FOMENTANDO TESTES DE MUTAÇÃO VIA EXECUÇÃO CIENTE DE VARIABILIDADE

JOÃO PAULO DE FREITAS DINIZ

FOMENTANDO TESTES DE MUTAÇÃO VIA EXECUÇÃO CIENTE DE VARIABILIDADE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Eduardo Figueiredo
Coorientador: Christian Kästner

Belo Horizonte
Outubro de 2018

JOÃO PAULO DE FREITAS DINIZ

FOSTERING MUTATION TESTING WITH VARIABILITY-AWARE EXECUTION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Figueiredo
Co-Advisor: Christian Kästner

Belo Horizonte
October 2018

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Diniz, João Paulo de Freitas

D585f Fostering Mutation Testing with Variability-Aware Execution / João Paulo de Freitas Diniz — Belo Horizonte, 2018.
xxiv, 62 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais — Departamento de Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo
Coorientador: Christian Kästner

1. Computação — Teses. 2. Engenharia de software. 3. Software — testes. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Fostering Mutation Testing with Variability-Aware Execution

JOÃO PAULO DE FREITAS DINIZ

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Eduardo Figueiredo

PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. CHRISTIAN KASTNER - Coorientador
Institute for Software Research - Carnegie Mellon University

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

PROFA. SILVIA REGINA VERGILIO
Departamento de Informática - UFPR

Belo Horizonte, 1 de outubro de 2018.

Dedico esta dissertação à minha esposa, ao meu pai, à minha mãe e à minha irmã pelo constante incentivo e por sempre acreditarem em mim, e ao meu lindo filho por tornar minha vida tão especial, por me motivar e me inspirar com seu carinho.

Agradecimentos

Agradeço primeiramente a Deus e à minha amada família. Aos meus pais, pela base familiar, bons exemplos, ensinamentos, suporte, força, incentivo e presença em minha vida, e por sempre terem feito de tudo para eu ter chegado até aqui. À minha irmã, pelo apoio incondicional, pelo eterno companheirismo e por acreditar mais em mim do que eu mesmo! Esta conquista eu divido com minha amada esposa, pois só foi possível cuidar de sua gravidez, do nascimento e da criação do nosso filho, do seu trabalho, do meu mestrado, do nosso lar e do nosso relacionamento com muita dedicação nossa, principalmente por parte dela. Ao meu lindo filho, que é minha inspiração, minha motivação e faz crescer cada vez mais minha vontade de fazer tudo dar certo! À minha sogra, que foi presença marcante nos momentos críticos desta caminhada, em que eu não pude ser tão presente como gostaria.

Agradeço também aos meus amigos mais próximos, aos distantes, aos antigos e aos que fiz recentemente, por sempre terem me apoiado e me encorajado desde o momento que tomaram conhecimento desta mudança radical em minha vida, ao entrar para o mestrado com dedicação exclusiva. Especialmente ao Fabrício, que me convidou para participar de um relevante projeto no Departamento de Ciência da Computação da UFMG juntamente com seus orientandos, o que fez aumentar muito minhas autoestima e confiança, até que criei coragem para retomar a carreira acadêmica. Também à Kattiana, pelas várias conversas sobre pesquisa em Engenharia de Software e por ter passado boas referências sobre mim ao nosso orientador.

Ao professor e orientador Eduardo Figueiredo, primeiramente o agradeço por ter aberto as portas do LabSoft para mim antes mesmo do período para a inscrição ao mestrado se iniciar. Desta forma, pude conhecer os trabalhos, participar de seminários e até mesmo de um projeto em andamento do laboratório, além de conhecer os futuros companheiros de caminhada na pós-graduação. Com toda minha admiração e todo meu respeito, o agradeço pela orientação que considero excelente, pela dedicação, pelos ensinamentos, e por me fazer sentir de fato "gerenciado" pela primeira vez em minha vida.

I am very grateful to Christian, Jens, Chu-Pan, and Serena from Carnegie Mellon University. Especially to Christian Kästner, my co-advisor. With him and his students, I learned much more than I could expect regarding research in Software Engineering, in many of its related and sub-areas. Besides that, I could practice English, which I have not done for more than a decade.

Aos amigos do LabSoft, não pode faltar meu agradecimento por todo companheirismo, além dos ensinamentos, conversas e parceria. Principalmente aos que juntos cursamos disciplinas, fizemos trabalhos, projetos, pesquisas, viagens e confraternizações. Foi um constante aprendizado, sempre me deixando motivado para não "ficar para trás" em relação aos seus progressos.

Agradeço também aos professores que cursei suas disciplinas tanto na graduação quanto recentemente no mestrado. Da mesma forma, não há como deixar de agradecer aos colegas que cursei disciplinas e que fizemos trabalhos juntos, também na pós-graduação e anteriormente na graduação, na excepcional grad001! Sempre há algo a se levar para o resto da vida!

Para finalizar, agradeço à secretaria da pós-graduação e à representação discente do departamento por toda importância durante o período do mestrado, além do CNPq, pela bolsa de estudos que me proporcionou a dedicação exclusiva.

"Done is better than perfect."
(Sheryl Sandberg)

Resumo

Teste de mutação é uma técnica baseada em defeitos, comumente utilizada para se verificar a eficácia de testes de software (casos de testes, conjunto de testes). Consiste basicamente na introdução de mudanças sintáticas no código fonte de um sistema e verificação se os casos de testes deste sistema conseguem localizá-las (ou seja, acusar os defeitos). Tais mudanças sintáticas são chamadas mutações e os programas resultantes destas, mutantes. Uma vez que há dezenas de tipos possíveis de mutações definidas na literatura e, além disto, em vários pontos de código fonte podem ser introduzidos tais defeitos, os problemas mais desafiadores enfrentados comunidade é o alto esforço computacional necessário para se executar todos os conjuntos de testes para cada mutante gerado. Nas últimas quatro décadas, pesquisadores propuseram técnicas que visam a redução do custo computacional em fases do processo de teste de mutação, ex., redução do número de mutantes e da quantidade de casos de testes executados, em geração e compilação de mutantes, bem como em termos de otimização. Nesta dissertação, nós propusemos uma técnica para o processo de teste de mutação para reduzir o esforço computacional em sua fase de execução, inspirada na "execução ciente de variabilidade", uma tćnica inovadora que visa a redução de custo computacional no amplo espaço de configurações presente em sistemas configuráveis. Nós avaliamos a técnica proposta, que se mostrou viável para mutantes de maior ordem e concluímos que esta merece continuar a ser investigada em trabalhos futuros.

Palavras-chave: Testes de Mutação, Execução Ciente de Variabilidade, Testes de Software, Engenharia de Software.

Abstract

Mutation testing is a fault-based technique commonly used to evaluate the effectiveness of software tests (test cases, test suite). It basically consists in introducing syntactical changes, called mutations, into source code and checking whether the test cases distinguish them (i.e., locate those faults). Such changes are named mutations and the resulting modified programs, mutants. Since there are dozens of distinct mutation types, one of the most challenging problems faced by the community is the high computational effort required for testing the whole test suites for each generated mutant. Over the last four decades, researchers proposed techniques aiming effort reduction in mutation testing phases, e.g., either by reducing the number of mutants or by optimization. In this dissertation, we proposed a technique for mutation testing cost reduction in running time, inspired by variability-aware execution, a novel approach that achieves cost reduction in the large configuration space of configurable software systems. We assessed the viability of the proposed technique and concluded that it can be recommended for higher order mutation analysis and does deserve further investigation.

Keywords: Mutation Testing, Variability-Aware Execution, Software Testing, Software Engineering.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software testing aims at improving the quality of a software program. Software tests look for faults, errors, bugs in a program and can be classified in many types, considering the scope of what is being covered. Some types of tests can be automated, i.e., they are implemented to run the system with predefined inputs, retrieve the outputs and compare them with expected outputs.

Even if the complete set of the system tests does not get any failure, does not produce any error or retrieves all the expected outputs, it can still have poor quality. This happens due to many reasons, such as: the test set was not well implemented, has low coverage, focuses on limited parts of the system, does not test boundary inputs. Software testing aims at assessing the quality of a software system but, since this process can only show the presence of errors, not their absence [Dijkstra, 1972; Sommerville, 2015], the need of assessing the quality of the test set is quite important. *Mutation testing* [DeMillo et al., 1978], researched over last four decades and yet in an increasing trend [Jia and Harman, 2011; Ferrari et al., 2018], is currently one recommended technique for evaluating the quality of a system test set [Gopinath et al., 2016].

The Mutation testing technique consists of inserting syntactical faults into source code, generating different versions named *mutants*, and running tests over the modified programs, the, in order to distinguish them from the original code [Offutt, 1992]. A mutant is distinguished if at least one system test fails for it. In this case, the mutant is considered "killed". Given that, the system tests quality can be assessed according to their ability to kill mutants.

Mutation testing is known for demanding high cost to be performed and for a lack of satisfactory tools in the last decades, which are the reasons that this technique did not cause considerable impact in the industry. However, this scenario is appearing

to change in the last years. More tools are keeping up-to-date by its developers, for instance, Schuler and Zeller [2009], and even Google applies mutation testing in its quality development processes [Petrovic and Ivankovic, 2018].

1.1 Motivation

The underlying mutation testing process is a set of steps defined for performing mutation analysis over a system [Usaola and Mateo, 2010]. The process is interactive, since it evaluates the current test set of a system and, at each interaction, indicates whether more tests may be developed based on the information of killed mutants. It ends after the final test set reaches a predefined acceptable threshold for quality.

Some steps in mutation testing process require high effort to be concluded, given the higher number of possible mutant generations. As examples, the ones for generating mutants and running tests on each one are the most computationally expensive [Jia and Harman, 2011]. Besides that, the analysis step requires human effort, since the identification of *equivalent mutants*, those that are syntactically distinct, but semantically identical to the original program, can not be fully automated [Budd and Angluin, 1982].

Many techniques, tools and analysis were proposed, launched and performed over the last decades, aiming at reducing the cost of the most expensive steps of mutation testing process. They are classified into three groups: *do fewer*, *do faster* and *do smarter* [Offutt and Untch, 2001]. The first aims at reducing the number of mutants, the second aims at executing mutants as quickly as possible and the third aims at performing enhancements on execution and clever usage of computational resources. Indeed, there is room for improvement in all of them.

1.2 Proposal

In this dissertation, we propose a technique that aims at reducing the computational effort to execute the test cases over the mutants, i.e., reducing the running time. As in mutation testing, configurable systems face high computational cost problems. To test all program variants (possible configurations) in isolation is inefficient. As it is to analyze all program mutants in isolation. There are in literature many works aiming at reducing computational cost in both areas, such as *sampling* strategies, which attempt to detect as many faults as possible by analyzing only a small subset of configurations only (the most representative ones).

The technique we are proposing was inspired in variability-aware execution, a novel technique for configurable systems, which efficiently eliminates redundant program executions, running all configurations simultaneously at once [Thüm et al., 2012]. In other words, we brought the concept of variability-aware execution to the context of mutation testing, where *the mutants are handled as variability*. We needed tools for both generating mutants and, most importantly, for variability-aware execution. The last was achieved by VarexJ, a Java bytecode interpreter that implements such technique [Meinicke et al., 2016].

Although some existing mutation tools are available in the literature, they did not fit our purposes, since VarexJ defines a particular mechanism to represent variability. Therefore, we implemented a mutant generator tool to achieve this preliminary goal.

1.3 Results

In order to assess the viability of our proposing technique, we set up a study that consisted on choosing four subject systems on which mutants were generated and we implemented two baseline mutation testing techniques to compare the execution time. In addition, we expected that the study could lead us to answer the following research questions:

- **RQ1**: In which conditions is mutation-aware execution more efficient than other techniques?
- **RQ2**: Does mutation-aware execution scale, in terms of **(a)** the size of a system, **(b)** the number of test cases, and **(c)** the number of generated mutants?

Given the incipient nature of our study both the proposed technique and the variability-aware execution system presented some limitations. Therefore, in the current study, our technique was not more efficient than the baseline (it was three times slower for the largest system studied) tanking into account only first-order mutants (single faults). However, this technique can be more efficient for handling higher-order mutants ([Jia and Harman, 2009], in comparison with the implemented baseline. In addition, we are encouraged to proceed with this study in order to improve the current status of our proposed technique.

1.4 Contributions

This dissertation brings the following contributions to mutation testing community:

- A novel technique for cost reduction on one of the most critical steps of the mutation testing process: to execute test cases over the mutants;
- A mutant generator tool that implements three mutation operators and generates mutants in the classical way and in the specific purpose that the proposed technique requires;
- Identification of key improvements to our technique, to be implemented in future work.

1.5 Dissertation Outline

This dissertation is structured as follows. We present in Chapter 2 the core concepts of Software Testing, Mutation Testing, Variability-Aware Execution technique and a Java bytecode interpreter that implements that, as the basis for introducing our proposed technique for mutation testing cost reduction in Chapter 3. Chapter 4 is dedicated to the implemented mutant generator tool. In Chapter 5, we present the study setup for assessing the viability of our technique, discuss the results and answer the research questions. At last, we list the limitations of the current study and the threats to its validity. In chapter 6, we present related work concerning previous tools and techniques that resemble steps of our research. Finally, Chapter 7 concludes this dissertation and suggests improvements and studies as future work.

Chapter 2

Background

Software testing is a relevant activity of the software development process, which attempts to detect errors before each release. The more able to accomplish this, the more effective the software tests are. However, not revealing errors does not ensure them high quality. Mutation testing was proposed to assess the quality of software tests but, unfortunately, it demands computationally expensive steps. This chapter encompasses the background for a novel technique for cost reduction in mutation testing that we are proposing in this dissertation. Section 2.1 presents some definitions upon software testing. Section 2.2 discusses mutation testing, its process and challenges. In Section 2.3, we explain variability-aware execution and, in Section 2.4, a Java bytecode interpreter that implements it. At last, Section 2.5 concludes this chapter.

2.1 Software Testing Definitions

First of all, it is necessary some definitions related to software testing, which are referenced along this dissertation text in order to explain the core concept, mutation testing, and to foster the technique we are proposing.

