48] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Zhendong Su. Synthesizing Method Sequences for High-coverage Testing. In *ACM SIGPLAN Notices*, volume 46, pages 189–206. ACM, 2011.

[49] Nikolai Tillmann and Jonathan De Halleux. Pex–White Box Test Generation for. NET. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

[50] Elaine J Weyuker. On Testing Non-testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[51] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Precise Identification of Problems for Structural Test Generation. In *Proceedings of 33rd International Conference on Software Engineering*, pages 611–620, May, 2011.

[52] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A Framework for Generating Object-oriented Unit Tests using Symbolic Execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

[53] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a Framework for Differential Unit Testing of Object-oriented Programs. In *Proceedings of the*

Second International Workshop on Automation of Software Test, page 5. IEEE Computer Society, 2007.

[54] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided Path Exploration in Dynamic Symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.

[55] Zhenyu Zhang, W. K. Chan, T. H. Tse, and Peifeng Hu. Experimental Study to Compare the Use of Metamorphic Testing and Assertion Checking. *JOURNAL OF SOFTWARE*, 20(10):2637–2654, 2009.

[56] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. Metamorphic Testing and its Applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 346–351, 2004.

[57] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Transactions on Software Engineering*, 42(3):264–284, 2016.