Software Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user [Pressman, 2009]. Testing shows errors, requirements conformance and software performance. It can also be used to indicate software quality.

Unit Testing is a step inside a testing strategy, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods [Sommerville, 2015]. In Object-Oriented Programming (OOP), class testing for object-oriented software is the equivalent of unit testing for conventional software [Pressman, 2009].

Test Cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested [Sommerville, 2015].

Test Suite is a collection of test cases implemented to assess the quality of a target system. In literature, it is also defined as **test set**.

Test Coverage measures the effectiveness of a test suite on testing the code of an entire system. It is calculated by the ratio of elements executed (for example, statements and functions) against the overall elements required. Some companies have standards for test coverage. For instance, the system tests shall ensure that all program statements are executed at least once [Sommerville, 2015].

JUnit[1] is the most popular Java automation framework for Unit Testing. JUnit is integrated with the most popular Java development IDEs and can also be launched from the command line (standalone/batch mode). In JUnit, test cases are implemented as methods identified by `org.junit.Test` annotation and, so, they can be recognized by the framework. Test cases are encapsulated in meaningful test classes and there is also a mechanism to group such classes in test suites.

2.2 Mutation Testing

Mutation testing, also known as *mutation analysis*, is a fault-based testing technique that consists in introducing artificial faults into source code [DeMillo et al., 1978]. Such faults are syntactic changes that intend to represent real common programming mistakes [Just et al., 2014]. Each faulty version is called *mutant*. The main goal of this technique is, by running the test set of a software, to distinguish as many mutants as possible [Offutt, 1992] from the original program. When a mutant is distinguished (i.e., at least one test case fails on or gets a different result from the expected for it), it is deemed to be dead. Otherwise, alive. Since an alive mutant represents a fault undistinguished, it indicates that the test suite must be enhanced in order to kill more mutants and, therefore, having its quality increased. Mutation testing can be used for testing software at unit, integration, specification and even design levels [Jia and Harman, 2011; Ferreira et al., 2017].

Inspired by Usaola and Mateo [2010], we illustrate in Figure 2.1 a generic *Mutation Testing Process* composed of six steps, which are explained as follows. (1) Software testing phases are executed, which consists in, repeatedly, running the test set T and (2) fixing the original source code of the program P until no failure is found. (3) Mu-

[1] junit.org

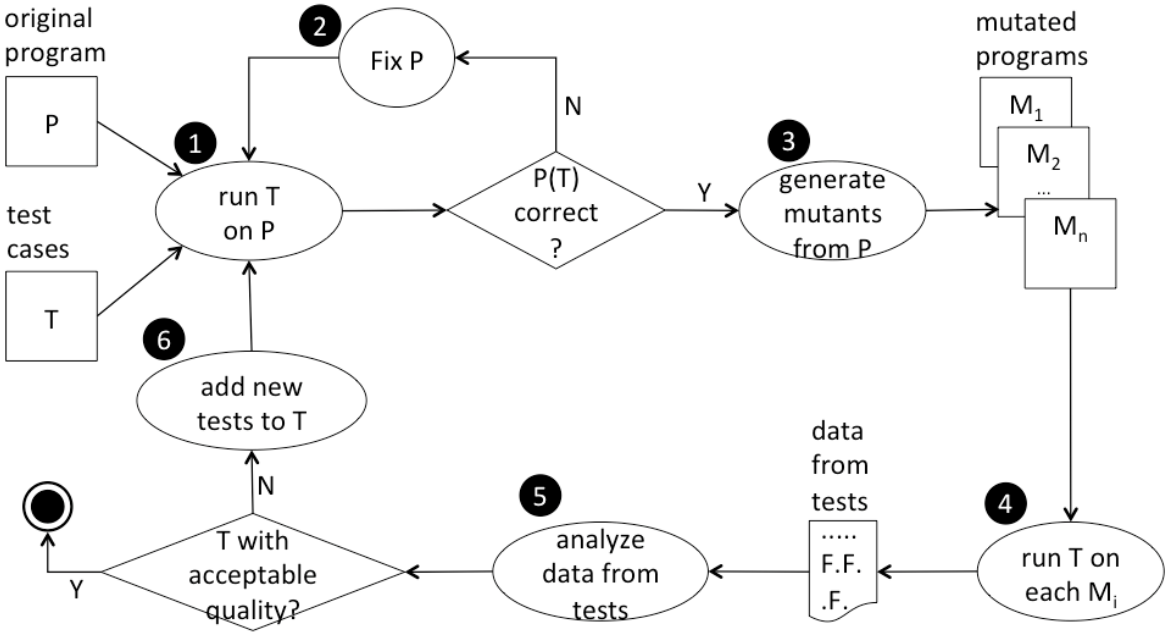

Figure 2.1. Generic mutation testing process.

tants are generated from P. (4) T is executed on each mutant. (5) All the information produced in the previous step is analyzed, in order to identify killed, live and equivalent mutants. The latter are mutants that can not be killed, since they are syntactically different but semantically identical to the original code. They will be better explained in the last paragraph of this section. If T achieves acceptable quality, the process ends. Otherwise, (6) new test cases are developed aiming at improving the quality of T and the mutation testing process starts again.

The *mutation score* is one of the existing kinds of quantitative measurement of the quality of a test suite [Zhu et al., 1997]. It is obtained by dividing the number of killed mutants by the number of all killable ones. Offutt [1992] formulates the mutation score as follows:

$$MS(P,T) = \frac{K}{(M-E)},$$

where P is the target program, T is its test set, K is the number of killed mutants, M is the number of generated mutants. E is the number of equivalent mutants. A quality threshold for the score may be defined in order to stop the mutation testing process (Figure 2.1). For instance, a software engineer may define 80% as a good threshold for a specific system.

The formalism that copes with the introduction of syntactic changes into source code is called *mutation operators*. Existing works define dozens of them in the literature

Listing 2.1. Mutation example over a Java source code

```
//(a) original code
public double simpleTax(double income) {
    int tax = 95.0;
    if (income > 500.0) {
        tax = income * 0.2;
    }
    return tax;
}

//(b) mutated code: AOR of * by +
public double simpleTax(double income) {
    int tax = 95.0;
    if (income > 500.0) {
        tax = income + 0.2;
    }
    return tax;
}
```

[King and Offutt, 1991; Ma et al., 2002; Ammann and Offutt, 2016]. Mutation operators are not the operators in a programming language. The most common are the ones that can be generally applied in most programming languages, such as, programming language operators replacement (e.g., arithmetic, logical and relational).

Listing 2.1 shows an example of the mutation operator Arithmetic Operator Replacement (AOR), where the Java multiplication operator `*` is replaced with Java addition operator `+`. There are also mutation operators for specific programming paradigms, such as object-oriented programming. For instance, the Super Keyword Deletion (ISK) operator removes the `super` keyword [Ma et al., 2002, 2005]. Mutation operators used in this dissertation are detailed in Chapter 4.

Regarding the way to analyze if a mutant is killed during the execution process, mutation testing techniques are classified into three types: Strong, Weak, and Firm mutation [Jia and Harman, 2011]. Strong mutation is the traditional mutation testing proposed by DeMillo et al. [1978] on which a mutant is considered killed if it produces different output from the original program. This technique was named this way after the proposal of weak mutation.

In weak mutation, the analysis is performed focusing on a component (variable reference, variable assignment, arithmetic expression, relational expression, and boolean expression) [Howden, 1982]. A mutant is created by changing a component and is considered killed if any execution of such component is different from the mu-

tant. "Firm mutation is thus the situation where a simple error is introduced into a program and which persists for one or more executions, but not for the entire program execution" [Woodward and Halewood, 1988]. In other words, by providing a sequence of intermediate possibilities. For example, a mutant is killed if the value of a variable is different from the original after the execution of a loop.

There are two considerable problems that prevent mutation testing from becoming a practical testing technique. One is the high computational cost of executing the enormous number of mutants against a test set during the mutation testing process. There are many computational cost reduction techniques proposed in literature over last decades, as the surveys Usaola and Mateo [2010], Jia and Harman [2011] and Ferrari et al. [2018] present. Techniques vary from reducing the number of test cases, the number of mutation operators, until runtime, compile-time, and parallel execution optimizations. The technique proposed in this dissertation also aims at cost reduction.

The other one is the equivalent mutant problem. A mutated program is said to be equivalent to the original program if they always produce the same output on every input. In other words, when they are submitted to the same input (or test case), both produce the same output (or results), whatever the input is. Thus, equivalent mutants are never killed and must be disregarded of the mutation testing process. Unfortunately, automatically finding all equivalent mutants is an undecidable problem, as was proven by Budd and Angluin [1982], and is only partially achieved [Offutt and Pan, 1997]. Therefore, identifying equivalent mutants requires a considerable amount of human effort in mutation testing process.

Mutants generated by single faults are called First Order Mutants (FOMs). Higher Order Mutants (HOMs) are generated by more than one fault [Jia and Harman, 2009]. Given their underlying nature, HOMs are more likely to be killed than FOMs. However, a small fraction of HOMs represents subtler faults, making them harder to be killed (distinguished by tests), which means that such HOMs are more representative than their constituent FOMs. Therefore, the ability to generate these mutants will lead to a decrease in the number of sufficient mutants in the mutation testing process, reducing its cost. Unfortunately, the search space of such valuable mutants increases exponentially in comparison to that of FOMs. As a consequence, the interest in discovering them has increased last years [Lima and Vergilio, 2018] and much is concerning at proposing optimized approaches for finding as many representative HOMs as possible. The results of such studies are expected to help the construction of algorithms to generate representative HOMs, in order to incorporate them in mutation testing tools [Harman et al., 2014]. In addition, it may be worth a tool that is able to inform the HOMs that have not been killed during a mutation testing process.

Besides assessing the quality of test suites, mutation testing has other applications, such as fault localization and bug fixing, test suite generation, test suite minimization, test prioritization (or selection), program verification, and security analysis [Just et al., 2014; Wang et al., 2017].

2.3 Variability-Aware Execution

Before defining variability-aware execution, it is necessary to explain the context from which this technique emerged. Highly Configurable Systems are software systems highly customized to provide many configurable options. They are expected to improve reuse and increase flexibility, quality, and security [Pohl et al., 2005]. Configuration options can be of many distinct types, e.g. boolean (enable/disabled), numerical, etc. We are inspired by a work focused only on binary (boolean) options, named *features*. Each distinct combination of features is called *configuration*. This subject is largely studied by researchers which are interested in investigating the *variabilities* of configurable systems. There are two main concerns regarding configurable systems. Testing all configurations of a system is impracticable because the number of configurations grows exponentially in relation to the number of features. In addition, a configuration can face unexpected behavior (e.g., crash, error, failure, bug or distinct output) caused by *feature interaction* [Nhlabatsi et al., 2008], which does not occur when the enabled features are tested in isolation. Many techniques in the literature were proposed to mitigate this problem [Meinicke et al., 2016]. In this dissertation, we are focusing on Variability-Aware Execution, applying it on a distinct, yet related context: Mutation Testing.

Variability-aware execution consists of a single run of software having all configurations executed "simultaneously". In other words, all program variants are analyzed at once, thereby reducing the analysis effort by *sharing* calculations and returning *aggregated* results. Variability-aware analysis is achieved by means of *runtime variability*, a variability encoding rather distinct from *compile-time variability*, one of the most widely used product generation techniques for configurable systems.

Compile-time variability is achieved by conditional compilation (e.g., `#ifdef` compilation directives in C/C++), where each distinct product is compiled separately and each "executable" version is not aware of any variability. In runtime variability, variabilities are identified in the source code by means of programming language conditional structures (e.g., `if/else` statements in Java) generating, after compilation, a so-called *metaproduct* [Thüm et al., 2012], in which all features can be enabled/disabled

Listing 2.2. Variability representation for VarexJ

```
1  ...
2  import gov.nasa.jpf.annotation.Conditional;
3  public class FeatureExample {
4      @Conditional public static boolean SECONDS;
5      String method1(long time) {
6          if (!valid(time)) return null;
7
8          if (SECONDS) {
9              return format(time, "hh:mm:ss");
10         }
11         else {
12             return format(time, "hh:mm");
13         }
14     }
15     void example() {
16         System.out.println(format(40953000L));
17     }
18 ...
19 }
20 ==========================================================
21 VarexJ console output after invoking "example()" method:
22     <11:22:33> : SECONDS
23     <11:22> : !SECONDS
```

at runtime. Finally, *sharing execution* consists in eliminating redundant executions, instructions being executed on different values at the same time and handling *aggregated results*. As expected, accomplishing variability-aware execution comes with additional overhead for each computation [Meinicke et al., 2016].

2.4 VarexJ

VarexJ is a variability-aware Interpreter for Java bytecode [Meinicke et al., 2016], implemented on top of Java PathFinder (JPF) [Havelund and Pressburger, 2000]. JPF is a model checker that uses Java bytecode instructions as transitions between states and it works as an interpreter when configurations of model checking are disabled [Meinicke, 2014].

VarexJ has a particular variability representation to distinguish features from system variables: the `Conditional` annotation was implemented for this purpose. In Listing 2.2, we show one example of source code adapted for VarexJ execution. Lines 2

Listing 2.3. Error caught by VarexJ

```
1  ...
2  public class ErrorExample {
3      @Conditional public static boolean F1;
4      @Conditional public static boolean F2;
5      int method2(int x) {
6          int i = x + 2;
7          if (F1) i - 4;
8          if (F2) i - 6;
9          return x/i;
10     }
11 ...
12 }
13 ================================================================
14 VarexJ console output (shortened) after invoking method2(8):
15 ...
16 if (F1&F2):
17 java.lang.ArithmeticException: division by zero
18     at ErrorExample.method2(ErrorExample.java:19)
19     at ...
```

and 4 show how to adapt source code to make VarexJ able to identify variabilities, for instance, the feature `SECONDS`. The executions of lines 6, 8 and 16 are *shared* among both possible configurations (`SECONDS` enabled or disabled). The returned value of `method1` carries an *aggregated result*, as lines 21-23 demonstrate. On a single run, lines 9 and 12 are executed, but not shared. Line 9 is executed only for the context `SECONDS` whilst line 12, only for the context `!SECONDS`.

When a VarexJ execution for a program gets exceptions or errors, as VarexJ is aware of variability, it logs the context (combination of features) that leads to them as well as the Java stack trace. Listing 2.3 shows a source code (lines 2-12) and an excerpt of the VarexJ output error (lines 16-18) for a particular execution of the program: when the features `F1` and `F2` are simultaneously enabled and, for the input argument `8`, a division by zero error occurs.

2.5 Conclusion

This chapter reviewed the main concepts of software testing and describes mutation testing. We presented the mutation testing process and discussed that some of its steps require considerable effort (computational or human) to be executed. This chapter also

presents variability-aware execution, a technique that achieves computational effort reduction on testing configurable systems and introduces VarexJ, a Java bytecode interpreter that implements variability-aware execution. The presented background was necessary to formulate the technique we are proposing in this dissertation, which aims at reducing computational effort in the step of test cases execution, in the mutation testing process. The next chapter presents the technique itself, for which we brought variability-aware execution to the context of Mutation Testing, treating mutations as variabilities (or features).

Chapter 3

The Proposed Technique

In this chapter, we propose a technique aiming at reducing computational effort in one of the most expensive steps of the mutation testing process: running the test set for each generated mutant. It consists in performing mutation testing over a given system by means of variability-aware execution. In Section 3.1, we define a new concept and provide a general overview of our proposed technique. Section 3.2 presents the steps for generating mutants in such a way they can be recognized as variability by the proposed technique. Next, Section 3.3 presents an algorithm for identifying killed and alive mutants. Finally, Section 3.4 concludes this chapter.

3.1 Definition and General Overview

Definition. The proposed technique consists in generating mutants and making them identified as variability runtime (or enabled/disabled features). Therefore, all mutants, as well as the original source code can be executed in a single run, via variability-aware execution, reducing the underlying computational effort of the classical approach. For this context, we define the concept of **Mutation-Aware Execution**, that is simply variability-aware execution for which the variability is represented by the mutants. Mutation-aware execution will be referenced as our proposed technique in the remainder of this dissertation.

It is worth to mention that VarexJ (see Section 2.4) is the most complete implementation of variability-aware execution available in the literature [Meinicke, 2014]. Therefore, we choose VarexJ as the tool to accomplish and evaluate our proposed technique. In other words, it is the tool for which the mutants will be generated and on which they will be executed. In addition, it will provide information for killed mutants identification.

Figure 3.1 shows an overview of the steps for achieving mutation-aware execution. Ellipses represent the steps and the other forms represent the artifacts already available or produced, which are inputs for the steps. The four main steps are described as follows. The first step is the generation of mutants as runtime variabilities, from the original source code of a system. The second step is the creation of a specific purpose class that will execute the test cases of a system in VarexJ. With the previous two artifacts, the third step consists in running the JUnit tests on VarexJ and retrieving its output log, as well as the running time it lasted. Finally, the fourth step consists in processing the VarexJ output log to retrieve information about killed and alive mutants. Section 3.2 details the first two steps and their respective artifacts while Section 3.3 is about the last two steps and artifacts.

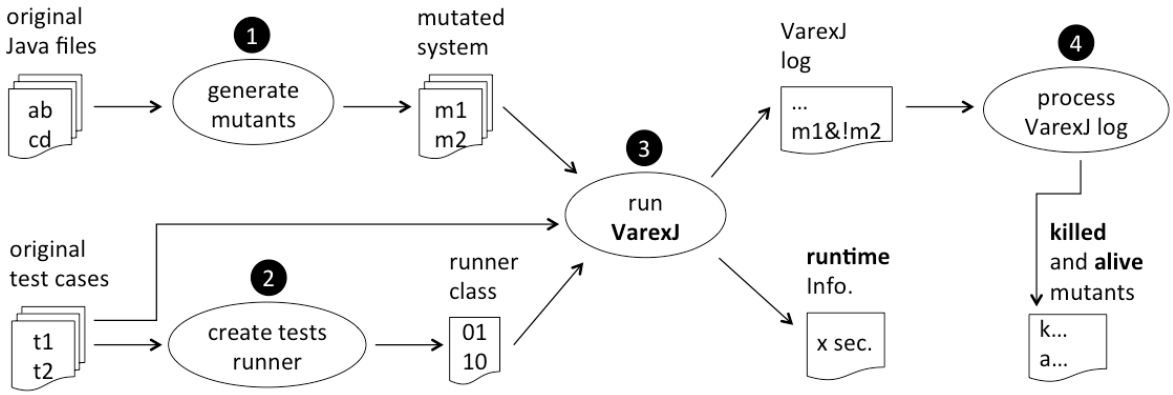

Figure 3.1. The proposed technique process overview.

3.2 Mutation-Aware Execution in VarexJ

VarexJ provides a quite basic integration with JUnit Framework, accomplished by a workaround implementation: the `gov.nasa.jpf.util.test.TestJPF` class, that calls the main engine for Java bytecode processing. Such integration is not transparent to the end user, the one who is adapting any configurable system tests for VarexJ. Therefore, some implementation is required so that the test cases could be the starting point of the variability-aware execution. The lack of annotations (e.g. `@Before`, `@After`, `@BeforeClass` and `@AfterClass` from package `org.junit`) is a considerable limitation and it leads to implementations of workaround solutions. Listing A.4 contains an example of the generated class for a particular system evaluated in this dissertation.

A mutant, as mentioned in Section 2.2, is related to a syntactical change applied to the original source file, creating a new (mutated) file that differs from the original by a delta (the syntactical change applied). There are in the literature many tools that generate mutants, for distinct programming languages, including Java. However, in order to execute mutants as variability of a system in VarexJ, their classical mutants generation techniques do not fit our purpose. As is done to represent all features of a highly configurable system in VarexJ, we also need to represent the original version and its mutants in a single file.

For instance, let's consider a file containing the statement `a = b * 3;` on which two mutants are to be applied: replacing * with `+` and with `-`. While a classical mutant generator tool creates a new file to represent each mutant, one containing `a = b + 3;` and another `a = b - 3;`, we need a single file encompassing all three variants: `a = mut2 ? (b - 3) : (mut1 ? : (b + 3) : (b * 3));`. Since we did not found any tool available that performs this kind of mutation for Java systems, we decide to develop our own mutant generator tool that provides the required mutation-aware systems, composed by mutation-aware classes. This mutation generator tool is presented in Chapter 4.

Listing 3.1 depicts an example containing two code excerpts. Lines 1-15 comprise the first one. In line 3, the mutant `mut` is defined as a variability for VarexJ. Lines 4-15 show a method containing one mutated code. Line 11 represents an arithmetic expression of the original source code, while lines 7-9 represent the code for mutation. Line 8 is the mutation itself, where the operator `*` was replaced with `+`. Finally, lines 17-20 comprise the second code excerpt. The statement in line 19 tests the method defined above.

3.3 VarexJ Running, Output Processing and Analysis

Once we have a mutation-aware system and the environment for running its JUnit tests, by running it as JUnit application, VarexJ starts automatically the variability-aware execution, running test cases (the methods annotated with `@Test`). During VarexJ execution, it is aware of all shared and configuration dependent instructions, as well as the configuration of variabilities that leads aggregated results of variables and return statements. Once an error, failure or exception is reached, VarexJ logs it. For example, Figure 3.2 shows the execution traces when line 19 of the Listing 3.1 is executed, for three distinct situations. (i) The trace in left: Normal execution, with `mut = false`.

Listing 3.1. Method with a mutant and a test for it.

```
1  //Main code
2  ...
3  @Conditional public static boolean mut;
4  public static double simpleTax(double income) {
5      double tax = 95.0;
6      if (income > 500.0) {
7          if (mut) {
8              tax = income + 0.2;
9          }
10         else {
11             tax = income * 0.2;
12         }
13     }
14     return tax;
15 }
16
17 //Test case
18 ...
19 assertEquals(120.0, simpleTax(600.0));
20 ...
```

(ii) The trace in center: Normal execution, with `mut = true`. (iii) The trace in right: Variability-aware execution. The number inside the rectangles represents the executed lines of the subject listing.

We can observe that, apart of variability-aware execution, if we want to know the results of the test case with and without mutation, it is necessary to execute it two times (traces in the left side and in the center of Figure 3.2). Using variability-aware execution, it is necessary only one execution (right side). In this case, the execution of lines 19, 5, 6, 7, 14 and 19 again is shared and the variable `tax` carries aggregate results. Lines 8 and 11 are executed sequentially, but their execution is not shared, since they are executed for distinct contexts (`mut` and `~mut`, respectively). Finally, the `AssertionError` occurs only for the `mut` context.

While VarexJ runs, it logs all information necessary for a variability-aware analysis or, in our case, a mutation-aware analysis. The output log contains, basically, three pieces of information: starting time in the beginning, finish time at the end, and information about test success or failure in the middle. If the test case succeeds, VarexJ logs the message "no errors detected". Otherwise, it logs the failure information composed by a boolean expression and the underlying stack trace. When one variabil-

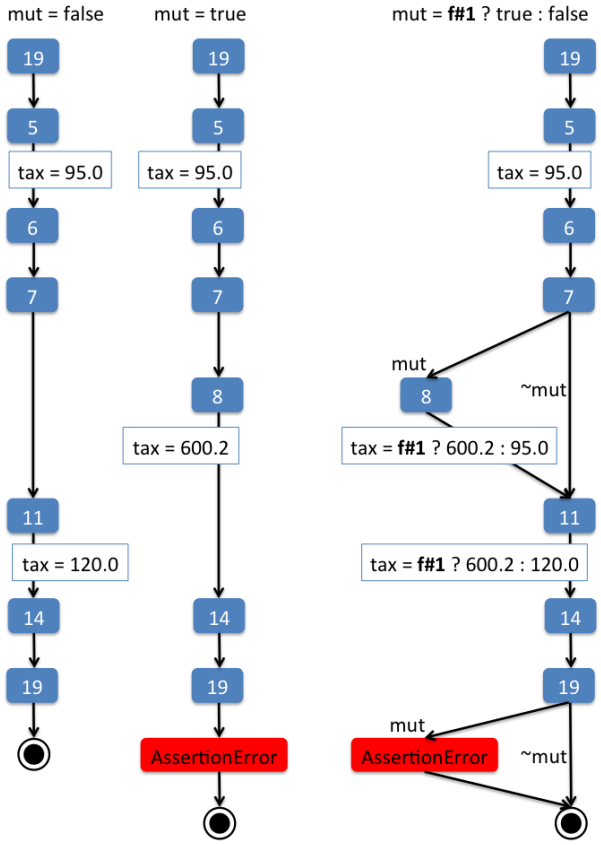

Figure 3.2. Normal execution (left and center) and mutation-aware execution (right) trace examples.

ity of the system or a combination of enabled/disable variabilities leads to a failure, the respective configuration is represented by the boolean expression. In the respective stack trace part, it is possible to identify the method test case (method) that has failed.

Listing 3.2 shows a VarexJ log example. Lines 8-10 identify the mutants (variabilities). The boolean expression in line 13 and the information in line 18 indicate that the test case `testEquilateral` fails for all combinations of mutants that evaluate such expression to `true`. Therefore, the expression leads to `true` for the following 5 out of 7 possible combinations with at least one mutant enabled: `{mut2}`, `{mut2,mut3}`, `{mut1}`, `{mut1,mut2}` and `{mut1,mut2,mut3}` . For instance, `{mut2,mut3}` means `mut1` disabled, `mut2` enabled and `mut3` enabled.

In this dissertation, we are focusing only on *First Order Mutants* (FOMs), i.e., the ones generated by only one single change in source code. Thus, in the example, two FOMs leads `testEquilateral` to an assertion error: `mut1` and `mut2`. Therefore, the mutants `mut1` and `mut2` were killed and `mut3` remained alive. To accomplish this,

Listing 3.2. Mutation example over a Java source code

```
1   running jpf with args: +search.class=.search.RandomSearch +classpath=...
2     VarexJ v1.0 (BDD, TreeChoiceFactory, HybridStackHandlerFactory(OneStack ->
          Buffered))
3   ============================================================ system under test
4   triangle.tests.varexj.TriangleTestSuiteForVarexJReflect.runTestMethod()
5   ============================================================ search started: 6/24/18 5:28 PM
6   ============================================================ Random Search
7   [WARNING] non-public peer class: gov.nasa.jpf.vm.JPF_java_lang_System$1
8   Found feature #1 - mut1 @triangle.Triangle
9   Found feature #2 - mut2 @triangle.Triangle
10  Found feature #3 - mut3 @triangle.Triangle
11  ============================================================ error 1
12  gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
13  if (mut2|(mut1&!mut3):
14  java.lang.AssertionError:
15          at gov.nasa.jpf.util.test.TestJPF.fail(TestJPF.java:167)
16          at gov.nasa.jpf.util.test.TestJPF.assertEquals(TestJPF.java:1051)
17          at gov.nasa.jpf.util.test.TestJPF.assertEquals(TestJPF.java:1055)
18          at triangle.tests.varexj.Triangle1TestForVarexJFull.testEquiliteral(
                Triangle1TestForVarexJFull.java:41)
19          at java.lang.reflect.Method.invoke(gov.nasa.jpf.vm.
                JPF_java_lang_reflect_Method)
20          at gov.nasa.jpf.util.test.TestJPF.runTestMethod(TestJPF.java:650)
21  ============================================================ snapshot #1
22  thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,lockCount:0,
        suspendCount:0}
23    call stack:
24          at gov.nasa.jpf.util.test.TestJPF.runTestMethod(TestJPF.java:652)
25  ============================================================ results
26  error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "if (mut0|mut2): java.lang.
        AssertionError:   at gov..."
27  ============================================================ search finished: 6/24/18 5:28 PM
```

Algorithm 1 shows the simplest form to identify killed and alive mutants by VarexJ output processing. In lines 6 to 9, each *boolean* expression is evaluated for each of its constituent mutants. In line 7, the current mutant is enabled (assigned to *true*) and the remaining are disabled (assigned to *false*) to get the expression evaluated.

Algorithm 1 for finding killed and alive mutants

```
1:  function KILLEDANDALIVEMUTANTS(varexjLog, allMutants)
2:      killedMutants ← ∅
3:      aliveMutants ← ∅
4:      for each expression expr ∈ varexjLog do
5:          mutants ← retrieve all mutants from expr
6:          for i ← 1 to |mutants| do
7:              killed ← evaluate(expr|mutants_i ← true, mutants_≠i ← false)
8:              if killed then
9:                  killedMutants ← killedMutants ∪ {mutants_i}
10:     aliveMutants ← allMutants \ killedMutants
11:     return killedMutants, aliveMutants
```

3.4 Conclusion

In this chapter, we defined Mutation-Aware Execution and proposed a novel technique that consists of performing mutation testing over a given system by means of variability-aware execution. This technique relies on VarexJ to execute the generated mutants. We also presented an overview of the process, describing each of its constituent steps (test adaptation, mutant generation, VarexJ execution, and log processing) as well as the artifacts it produces and consumes. Our proposed technique relies on strong mutation testing since we take into account the result of the entire test suite to find the killed mutants. In the next chapter, we will describe in detail our developed mutant generator tool, introduced in Section 3.2.

Chapter 4

MutVariants: A Mutant Generator Tool

A classical mutant generator tool introduces syntactical faults into original source code, creating faulty variations of the target system, the mutants. Since the technique we proposed needs mutants being represented with the particular variability mechanisms and we did not find any available tool that provides a similar solution, we opted for the implementation of a prototype tool. Therefore, this chapter depicts the tool we developed to support the technique we proposed in Chapter 3. In Section 4.1, we contextualize mutation tools, a particular mutant generation strategy, and justify why we decided to develop our tool. Section 4.2 presents the architecture and overview of our tool. In Section 4.3, we present some details of design and implementation and, in Section 4.4, we discuss how our tool can be extended. Section 4.5 provides some examples of generated mutants. Finally, Section 4.6 concludes this chapter.

4.1 Motivation

As described in Chapter 3, our proposed technique requires mutants generated in such a way they can be recognized as program variants by VarexJ (see Section 2.4). This Java bytecode interpreter runs a *metaproduct*, a compiled program containing all of its features codified in conditional statements. In mutation testing, a mutant generation approach that provides a single system having all mutants codified similarly to the features in variability-aware execution has already been proposed [Untch et al., 1993]. Such approach is called *Mutant Schemata* and the mutated system, *metamutant*. Mutant schemata was proposed for improving speedup in mutation testing, since compilation and loading are performed just once [Madeyski and Radyk, 2010].

Since mutation-aware execution is also based on mutant schemata, at this stage of our research we needed a mutant generator tool for Java systems in such specific format. Unfortunately, little research concerning mutant schemata is available in the literature, the main work by Untch et al. [1993] is not recent, and tools specifically developed to evaluate such approach are no longer available for download. In fact, the most recent tools that implement mutant schemata either do not fit our purposes or are out of date.

Table 4.1 summarizes mutation testing systems that, among many functionalities, generate mutants using mutant schemata approach. MuClipse is an Eclipse Plugin which provides a bridge between the existing MuJava [Ma et al., 2005] mutation engine and the Eclipse IDE. Judy [Madeyski and Radyk, 2010] generates each mutant in a separate method and controls the invocations by means of AspectJ[1] [Kiczales et al., 2001]. It follows a quite distinct process if compared with variabilities in Listings 2.2 and 2.3. Bacterio [Mateo and Usaola, 2012a] became a commercial tool. Although Javalanche [Schuler and Zeller, 2009], was released in the last decade, it has been constantly updated. PIT [Coles et al., 2016] is the most recent tool. Mateo and Usaola [2012b] published a work for which they applied mutant schemata generation in such a way that most resembled our needs. Nevertheless, we did not find the implementation of their proposed technique, named MUSIC, available for download.

Table 4.1. Mutation systems known for mutant schemata generation.

Tool	*Description*
Bacterio[2]	No longer available. Latest available version was commercial.
Javalanche[3]	Available, but generates mutant schemata at the bytecode level.
Judy[4]	Available, but mutant schemata generation and activation are distinct from what we need.
MuJava[5]/MuClipse[6]	Available, but out of date.
PIT[7]	Available, but generates mutant schemata at the bytecode level.

Since we could not use any of the available systems, there were two options left for the *metamutants* achievement: (1) taking an existing classic mutant generator tool for providing all isolated mutated systems and developing a utility for merging them

[1]https://www.eclipse.org/aspectj
[2]https://alarcos.esi.uclm.es/per/preales/bacterio/bacterio.htm
[3]https://github.com/david-schuler/javalanche
[4]http://mutationtesting.org
[5]https://cs.gmu.edu/ offutt/mujava
[6]http://muclipse.sourceforge.net
[7]http://pitest.org

into a single *metamutant* one; or (2) implementing, from scratch, a specific purpose prototype tool for mutants generation. We decided on the latter option for three reasons. First, we believed it would take less effort. Second, we would not be subject to any tool limitations. The third reason for implementing our own tool is to provide thorough control of how, where and which mutants could be generated at a time, besides tracing them during analysis. In addition, the specific code for VarexJ `@Conditional` annotation could be fully automated. We refer to the prototype in the remainder of this dissertation as **MutVariants**.

4.2 Overview

MutVariants is a mutant generator tool available at GitHub[8] as a Maven[9] project. It requires two pieces of information for performing mutation in the project's source code, which are, (i) the (input) paths for original classes and for its dependent `jar` files, and (ii) the (output) path where the mutated classes must be generated. As dependency, MutVariants requires only the JavaParser[10] library, with its JavaSymbolSolver module enabled.

Mutants generation in MutVariants consists of six steps, that are described as follows, for each original Java source code file. (1) It calls `JavaParser` core to realize a parse and to provide a `CompilationUnit` object, which is an Abstract Syntax Tree (AST) representation of the source code. (2) It clones the CompilationUnit and (3) uses the JavaParser's provided *Visitor* Design Pattern [Gamma et al., 1994] implementation to traverse and manipulate the clone's AST. When a mutable node is found, (4) MutVariants generates a new mutated node and replaces the current node by the mutated one in the AST.

MutVariants realizes mutations from leaves to root, ensuring precedence. For instance, in expressions like `a + b / c`, the division operator, which has precedence over the addition, is mutated at first, generating a mutated expression like `a + (mut1 ? b*c : b/c)` and, at last, the addition, resulting in `mut2 ? a - (mut1 ? b*c : b/c) : a + (mut1 ? b*c : b/c)`. After the AST be traversed and mutants be generated, (5) MutVariants generates code for the `@Conditional` and the mutant variable declarations. (6) Finally, the mutated `CompilationUnit` is saved as a Java file with the same name as the original, in the desired output folder, respecting its underlying package folder structure.

[8] https://github.com/jpaulodiniz/MutVariants
[9] https://maven.apache.org
[10] https://javaparser.org

4.3 Design and Implementation Decisions

MutVariants does not have a graphical user interface (GUI), does not provide execution via command-line, and is not integrated with any IDE yet. However, it has implemented some features such as:

- mutation rate: a probability for generating a mutant on a given mutation candidate spot in the source code, ranging from 0.0 to 1.0 (the default rate);
- all mutations per spot: when a mutation operator has more than one mutation available for a given spot (for instance, **AOR** in Table 4.2 can generate four mutants for the expression `a * b`), MutVariants can generate either only one (randomly chosen) or all mutants.
- two mutation strategies for binary expressions: mutate only one binary expression per statement and mutate all binary expressions in source code;
- option for mutating loop conditions: it is useful when reducing infinite loops due to mutants is necessary;
- facility for reading project input and output paths from a Java `.properties` file.

As introduced in Section 2.2, dozens of **mutation operators** were defined in the literature. In addition, there are many distinct naming for the same operators as well as one operator being divided into less complex ones, which makes the search for them quite confusing. However, studies demonstrated that just a few mutation operators can be considered representative enough for Mutation Testing, without significant loss of effectiveness [Offutt et al., 1996; Barbosa et al., 2001; Siami Namin et al., 2008; Banzi et al., 2012]. For instance, the study of Offutt et al. [1996] is the most referenced in literature and its conclusion provides a clear set of five operators.

We decided to implement three out of those five operators in MutVariants in order to begin the assessment of our proposed technique. Table 4.2 describes and exemplifies them. We decided not to implement specific object-oriented mutation operators in our tool due to two reasons. First, we aim to provide and validate mutation-aware execution technique as general as possible. Second, we acknowledge limitations of the mutant schemata technique that make it impossible to represent field/method access modifier keyword replacement (e.g. `public` with `private`), for instance [Mateo et al., 2013].

Therefore, MutVariants current version mutates only binary expressions. Since JavaParser core provides extra information besides the AST, MutVariants had to ensure

Table 4.2. Mutation operators implemented.

Mutation Operator		*Example in Java*	
		Original	*Mutation*
AOR	Arithmetic Operator Replacement	`a - b`	`a * b`
ROR	Relational Operator Replacement	`a <= b`	`a > b`
LCR	Logical Connector Replacement	`a || b`	`a && b`

not to mutate expressions such as `String` concatenations and (in)equalities between objects or between an object and `null`. To overcome this issue, MutVariants requests expression types for the previously mentioned JavaParser module, JavaTypeSolver.

Based on all those definitions, we had to define a template for mutants in the source code, in mutant schemata representation. For a single mutant, the simplified template is defined as

$$(mut\# \ ? \ (mutatedExpr) : (originalExpr)),$$

where *originalExpr* has the format *operand*1 *originalJavaOperator* *operand*2 and *mutatedExpr* has the format *operand*1 *mutatedJavaOperator* *operand*2. When n mutants can be generated for a single spot, the template can be redefined, iteratively, as

$$[(mut\#i \ ? \ (mutatedExpr\_i) :]\{i = n : 1\} \ (originalExpr)[])]\{n\}$$

4.4 Discussion

In object-oriented software testing, Ma et al. [2002] classified faults in classes as occurring at the intra-method level, the inter-method level, the intra-class level, and the inter-class level. Therefore, mutation operators have been proposed to address those levels of faults. Simplifying, for Java, mutation operators are classified into method-level (inter-method) and class-level operators (the last three) [Kim et al., 2013].

Traditional mutation operators for procedural programs will suffice for representing method-level faults. For class-level, it is required mutation operators generating faults due to the object-oriented specific features (encapsulation, inheritance, polymorphism, and dynamic binding) [Ma et al., 2002]. Class-level mutation operators can change access modifiers, inheritance tree, constructors, annotations, etc.

MutVariants implements three method-level mutation operators. Variability-aware execution implemented in VarexJ together with MutVariants (the tooling for

our proposed technique) can support basically the range of all other method-level mutation operators already defined in the literature since all mutation operators that change *statements* and *expressions* in Java can be implemented in MutVariants. For example, Unary Operator Insertion (UOI), Statement Block Removal (SBR), Assignment Shortcut Operator Replacement (ASR), Absolute Value Insertion (ABS), and many others.

The implementation of class-level mutation operators into MutVariants, and to employ them in mutation-aware execution is out of the scope of this dissertation due to two reasons. First, it would require combining metamutant and source code manipulation techniques already implemented with other mechanisms such as bytecode modification technique, a quite similar problem faced by the developers of Judy mutation system [Madeyski and Radyk, 2010]. Second, our proposed technique aims to improve running time due to a large number of mutants that can be generated for a system, but class-level mutation operators usually do not produce enough mutants to be considered a problem [Kim et al., 2013].

4.5 Examples of Generated Metamutants

This section contains some examples of mutated classes by MutVariants, using distinct configurations. Listing 4.1 shows a complete class, containing all configurations for VarexJ, mutants declarations as `static boolean` variables and two mutants in distinct statements, at lines 6 and 7. The original class was already depicted in Listing 2.1(a).

Listing 4.2 shows the same method `simpleTax` mutated by two mutation operators with all possible mutants, five in each case, at lines 5 in (a) and 13 in (b).

Finally, Listing A.2 shows an original Java class file used in our evaluations and Listing A.3 shows all 128 mutants generated by MutVariants.

4.6 Conclusion

This chapter presented MutVariants, the prototype tool for generating mutants that we developed. It also provided some arguments and information concerning mutants generation activity. Our tool generates mutants in mutant schemata representation, using three of the most representative mutation operators, known from the literature. In addition, we explained some technical information about MutVariants and provided some examples of the generated mutants. In the next chapter, we evaluate the proposed technique and present the results.

Listing 4.1. Mutation customizations for VarexJ

```
1  import gov.nasa.jpf.annotation.Conditional;
2  public class Taxes {
3      @Conditional public static boolean mut1=false, mut2=false;
4      public double simpleTax(double income) {
5          double tax = 95.0;
6          if ((mut1 ? (income < 500.0) : (income > 500.0))) {
7              tax = (mut2 ? (income + 0.2) : (income * 0.2));
8          }
9          return tax;
10     }
11 }
```

Listing 4.2. Examples of more than one mutant

```
1  //(a) all 5 possible AOR mutants at the same spot
2  public double simpleTax(double income) {
3      double tax = 95.0;
4      if (income > 500.0) {
5          tax = (mut4 ? (income % 0.2) : (mut3 ? (income / 0.2)
               : (mut2 ? (income - 0.2) : (mut1 ? (income + 0.2)
               : (income * 0.2))));
6      }
7      return tax;
8  }
9
10 //(b) all 5 possible ROR mutants at the same spot
11 public double simpleTax(double income) {
12     double tax = 95.0;
13     if ((mut5 ? (income != 500.0) : (mut4 ? (income == 500.0)
          : (mut3 ? (income <= 500.0) : (mut2 ? (income < 500.0)
          : (mut1 ? (income >= 500.0) : (income > 500.0))))))) {
14         tax = income * 0.2;
15     }
16     return tax;
17 }
```

Chapter 5

Mutation-Aware Execution Viability Assessment

In Chapter 3, we proposed mutation-aware execution, a technique for mutation testing cost reduction based on variability-aware execution, on which mutants handled as variability. In this chapter, we evaluate the proposed technique in terms of running time, comparing it against the mutation tests in its classical approach and the mutant schemata approach [Untch et al., 1993]. We choose four systems, which are either open-source or well established in previous studies in the literature, for such assessment. Section 5.1 presents study design, subject systems and research questions we expect to answer. Section 5.2 presents the results, attempts to answer the RQs and discusses the findings. In Section 5.3, we present the limitations we faced and threats to the validity of our results. Finally, Section 5.4 concludes this chapter.

5.1 Study Setup

Mutation testing process contains six steps presented in Figure 2.1 and, for some of them, there can be found in the literature studies and tools aiming at achieving computational effort reduction. We propose mutation-aware execution, a *do smarter* approach aiming computational effort reduction for the test cases execution step of mutation testing process, which leads us to formulate the following research questions:

- **RQ1**: In which conditions is mutation-aware execution more efficient than other techniques?
- **RQ2**: Does mutation-aware execution scale, in terms of **(a)** the size of a system, **(b)** the number of test cases, and **(c)** the number of generated mutants?

In order to answer the research questions, mutation-aware execution running time must be *compared against other mutation testing techniques*. In addition, to be fair, it is essential that the comparison occurs running the *same systems* with exactly the *same mutants*. In addition, we could not perform our study with all of the available test cases of the subject systems and an additional step of the study setup was necessary: *test cases removal*. Figure 5.1 gives a general overview of the study setup, which is replicated for each of the four evaluated systems, and the next subsections describe each of such items.

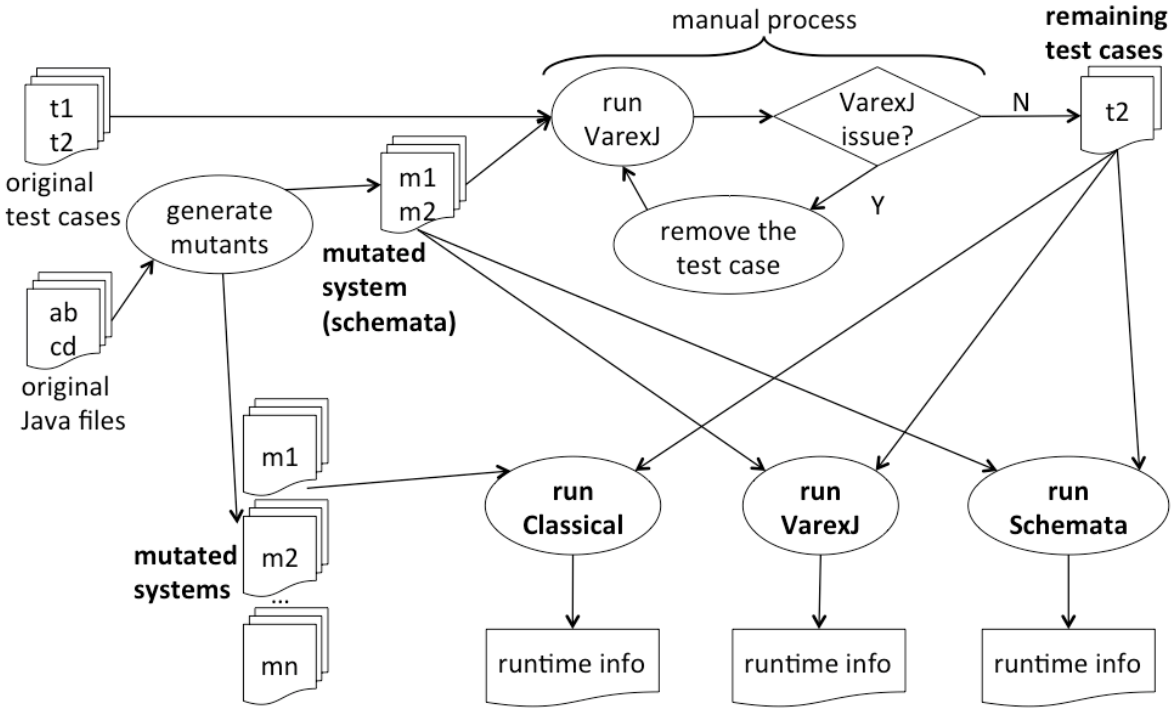

Figure 5.1. Study setup overview.

5.1.1 Subject Systems

We choose four systems to evaluate the viability of mutation-aware execution, namely Triangle, Monopoli, Commons CLI and Commons Validator, which are described hereafter. **Triangle** is a simple program that determines the type of a triangle from the length of its sides. Although it is small, it is a widely studied benchmark in the literature [Jia and Harman, 2009]. We find two complementary versions of Triangle, implemented in Java, at GitHub[1]. We then merged them to construct our own stable

[1]https://github.com/david-schuler/javalanche/tree/master/examples/triangle/src/triangle and https://github.com/hcoles/triangle-example

version of it. **Monopoli**[2] is the domain logic of a system to play the famous Monopoly game. It has also been present in important mutation testing studies [Mateo and Usaola, 2012b; Mateo et al., 2013; Harman et al., 2014]. Differently from those first systems, the next two are "real world" projects, maintained by Apache Software Foundation © and they have also been present in the related literature [Harman et al., 2014; Madeyski and Radyk, 2010]. **Commons CLI**[3] is a command-line arguments parser and **Commons Validator**[4] is a framework to define validators and validation rules in XML files.

Table 5.1 depicts some collected attributes from the four subject systems. In the second column, we informed "n/a" for the systems which *Version* information was not available. In the third column, the word *Main* means the Java source code of the system, excluding tests and other kinds of files. The column *# Test cases* means the number of original test cases recognized by JUnit. *JUnit version* is the version of JUnit that the test classes were implemented. For instance, Commons Validator has one class implemented for JUnit 4 and the remaining classes for JUnit 3. The last column shows the time spent by JUnit for running the system's original test suite. It is worth to mention that a single Monopoli test case fails, whilst the other three systems have no failures caught by their test suite.

Table 5.1. Subject systems.

System	*Version*	*LOC main*	*LOC tests*	*# Test cases*	*JUnit version*	*JUnit time (ms)*
Triangle	n/a	39	77	12	4	0.013
Monopoli	n/a	1,181	3,094	124	3	0.1
Commons CLI	1.4	2,699	3,932	318	4	0.1
Commons Validator	1.6	7,409	8,352	536	3 and 4	˜3,000

5.1.2 Mutants Generation, Code Customization and Test cases Removal for VarexJ

We configured MutVariants for generating all possible mutants that AOR, ROR and LCR mutation operators can create, except in loop conditional expressions (places with high probability to get an infinite loop). MutVariants, its functionalities, and

[2]http://mutationandcombinatorialtesting.blogspot.com/2012/01/systems-under-test.html
[3]http://commons.apache.org/proper/commons-cli/download_cli.cgi
[4]https://commons.apache.org/proper/commons-validator/download_validator.cgi

mutation operators were described in Chapter 4. For each subject system, we first generated mutants with MutVariants and then executed their test cases with VarexJ.

The number of mutants generated for each subject system is shown in the second column of Table 5.2. Comparing such information with systems LOC (Table 5.1), it can be seen that, proportionally, Triangle is the system with the highest density of mutants generated per line of code. As expected, the number of mutants increases with the systems LOC, but not linearly, due to the particularities of each system which belongs to four distinct domains.

Table 5.2. Mutants generated, covered, and test cases executed.

System	*Mutants generated*	*Mutants covered*	*Remaining test cases*
Triangle	128	128 (100%)	12 (100%)
Monopoli	408	408 (100%)	37 (30%)
Commons CLI	485	480 (99%)	48 (15%)
Commons Validator	1,989	1016 (51%)	161 (30%)

When we started to work with the largest subject systems, we faced some issues that did not happen with the smallest one. Therefore, some customizations had to be performed, which are explained along this and next paragraphs. For instance, while running Monopoli test cases, VarexJ have the execution aborted always when the GUI class `java.awt.Color` is used. More specifically, when `Color` variables attribution statements were executed. As a workaround, we manually replaced all occurrences of such class with an *ad-hoc* `enum` of the same name and with the same values for the colors. Since Monopoli source code is just the domain logic of a game, without GUIs, such change did not cause any impact.

In this phase of our study, we decided to not handle infinite loops in mutation-aware execution, since simply configuring a timeout in VarexJ would not make it provide any log information concerning which mutants lead to a long execution. We also attempt to configure JUnit timeouts, but without success. Therefore, in order to proceed with our viability assessment, we manually removed all test cases of the three largest subject systems that seemed to reach an infinite loop with mutants generated.

Lastly, while running the test cases of the three largest projects, VarexJ presented some complex issues that also caused execution abortion. To address these issues, we also had to remove all test cases for which such issues occurred. In Section 5.3, we discuss limitations in more detail. The number of test cases left (due to infinite loops and VarexJ issues) is depicted in the fourth column of Table 5.2. For the three largest systems, there can be seen a considerable reduction. Since fewer test cases were

executed, their execution covered fewer mutants than the originally generated ones, which number is depicted in the third column. Besides the absolute numbers, the third and fourth columns of Table 5.2 show, in parenthesis, the percentage of mutants covered in relation to the originally generated, and the number of test cases executed in relation to the number of test cases originally available, respectively.

5.1.3 Mutation Testing Techniques for Comparison

In order to perform the viability assessment of mutation-aware execution, it is important to compare it against other technique. For a fair evaluation, it is required to ensure exactly the same mutants generated for all techniques. We could look for something available in the literature or reproduce a feasible and easy to implement the technique. That is, we did not find a tool or how to configure any in such a way we could guarantee the same mutants. Five tools were presented in Chapter 4. For instance, PIT and JavaLanche mutation systems generate all possible mutants, but with more and distinct mutation operators; MuClipse does not work in source code written in recent Java versions.

Therefore, as a worthy assessment starting point, we decided to implement two baselines, consisting on simple (brute-force) approaches for mutation testing, without providing any enhancements proposed in the literature for them, which would demand considerable development effort. One is the *classical mutation testing process*. Since mutation-aware execution brings a technique and a related tool from a distinct software testing context, we wondered that running time behavior comparison of it and the classical approach might give us important findings for verifying whether our proposed technique could be more efficient than the most costly mutation testing approach. If so, how much more efficient it is or could be. If not, helping us to find out what would be necessary to fix and improve it to go further and be able to compare it against more efficient approaches. The implementation of the classical mutation testing for this research is described later, in this same subsection.

The other technique we implemented is *mutant schemata* [Untch et al., 1993], already explained in Chapter 4. This technique was proposed for mutation testing speedup in both compiling generated mutants and execution of test suite phases. In other words, it is a well-known technique for enhancement of mutation testing efficiency. Besides that, we can verify, once more, that mutant schemata is more efficient than the classical mutation testing. Our implementation of the mutant schemata technique is described later in this subsection and does not encompass the improvements proposed by Mateo and Usaola [2012b]. As explained in the previous paragraph, the comparison

with the mutant schemata original technique can lead us to important findings on how our proposed technique can be explored or evolved until it can be compared with some of the state-of-the-art techniques.

Classical Mutation Testing Process Implementation

For the classical mutation technique, we implemented (i) mutants generation, (ii) compilation and (iii) test cases execution, with related running time measuring. For (i), each mutant is generated as a new clone of the original project, containing a single syntactical change. As mentioned before, we need the same mutants generated for our proposed technique. To achieve this, we implemented an *ad-hoc* change in our MutVariants tool (presented in Chapter 4).

During the AST traversal, at each mutable place in the original source code, this version of MutVariants introduces the subject syntactical change, clones the current mutated compilation unit and saves it as a `.java` file in a specific directory. After that, the tool realizes an "undo" in the syntactical change and resumes the AST traversal looking for next mutation place, where the process repeats. Then, (ii) and (iii) are performed via shell scripts.

Mutant Schemata Implementation

The main concept of mutant schemata was presented in Chapter 4. Therefore, we could reuse the *metamutant* system generated for VarexJ in our proposed technique, as the mutants artifact. We implemented the mutant schemata technique as follows, for each subject system.

1. all mutants are initially disabled (i.e., their correspondent *boolean* variables are set to *false*);

2. for each mutant,

   2.1. enable the current mutant via Java Reflection (*true* value), while all others have to be disabled (*false*);

   2.2. run all test cases via JUnit;

3. retrieve the overall running time to compare with the other techniques.

5.2 Results

The last three columns of Table 5.3 depict the measured running time results for the three compared mutation testing techniques. As described in Section 5.1, it was necessary to remove test cases from subject systems in order to perform our proposed assessment and, due to that, less mutants than the overall generated ones were covered (see Table 5.2) by the remaining test cases. As expected, the classical implementation of the mutation process presented actually the highest running time values, leading to the worst results, in comparison with the other two techniques. Loading and execution time for each mutated system contributes to these results. In addition, it can be seen that running time increases with the number of LOC, mutants generated and test cases of the systems.

Table 5.3. Test cases running time for each implemented technique.

System	*Running time (ms) with mutants*		
	Classical	*Schemata*	*VarexJ*
Triangle	20,103	271	11,874
Monopoli	82,278	1,501	2,581
Commons CLI	107,454	1,483	2,355
Commons Validator	803,054	15,647	47,973

For the mutant schemata technique, the execution time is considerably inferior to classical mutation testing. On the one hand, it presents basically the same value for the two middle-sized systems, but raised in comparison with Triangle. Although the number of test cases and mutants in Commons CLI is higher than in Monopoli, its remaining test cases can be considered simpler and with generated mutants. On the other hand, there can be seen a considerable raise from middle systems to Commons Validator. This result can be explained by the difference of running time in Table 5.1, when they were executed without mutants. It may also require more memory for test suite completion.

Finally, and most important, the last column of Table 5.3 shows the running time related to our proposed technique. We could observe four main findings. (1) It is more efficient than the classical mutation testing for all four systems. (2) The running time of Monopoli and Commons CLI is quite similar, but it was already expected, given the results found and explanations we made for the schemata technique. (3) We can see unexpected running time comparing the middle systems with Triangle, the smallest one. Although the number of mutants in Triangle is rather inferior, all 128 mutants were generated in the unique method, having a high density of mutants per

LOC. Besides, this higher density of mutants in Triangle leads to almost no mutant free instruction and the main variable (`trian`) carrying a high number of mutant-dependent values. (4) For all of the subject systems, the running time of our proposed technique was superior to that of the mutant schemata. However, we can observe that the impact of the overhead carried by VarexJ can decrease, since the running time of VarexJ was 43 times higher than that of mutant schemata for Triangle and *decreases to only three times for Commons Validator*. The running time for middle-sized systems, once more, apparently was affected by the drastic reduction of the number of test cases, since the ratio obtained was about 1.6 times.

5.2.1 Answering RQs

This section aims to answer the two research questions (**RQ1** and **RQ2**) posted in Section 5.1.

RQ1: In which conditions is mutation-aware execution more efficient than other techniques in terms of running-time?

Taking into account just first order mutants (FOMs) in our study, we did not find a situation in which mutation-aware execution was more efficient than mutant schemata. Considering the inherent overhead of VarexJ, we can conclude that mutation-aware execution is not recommended for small and medium-sized systems, inferior to 8KLOC. A novel study with larger systems is necessary to investigate further this subject. We discuss higher order mutants in Subsection 5.2.2.

RQ2: Does mutation-aware execution scale, in terms of **(a)** the size of a system, **(b)** the number of test cases, and **(c)** the number of generated mutants?

We could verify in this study that the proposed technique execution seems to scale, ranging from an apparent exponential to a smoother growth. While the number of LOC, test cases, and generated mutants increase, the impact of the VarexJ overhead decreases. Nevertheless, a direct correlation can not be seen. The proportion of generated mutants caused the higher impact in VarexJ execution, as could be verified by comparing the results of the first three systems in Table 5.3. Although this fact happened on small systems, while analyzing source code with generated mutants, it is understood that VarexJ took more time processing statements and variable assignments in Triangle, due to the quantity of mutants influencing them than in Monopoli and Commons CLI. Therefore, it is worth to design another study to investigate the impact of varying the distribution of the same quantity of mutants on a given source code.

It is worth to mention a situation we faced while we were removing test cases

of Commons Validator (the largest subject system). There actually are more than those 161 test cases depicted in Table 5.2. We found this by running each test class apart from the other. However, when running all of them together, VarexJ execution was aborted due to an out of memory issue. Before concluding that mutation-aware execution does not scale, further investigation on VarexJ needs to be performed.

5.2.2 Discussion

Mutation-aware execution does much more than only First Order Mutant (FOM) analysis. Variability-aware execution, performed via VarexJ, provides information enough to investigate feature interactions that, bringing to our context, are interactions of mutants, i.e. more than one mutant directly influencing statement executions and variable values. Such interaction of mutants led us to reason about higher order mutants (HOMs) [Jia and Harman, 2009], already introduced in Section 2.2.

In HOM context, mutation-aware execution can be useful twofold. (i) Finding more representative HOMs from VarexJ log, which provides information about all combinations of mutants that cause tests failures. Indeed, there is a preliminary study that we are collaborators [Chen, 2018]. (ii) Reporting alive HOMs during a mutation testing process, functionality not implemented by any available tool as well as was not in this dissertation.

For instance, to analyze second order mutants with mutant schemata technique, it is necessary to run the test suite for each combination of two enabled mutants, which leads to a rise in running time from $O(n)$ to $O(n^2)$. Therefore, if one desires to investigate HOMs, mutation-aware execution might be recommended.

Mutation-aware execution may have great potential to be more efficient than other techniques for handling infinite loops in mutation testing, since it performs a single execution. When a mutant causes an infinite loop, the mutant is considered killed. One technique for supporting that is configuring in test cases a timeout to be handled by JUnit. Suppose that 100 mutants cause an infinite loop when tested and a timeout of two seconds was configured to each test case. In theory, test suite running time will raise in, at least, 200 seconds.

However, JUnit also takes a huge overhead on dealing with preconfigured timeouts, since that functionality "is implemented by running the test method in a separate thread"[5]. We could also confirm that while running the classical mutation testing implementation for the subject system Commons CLI. Such implementation works in software testing, where the number of infinite loops caused by programmers failures is

[5]https://github.com/junit-team/junit4/wiki/timeout-for-tests

expected to be quite small than that caused by a high amount of mutants. Therefore, this fact may encourage researchers to go further on discovering a technique to identify infinite loops in variability-aware execution and, more remarkably, identify which configuration causes them.

5.3 Limitations and Threats to Validity

We faced some **limitations** while we evolved this research. For instance, the VarexJ version used for generating the results of this study (from August 2, 2018) presented some issues. The most relevant one was that, as mentioned in Section 5.1, some test cases had to be removed in order to proceed with our viability assessment. Before that, due to some bug reporting we made to VarexJ main developers, previous versions of VarexJ was fixed in July 2018.

Unfortunately, due to the amount of new failures found, we decided to remove test cases besides reporting those issues, in order to proceed with our viability assessment of mutation-aware execution. Except for Triangle system, running the test cases combined with generated mutants, led to execution abortion due to many native method invocations, among other failures. It is worth to mention that the abortion of VarexJ does not always happen on all invocations of those natives. It depends on the variability context caused by many mutants influencing the execution. Some examples of failures include `String.hashCode`, `String.indexOf`, `toCharArray`, stack overflow, out of memory, unexpected calling of `getValue` on Choice objects [Meinicke, 2014].

The implementation of mutation-aware execution done for this dissertation may lead to **threats to the validity** of the achieved results. Test cases removal, due both to VarexJ issues and infinite loops situations, caused many mutants not being executed, i.e., code where they were generated was not covered by remaining test cases. In addition, the remaining test cases might not be enough to represent a real system test suite, since parts of the source code are no longer tested and generated mutants not reached. The remaining test cases could be considered less representative, leading us to inconsistent conclusions. However, we yet could analyze many situations we faced and were able to justify all of the measured data presented in Table 5.3 and also to identify scenarios for future work.

MutVariants implements three mutation operators (Section 4.2). One may point they are not sufficient to generalize a mutation testing study. However, such operators are three out of five more representative ones, according to the literature [Offutt et al., 1996] and we configured MutVariants to generate all possible mutants in each subject

system. The fourth operator, Absolute Value Insertion (ABS), which inserts absolute values into variable expressions, was reported predominantly not useful in some contexts [Petrovic and Ivankovic, 2018]. Concerning the last operator, Unary Operator Insertion (UOI), its original definition is quite broad (could lead to an impracticable number of generated mutants, given the nature of our study), has many restrictions, and there is a lack of examples of it in the literature. Therefore, we could not ensure the correctness of any implementation for UOI, before performing a more refined review in the literature.

As mentioned in Subsection 5.2.1, studies with more systems and more flexibility in varying the number of generated mutants. More systems could be included in the study, as well as other evaluations varying the quantity, places and types of mutants. Nevertheless, doing that would require redoing the test cases removal, which would become unfeasible.

5.4 Conclusion

In this chapter, we evaluated the viability of mutation-aware execution, a technique we proposed in Chapter 3. To accomplish this, we performed an empirical study setup, for which we choose four subject systems, implemented two other mutation techniques and ensure the same setup. Some limitations made us unable to affirm that our proposed technique scales and is more efficient than our implementation of the mutant schemata technique. However, the results and findings we discussed can indeed encourage researchers to investigate further mutation-aware execution, mainly for the context of higher order mutation testing. The next chapter will present the related work we found in the literature.

Chapter 6

Related Work

To the best of our knowledge, sharing execution and aggregated results (concepts implemented in variability-aware execution - see Section 2.3) are brought to mutation testing context for the first time in this dissertation and in a preliminary work that we collaborated [Chen, 2018]. Chen proposes a novel approach for finding strongly subsuming higher order mutants (SSHOMs) [Jia and Harman, 2009], from VarexJ log expressions, and validated it with the Triangle system. In addition, she compares such approach performance with that of evolutionary algorithms for finding SSHOMs. Therefore, that work is not concerned with the efficiency and the scalability of the running time phase of the mutation testing process, as this work is. In the remainder of this section, we presented some works that performed and achieved something in common with parts of our proposed technique and assessment study.

Cost reduction techniques for mutation testing were classified into three categories by Offutt and Untch [2001]: *do faster*, *do smarter* and *do faster*. A technique may not necessarily be classified into one single category (may combine multiple techniques) and, furthermore, recent techniques are hard to categorize [Ferrari et al., 2018]. Both happen with mutation-aware execution, the technique proposed in this dissertation.

Do fewer is related to reducing the number of generated (and executed) mutants in mutation testing process, without significant impact in mutation score. Our proposed technique is not related to this category, since we just used the findings of a previous study [Offutt et al., 1996] for generating mutants with three out of five more representative mutation operators. Mutation testing at Google basically uses the operators defined by the same study.

Do faster is related to generating and running mutants as quickly as possible. Mutant schemata, a compiling and running time speedup technique, in which all mutants are generated into a single program (*metamutant*) [Untch et al., 1993; Madeyski and

Radyk, 2010], is classified into this category. Mateo and Usaola propose *metamutant* generation in the source code in a quite similar way that is required by mutation-aware execution, our proposed technique. We also implemented mutant schemata execution in the phase of our study (see Section 5.1), but without any improvements proposed by such work and distinct way for mutants activation (enabling/disabling).

According to Offutt and Untch [2001], "the *do smarter* approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution". Wang et al. [2017] presented "AccMut", a framework that aims to reduce redundant execution, improving split-stream via equivalence modulo state detection. Split-stream execution consists of creating a parallel process when each mutant is reached. Equivalence modulo state ensures that a new parallel process will be created only if, given the current stated of the program, the execution of different statements lead to different states. AccMut performs mutation testing in C programs. This framework does not implement merging processes, which could decrease redundant executions. If it is feasible, it might require considerable implementation effort.

Our proposed technique relies on variability-aware execution, which eliminates redundant executions via sharing mechanisms. Aggregated results ensure keeping different states of a program while a single execution is performed. Specifically, VarexJ supports *joining*, as can be seen in Figure 3.2.

Chapter 7

Conclusion and Future Work

Mutation testing is a technique to assess the quality of software tests. Although this research theme has been studied in literature over last four decades, it did not reach considerable popularity in the industry, due to the expensive nature of some steps of the mutation testing process and a lack of up-to-date tools and techniques for automating them. In this dissertation, we propose a technique that aims at reducing the computational effort in the test cases execution step. In Section 7.1, we presented final remarks about our work. Finally, in Section 7.2, we discuss future work.

7.1 Final Remarks

The technique we proposed was inspired in variability-aware execution, a technique developed for configurable systems, on which it is possible to execute all configurations simultaneously by sharing executions with an internal representation of variability [Thüm et al., 2012; Meinicke et al., 2016]. We named our technique mutation-aware execution, since it brings the concept of variability-aware execution to the context of mutation tests, where the mutants are handled as variability. We also used VarexJ, a Java bytecode interpreter that can be considered a JVM with variability-aware execution [Meinicke, 2014].

For generating mutants, we first developed a mutant generator tool, named MutVariants (Chapter 4), which implements three out of the most representative mutation operators known in the literature (about 100 mutation operators were proposed at all). We decided to implement MutVariants because it was necessary for VarexJ to recognize the generated mutants in the same way it recognizes features as variability.

To evaluate the viability of mutation-aware execution, we choose four subject systems to generate mutants and implemented two baseline mutation testing techniques

to compare the test cases execution time. Our goal was to answer two research questions concerning efficiency and scalability of our proposed technique. Given the incipient nature of our study, we faced situations that limited our study. VarexJ presented some issues on running several test cases considering variability seeded on systems (mutants). Furthermore, we discovered the lack of handling infinite loops in mutation-aware execution. Therefore, we were compelled to remove test cases of the three largest subject systems until test case execution became feasible.

Despite those limitations, since mutation-aware execution was not more efficient than the baseline mutant schemata we implemented, we could perceive that the VarexJ overhead is smoothed when the size of the evaluated system increases. For the smallest system, out technique last over 43 times than mutant schemata and, for the largest system, only three times. We have little information to affirm the following, but we wonder that mutation-aware execution can be more efficient and scale when running larger systems, since the limitations are properly resolved. In addition, mutation-aware execution does much more than simply analyzing first order mutants and can be recommended for handling higher order mutants.

7.2 Future Work

We discussed a significant limitation of the current state of our proposed technique: the lack for handling infinite loops. As future work, it is expected to make mutation-aware execution able to identify mutants (or combination of them) that leads the execution to an infinite loop, handle this properly and resume the execution by joining statement executions.

After handling infinite loops and other limitations of VarexJ implementation, it is imperative to replicate this study with complete test cases of the subject systems as well as evaluating larger systems and state-of-art mutation techniques, in order to be able to answer clearly the research questions we proposed.

Our mutant generator tool implemented three mutation operators so far. When studying the behavior of many mutation operators, we could observe a lack of consensus in the literature, in terms of naming, scope, applicability, effectiveness, usefulness, etc. For instance, unary operator insertion (UOI) is quite complex in its definition and quite confusing when its usage is reported. In other works, Java operators are replaced only by the opposite one (e.g., `+` by `-` and `<` by `>=`) and the mutation operators are given other names. Therefore, we identify the need for a literature review in order to map all of the characteristics them, from studies reporting their definition and/or their usage.

As discussed in Subsection 5.2.2, it is worth an implementation for identifying higher order mutants not killed after a mutation-aware execution. It can be based on the approach presented by Chen [2018] or can be another approach that seeks for HOMs of specific small orders (second or third, for instance), which is already in progress.

Finally, the development of a complete mutation system that (i) automates all possible steps of mutation testing and (ii) implements mutation-aware execution. (iii) It may implement more, but sufficient mutation operators. (iv) It also may recognize as many equivalent mutants as possible and provide (v) an interface to support the user to identify the remaining ones. Besides that, providing (vi) a complete report containing mutation score, killed and live mutants, which could be even first and higher order mutants. Finally, (vii) mechanisms to locate reported mutants and ease the creation of new test cases to kill live ones. Items (ii) and (vi) are directly related to our study.

Bibliography

Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Banzi, A. S., Nobre, T., Pinheiro, G. B., ÁRias, J. C. G., Pozo, A., and Vergilio, S. R. (2012). Selecting mutation operators with a multiobjective approach. *Expert Systems with Applications*, 39(15):12131--12142.

Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113--136.

Budd, T. A. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31--45.

Chen, S. (2018). Finding higher order mutants using variational execution. Technical report 1809.04563.arXiv. Accepted to SPLASH'18 student research competition.

Coles, H., Laurent, T., Henard, C., Papadakis, M., and Ventresque, A. (2016). Pit: a practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449--452. ACM.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34--41.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859--866.

Ferrari, F. C., Pizzoleto, A. V., and Offutt, A. J. (2018). A systematic review of cost reduction techniques for mutation testing: Preliminary results. In *Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), 2018*, pages 1--10. IEEE.

Ferreira, J. M., Vergilio, S. R., and Quinaia, M. (2017). Software product line testing based on feature model mutation. *International Journal of Software Engineering and Knowledge Engineering*, 27(05):817--839.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns*, volume 1. Addison-Wesley.

Gopinath, R., Alipour, M. A., Ahmed, I., Jensen, C., and Groce, A. (2016). On the limits of mutation reduction strategies. In *Proceedings of the 38th international conference on software engineering*, pages 511--522. ACM.

Harman, M., Jia, Y., Reales Mateo, P., and Polo, M. (2014). Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 397--408. ACM.

Havelund, K. and Pressburger, T. (2000). Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366--381.

Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371--379.

Jia, Y. and Harman, M. (2009). Higher order mutation testing. *Information and Software Technology*, 51(10):1379--1393.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649--678.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654--665. ACM.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327--354. Springer.

Kim, S.-W., Ma, Y.-S., and Kwon, Y.-R. (2013). Combining weak and strong mutation for a noninterpretive java mutation system. *Software Testing, Verification and Reliability*, 23(8):647--668.

King, K. N. and Offutt, A. J. (1991). A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685--718.

Lima, J. A. P. and Vergilio, S. R. (2018). Search-based higher order mutation testing: A mapping study. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, pages 87--96. ACM.

Ma, Y.-S., Kwon, Y.-R., and Offutt, J. (2002). Inter-class mutation operators for Java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 352--363. IEEE.

Ma, Y.-S., Offutt, J., and Kwon, Y. R. (2005). MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97--133.

Madeyski, L. and Radyk, N. (2010). Judy–a mutation testing tool for Java. *IET software*, 4(1):32--42.

Mateo, P. R. and Usaola, M. P. (2012a). Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 646--649. IEEE.

Mateo, P. R. and Usaola, M. P. (2012b). Mutant execution cost reduction: Through MUSIC (mutant schema improved with extra code). In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 664--672. IEEE.

Mateo, P. R., Usaola, M. P., and Aleman, J. L. F. (2013). Validating second-order mutation at system level. *IEEE Transactions on Software Engineering*, 39(4):570--587.

Meinicke, J. (2014). VarexJ: A variability-aware interpreter for Java applications. *Master's thesis, University of Magdeburg.*

Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., and Saake, G. (2016). On essential configuration complexity: measuring interactions in highly-configurable systems. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 483--494. IEEE.

Nhlabatsi, A., Laney, R., and Nuseibeh, B. (2008). Feature interaction: The security threat from within software systems. *Progress in Informatics*, 5:75--89.

Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5--20.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99--118.

Offutt, A. J. and Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165--192.

Offutt, A. J. and Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34--44. Springer.

Petrovic, G. and Ivankovic, M. (2018). State of mutation testing at google. In *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)*.

Pohl, K., Böckle, G., and van Der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

Pressman, R. S. (2009). *Software engineering: a practitioner's approach*. McGraw-Hill Education.

Schuler, D. and Zeller, A. (2009). Javalanche: efficient mutation testing for Java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297--298. ACM.

Siami Namin, A., Andrews, J. H., and Murdoch, D. J. (2008). Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, pages 351--360. ACM.

Sommerville, I. (2015). *Software Engineering*. Pearson.

Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-based deductive verification of software product lines. In *ACM SIGPLAN Notices*, volume 48, pages 11--20. ACM.

Untch, R. H., Offutt, A. J., and Harrold, M. J. (1993). Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139--148. ACM.

Usaola, M. P. and Mateo, P. R. (2010). Mutation testing cost reduction techniques: a survey. *IEEE software*, 27(3):80--86.

Wang, B., Xiong, Y., Shi, Y., Zhang, L., and Hao, D. (2017). Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 295--306. ACM.

Woodward, M. and Halewood, K. (1988). From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 152--158. IEEE.

Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366--427.

Attachment A

Triangle System

Listing A.1. TriangleType.java enum.

```
1  package triangle;
2
3  public enum TriangleType {
4          INVALID, SCALENE, EQUILATERAL, ISOSCELES
5  }
```

Listing A.2. Triangle.java original class.

```
1  package triangle;
2
3  import static triangle.TriangleType.*;
4
5  public class Triangle {
6
7          public static TriangleType classify(int a, int b, int c) {
8                  int trian;
9                  if (a <= 0 || b <= 0 || c <= 0)
10                         return INVALID;
11                 trian = 0;
12                 if (a == b)
13                         trian = trian + 1;
14                 if (a == c)
15                         trian = trian + 2;
16                 if (b == c)
17                         trian = trian + 3;
18                 if (trian == 0)
19                         if (a + b < c || a + c < b || b + c < a)
20                                 return INVALID;
21                         else
22                                 return SCALENE;
23                 if (trian > 3)
24                         return EQUILATERAL;
25                 if (trian == 1 && a + b > c)
26                         return ISOSCELES;
27                 else if (trian == 2 && a + c > b)
```

```
28                      return ISOSCELES;
29                  else if (trian == 3 && b + c > a)
30                      return ISOSCELES;
31                  return INVALID;
32          }
33
34  }
```

Listing A.3. Triangle.java class with 128 mutants.

```
1   package triangle;
2
3   import static triangle.TriangleType.*;
4   import gov.nasa.jpf.annotation.Conditional;
5
6   public class Triangle {
7
8       @Conditional
9       public static boolean _mut0 = false, _mut1 = false, _mut2 = false, _mut3 = false,
            _mut4 = false, _mut5 = false, _mut6 = false, _mut7 = false, _mut8 = false,
            _mut9 = false, _mut10 = false, _mut11 = false, _mut12 = false, _mut13 = false,
             _mut14 = false, _mut15 = false, _mut16 = false, _mut17 = false, _mut18 =
            false, _mut19 = false, _mut20 = false, _mut21 = false, _mut22 = false, _mut23
            = false, _mut24 = false, _mut25 = false, _mut26 = false, _mut27 = false,
            _mut28 = false, _mut29 = false, _mut30 = false, _mut31 = false, _mut32 = false
            , _mut33 = false, _mut34 = false, _mut35 = false, _mut36 = false, _mut37 =
            false, _mut38 = false, _mut39 = false, _mut40 = false, _mut41 = false, _mut42
            = false, _mut43 = false, _mut44 = false, _mut45 = false, _mut46 = false,
            _mut47 = false, _mut48 = false, _mut49 = false, _mut50 = false, _mut51 = false
            , _mut52 = false, _mut53 = false, _mut54 = false, _mut55 = false, _mut56 =
            false, _mut57 = false, _mut58 = false, _mut59 = false, _mut60 = false, _mut61
            = false, _mut62 = false, _mut63 = false, _mut64 = false, _mut65 = false,
            _mut66 = false, _mut67 = false, _mut68 = false, _mut69 = false, _mut70 = false
            , _mut71 = false, _mut72 = false, _mut73 = false, _mut74 = false, _mut75 =
            false, _mut76 = false, _mut77 = false, _mut78 = false, _mut79 = false, _mut80
            = false, _mut81 = false, _mut82 = false, _mut83 = false, _mut84 = false,
            _mut85 = false, _mut86 = false, _mut87 = false, _mut88 = false, _mut89 = false
            , _mut90 = false, _mut91 = false, _mut92 = false, _mut93 = false, _mut94 =
            false, _mut95 = false, _mut96 = false, _mut97 = false, _mut98 = false, _mut99
            = false, _mut100 = false, _mut101 = false, _mut102 = false, _mut103 = false,
            _mut104 = false, _mut105 = false, _mut106 = false, _mut107 = false, _mut108 =
            false, _mut109 = false, _mut110 = false, _mut111 = false, _mut112 = false,
            _mut113 = false, _mut114 = false, _mut115 = false, _mut116 = false, _mut117 =
            false, _mut118 = false, _mut119 = false, _mut120 = false, _mut121 = false,
            _mut122 = false, _mut123 = false, _mut124 = false, _mut125 = false, _mut126 =
            false, _mut127 = false;
10
11      public static TriangleType classify(int a, int b, int c) {
12          int trian;
13          if ((_mut16 ? ((_mut10 ? ((_mut4 ? (a >= 0) : (_mut3 ? (a > 0) : (_mut2 ? (a <
                0) : (_mut1 ? (a != 0) : (_mut0 ? (a == 0) : (a <= 0)))))) && (_mut9 ? (b
               >= 0) : (_mut8 ? (b > 0) : (_mut7 ? (b < 0) : (_mut6 ? (b != 0) : (_mut5
              ? (b == 0) : (b <= 0))))))) : ((_mut4 ? (a >= 0) : (_mut3 ? (a > 0) : (
              _mut2 ? (a < 0) : (_mut1 ? (a != 0) : (_mut0 ? (a == 0) : (a <= 0)))))) ||
               (_mut9 ? (b >= 0) : (_mut8 ? (b > 0) : (_mut7 ? (b < 0) : (_mut6 ? (b !=
              0) : (_mut5 ? (b == 0) : (b <= 0))))))) && (_mut15 ? (c >= 0) : (_mut14 ?
```

```
            (c > 0) : (_mut13 ? (c < 0) : (_mut12 ? (c != 0) : (_mut11 ? (c == 0) : (
        c <= 0)))))) : ((_mut10 ? ((_mut4 ? (a >= 0) : (_mut3 ? (a > 0) : (_mut2
        ? (a < 0) : (_mut1 ? (a != 0) : (_mut0 ? (a == 0) : (a <= 0)))))) && (
        _mut9 ? (b >= 0) : (_mut8 ? (b > 0) : (_mut7 ? (b < 0) : (_mut6 ? (b != 0)
            : (_mut5 ? (b == 0) : (b <= 0))))))) : ((_mut4 ? (a >= 0) : (_mut3 ? (a >
            0) : (_mut2 ? (a < 0) : (_mut1 ? (a != 0) : (_mut0 ? (a == 0) : (a <= 0))
        )))) || (_mut9 ? (b >= 0) : (_mut8 ? (b > 0) : (_mut7 ? (b < 0) : (_mut6 ?
            (b != 0) : (_mut5 ? (b == 0) : (b <= 0)))))))) || (_mut15 ? (c >= 0) : (
        _mut14 ? (c > 0) : (_mut13 ? (c < 0) : (_mut12 ? (c != 0) : (_mut11 ? (c
        == 0) : (c <= 0)))))))))
14          return INVALID;
15      trian = 0;
16      if ((_mut21 ? (a >= b) : (_mut20 ? (a <= b) : (_mut19 ? (a > b) : (_mut18 ? (a
            < b) : (_mut17 ? (a != b) : (a == b)))))))
17          trian = (_mut25 ? (trian % 1) : (_mut24 ? (trian / 1) : (_mut23 ? (trian *
                1) : (_mut22 ? (trian - 1) : (trian + 1)))));
18      if ((_mut30 ? (a >= c) : (_mut29 ? (a <= c) : (_mut28 ? (a > c) : (_mut27 ? (a
            < c) : (_mut26 ? (a != c) : (a == c)))))))
19          trian = (_mut34 ? (trian % 2) : (_mut33 ? (trian / 2) : (_mut32 ? (trian *
                2) : (_mut31 ? (trian - 2) : (trian + 2)))));
20      if ((_mut39 ? (b >= c) : (_mut38 ? (b <= c) : (_mut37 ? (b > c) : (_mut36 ? (b
            < c) : (_mut35 ? (b != c) : (b == c)))))))
21          trian = (_mut43 ? (trian % 3) : (_mut42 ? (trian / 3) : (_mut41 ? (trian *
                3) : (_mut40 ? (trian - 3) : (trian + 3)))));
22      if ((_mut48 ? (trian >= 0) : (_mut47 ? (trian <= 0) : (_mut46 ? (trian > 0) :
            (_mut45 ? (trian < 0) : (_mut44 ? (trian != 0) : (trian == 0)))))))
23          if ((_mut77 ? ((_mut67 ? ((_mut57 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b)
                : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) >= c) : (_mut56
                ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (
                _mut49 ? (a - b) : (a + b))))) <= c) : (_mut55 ? ((_mut52 ? (a % b) :
                (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))
                ) > c) : (_mut54 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ?
                (a * b) : (_mut49 ? (a - b) : (a + b))))) != c) : (_mut53 ? ((_mut52 ?
                 (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) :
                 (a + b))))) == c) : ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50
                ? (a * b) : (_mut49 ? (a - b) : (a + b))))) < c)))))) && (_mut66 ? ((
                _mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (
                a - c) : (a + c))))) >= b) : (_mut65 ? ((_mut61 ? (a % c) : (_mut60 ?
                (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) <= b) :
                (_mut64 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) :
                 (_mut58 ? (a - c) : (a + c))))) > b) : (_mut63 ? ((_mut61 ? (a % c) :
                 (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c)))
                )) != b) : (_mut62 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59
                ? (a * c) : (_mut58 ? (a - c) : (a + c))))) == b) : ((_mut61 ? (a % c)
                 : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c)
                ))))) < b)))))) : ((_mut57 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) :
                (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) >= c) : (_mut56 ?
                ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ?
                 (a - b) : (a + b))))) <= c) : (_mut55 ? ((_mut52 ? (a % b) : (_mut51
                ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) > c) :
                 (_mut54 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b)
                : (_mut49 ? (a - b) : (a + b))))) != c) : (_mut53 ? ((_mut52 ? (a % b)
                 : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b)
                )))) == c) : ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b
                ) : (_mut49 ? (a - b) : (a + b))))) < c)))))) || (_mut66 ? ((_mut61 ?
```

```
(a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) :
(a + c))))) >= b) : (_mut65 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) :
 (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) <= b) : (_mut64 ?
 ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58
? (a - c) : (a + c))))) > b) : (_mut63 ? ((_mut61 ? (a % c) : (_mut60
? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) != b)
: (_mut62 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c)
 : (_mut58 ? (a - c) : (a + c))))) == b) : ((_mut61 ? (a % c) : (
_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c)))))
 < b)))))))) && (_mut76 ? ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (
_mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c))))) >= a) : (_mut75 ?
((_mut71 ? (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c) : (_mut68 ?
 (b - c) : (b + c))))) <= a) : (_mut74 ? ((_mut71 ? (b % c) : (_mut70
? (b / c) : (_mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c))))) > a) :
 (_mut73 ? ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c)
: (_mut68 ? (b - c) : (b + c))))) != a) : (_mut72 ? ((_mut71 ? (b % c)
 : (_mut70 ? (b / c) : (_mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c)
)))) == a) : ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c
) : (_mut68 ? (b - c) : (b + c))))) < a)))))))) : ((_mut67 ? ((_mut57 ?
 ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49
? (a - b) : (a + b))))) >= c) : (_mut56 ? ((_mut52 ? (a % b) : (_mut51
 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) <= c)
 : (_mut55 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b
) : (_mut49 ? (a - b) : (a + b))))) > c) : (_mut54 ? ((_mut52 ? (a % b
) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b
)))))) != c) : (_mut53 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (
_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) == c) : ((_mut52 ?
(a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) :
(a + b))))) < c)))))) && (_mut66 ? ((_mut61 ? (a % c) : (_mut60 ? (a /
 c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) >= b) : (
_mut65 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) :
(_mut58 ? (a - c) : (a + c))))) <= b) : (_mut64 ? ((_mut61 ? (a % c) :
 (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c)))
)) > b) : (_mut63 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ?
 (a * c) : (_mut58 ? (a - c) : (a + c))))) != b) : (_mut62 ? ((_mut61
? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c)
: (a + c))))) == b) : ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59
 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) < b)))))))) : ((_mut57 ?
((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ?
 (a - b) : (a + b))))) >= c) : (_mut56 ? ((_mut52 ? (a % b) : (_mut51
? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) <= c)
: (_mut55 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b)
 : (_mut49 ? (a - b) : (a + b))))) > c) : (_mut54 ? ((_mut52 ? (a % b)
 : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b)
)))) != c) : (_mut53 ? ((_mut52 ? (a % b) : (_mut51 ? (a / b) : (
_mut50 ? (a * b) : (_mut49 ? (a - b) : (a + b))))) == c) : ((_mut52 ?
(a % b) : (_mut51 ? (a / b) : (_mut50 ? (a * b) : (_mut49 ? (a - b) :
(a + b))))) < c)))))) || (_mut66 ? ((_mut61 ? (a % c) : (_mut60 ? (a /
 c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c))))) >= b) : (
_mut65 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) :
(_mut58 ? (a - c) : (a + c))))) <= b) : (_mut64 ? ((_mut61 ? (a % c) :
 (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c) : (a + c)))
)) > b) : (_mut63 ? ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59 ?
 (a * c) : (_mut58 ? (a - c) : (a + c))))) != b) : (_mut62 ? ((_mut61
? (a % c) : (_mut60 ? (a / c) : (_mut59 ? (a * c) : (_mut58 ? (a - c)
```

```
            : (a + c))))) == b) : ((_mut61 ? (a % c) : (_mut60 ? (a / c) : (_mut59
             ? (a * c) : (_mut58 ? (a - c) : (a + c))))) < b)))))))) || (_mut76 ?
            ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c) : (_mut68 ?
             (b - c) : (b + c))))) >= a) : (_mut75 ? ((_mut71 ? (b % c) : (_mut70
            ? (b / c) : (_mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c))))) <= a)
            : (_mut74 ? ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c)
             : (_mut68 ? (b - c) : (b + c))))) > a) : (_mut73 ? ((_mut71 ? (b % c)
             : (_mut70 ? (b / c) : (_mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c)
            )))) != a) : (_mut72 ? ((_mut71 ? (b % c) : (_mut70 ? (b / c) : (
            _mut69 ? (b * c) : (_mut68 ? (b - c) : (b + c))))) == a) : ((_mut71 ?
            (b % c) : (_mut70 ? (b / c) : (_mut69 ? (b * c) : (_mut68 ? (b - c) :
            (b + c))))) < a)))))))))
24              return INVALID;
25          else
26              return SCALENE;
27      if ((_mut82 ? (trian >= 3) : (_mut81 ? (trian <= 3) : (_mut80 ? (trian < 3) :
            (_mut79 ? (trian != 3) : (_mut78 ? (trian == 3) : (trian > 3)))))))
28          return EQUILATERAL;
29      if ((_mut97 ? ((_mut87 ? (trian >= 1) : (_mut86 ? (trian <= 1) : (_mut85 ? (
            trian > 1) : (_mut84 ? (trian < 1) : (_mut83 ? (trian != 1) : (trian == 1)
            ))))) || (_mut96 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a *
             b) : (_mut88 ? (a - b) : (a + b))))) >= c) : (_mut95 ? ((_mut91 ? (a % b)
             : (_mut90 ? (a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b)))))
             <= c) : (_mut94 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a *
             b) : (_mut88 ? (a - b) : (a + b))))) < c) : (_mut93 ? ((_mut91 ? (a % b)
            : (_mut90 ? (a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b)))))
            != c) : (_mut92 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a *
            b) : (_mut88 ? (a - b) : (a + b))))) == c) : ((_mut91 ? (a % b) : (_mut90
            ? (a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b))))) > c)))))))
             : ((_mut87 ? (trian >= 1) : (_mut86 ? (trian <= 1) : (_mut85 ? (trian >
            1) : (_mut84 ? (trian < 1) : (_mut83 ? (trian != 1) : (trian == 1)))))) &&
             (_mut96 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a * b) : (
            _mut88 ? (a - b) : (a + b))))) >= c) : (_mut95 ? ((_mut91 ? (a % b) : (
            _mut90 ? (a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b))))) <=
            c) : (_mut94 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a * b)
            : (_mut88 ? (a - b) : (a + b))))) < c) : (_mut93 ? ((_mut91 ? (a % b) : (
            _mut90 ? (a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b))))) !=
            c) : (_mut92 ? ((_mut91 ? (a % b) : (_mut90 ? (a / b) : (_mut89 ? (a * b)
            : (_mut88 ? (a - b) : (a + b))))) == c) : ((_mut91 ? (a % b) : (_mut90 ? (
            a / b) : (_mut89 ? (a * b) : (_mut88 ? (a - b) : (a + b))))) > c)))))))))
30          return ISOSCELES;
31      else if ((_mut112 ? ((_mut102 ? (trian >= 2) : (_mut101 ? (trian <= 2) : (
            _mut100 ? (trian > 2) : (_mut99 ? (trian < 2) : (_mut98 ? (trian != 2) : (
            trian == 2)))))) || (_mut111 ? ((_mut106 ? (a % c) : (_mut105 ? (a / c) :
            (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c))))) >= b) : (_mut110 ?
            ((_mut106 ? (a % c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) : (_mut103 ?
             (a - c) : (a + c))))) <= b) : (_mut109 ? ((_mut106 ? (a % c) : (_mut105 ?
             (a / c) : (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c))))) < b) : (
            _mut108 ? ((_mut106 ? (a % c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) :
            (_mut103 ? (a - c) : (a + c))))) != b) : (_mut107 ? ((_mut106 ? (a % c) :
            (_mut105 ? (a / c) : (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c)))))
             == b) : ((_mut106 ? (a % c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) : (
            _mut103 ? (a - c) : (a + c))))) > b)))))))) : ((_mut102 ? (trian >= 2) : (
            _mut101 ? (trian <= 2) : (_mut100 ? (trian > 2) : (_mut99 ? (trian < 2) :
            (_mut98 ? (trian != 2) : (trian == 2)))))) && (_mut111 ? ((_mut106 ? (a %
```

```
            c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c
            ))))) >= b) : (_mut110 ? ((_mut106 ? (a % c) : (_mut105 ? (a / c) : (
            _mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c))))) <= b) : (_mut109 ? ((
            _mut106 ? (a % c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) : (_mut103 ? (
            a - c) : (a + c))))) < b) : (_mut108 ? ((_mut106 ? (a % c) : (_mut105 ? (a
             / c) : (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c))))) != b) : (
            _mut107 ? ((_mut106 ? (a % c) : (_mut105 ? (a / c) : (_mut104 ? (a * c) :
            (_mut103 ? (a - c) : (a + c))))) == b) : ((_mut106 ? (a % c) : (_mut105 ?
            (a / c) : (_mut104 ? (a * c) : (_mut103 ? (a - c) : (a + c))))) > b)))))))
            ))
32          return ISOSCELES;
33      else if ((_mut127 ? ((_mut117 ? (trian >= 3) : (_mut116 ? (trian <= 3) : (
            _mut115 ? (trian > 3) : (_mut114 ? (trian < 3) : (_mut113 ? (trian != 3) :
             (trian == 3)))))) || (_mut126 ? ((_mut121 ? (b % c) : (_mut120 ? (b / c)
            : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c))))) >= a) : (_mut125 ?
             ((_mut121 ? (b % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) : (_mut118
            ? (b - c) : (b + c))))) <= a) : (_mut124 ? ((_mut121 ? (b % c) : (_mut120
            ? (b / c) : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c))))) < a) : (
            _mut123 ? ((_mut121 ? (b % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) :
            (_mut118 ? (b - c) : (b + c))))) != a) : (_mut122 ? ((_mut121 ? (b % c) :
            (_mut120 ? (b / c) : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c)))))
             == a) : ((_mut121 ? (b % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) : (
            _mut118 ? (b - c) : (b + c))))) > a)))))))) : ((_mut117 ? (trian >= 3) : (
            _mut116 ? (trian <= 3) : (_mut115 ? (trian > 3) : (_mut114 ? (trian < 3) :
             (_mut113 ? (trian != 3) : (trian == 3)))))) && (_mut126 ? ((_mut121 ? (b
            % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b +
             c))))) >= a) : (_mut125 ? ((_mut121 ? (b % c) : (_mut120 ? (b / c) : (
            _mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c))))) <= a) : (_mut124 ? ((
            _mut121 ? (b % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) : (_mut118 ? (
            b - c) : (b + c))))) < a) : (_mut123 ? ((_mut121 ? (b % c) : (_mut120 ? (b
             / c) : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c))))) != a) : (
            _mut122 ? ((_mut121 ? (b % c) : (_mut120 ? (b / c) : (_mut119 ? (b * c) :
            (_mut118 ? (b - c) : (b + c))))) == a) : ((_mut121 ? (b % c) : (_mut120 ?
            (b / c) : (_mut119 ? (b * c) : (_mut118 ? (b - c) : (b + c))))) > a)))))))
            ))
34          return ISOSCELES;
35      return INVALID;
36    }
37  }
```

Listing A.4. TestSuiteForVarexJReflect.java class for running tests in VarexJ.

```
1   package br.ufmg.labsoft.cmu;
2
3   import java.lang.reflect.InvocationTargetException;
4   import java.lang.reflect.Method;
5   import java.util.HashSet;
6   import java.util.Set;
7
8   import org.junit.After;
9   import org.junit.Before;
10  import org.junit.Ignore;
11  import org.junit.Test;
12
13  import gov.nasa.jpf.util.test.TestJPF;
14  import triangle.tests.Triangle1Test;
```

```
15 import triangle.tests.Triangle3Test;
16
17 public class TestSuiteForVarexJReflect extends TestJPF {
18
19     /**
20      * modify only here
21      */
22     @SuppressWarnings("unchecked")
23     static Class<?>[] testCaseClasses = new Class[] {
24             Triangle1Test.class,
25             Triangle3Test.class
26     };
27
28     @Test
29     public void testAll() {
30         int countTests = 0;
31         Set<String> methodsSet = new HashSet<>();
32         long initTime = System.currentTimeMillis();
33
34         if (verifyNoPropertyViolation(VarexJConstants.JPF_CONFIGURATION)) { // VarexJ
               specific invocation (must be if)
35
36             for (Class<?> testCaseClass : testCaseClasses) {
37
38                 Object testCaseInstance = null;
39                 try {
40                     testCaseInstance = testCaseClass.newInstance();
41                 } catch (InstantiationException | IllegalAccessException e1) {
42                     e1.printStackTrace();
43                 }
44
45                 Method beforeMethod = null;
46                 Method afterMethod = null;
47                 Method[] testCase = testCaseClass.getMethods();
48
49                 for (Method m : testCase) {
50                     if (m.isAnnotationPresent(Before.class)) {
51                         beforeMethod = m;
52                         try {
53                             beforeMethod.setAccessible(true);
54                         } catch (SecurityException e1) {
55                             e1.printStackTrace();
56                         }
57                     }
58                     else if (m.isAnnotationPresent(After.class)) {
59                         afterMethod = m;
60                         try {
61                             afterMethod.setAccessible(true);
62                         } catch (SecurityException e1) {
63                             e1.printStackTrace();
64                         }
65                     }
66                 }
67
68                 for (Method m : testCase) {
```

```
69
70                      if (m.isAnnotationPresent(Test.class) && !m.isAnnotationPresent(
                            Ignore.class)) {
71
72                          try {
73                              if (beforeMethod != null)
74                                  beforeMethod.invoke(testCaseInstance);
75
76                              Class<? extends Throwable> expectedExeption = m.
                                    getAnnotation(Test.class).expected();
77                              if (!expectedExeption.isAssignableFrom(Test.None.class)) {
78
79                                  try {
80                                      m.invoke(testCaseInstance);
81                                  } catch (InvocationTargetException ite) {
82                                      if (!ite.getCause().getClass().isAssignableFrom(
                                            expectedExeption)) {
83                                          throw ite;
84                                      }
85                                  }
86                              }
87                              else {
88                                  m.invoke(testCaseInstance);
89                              }
90
91                              if (afterMethod != null)
92                                  afterMethod.invoke(testCaseInstance);
93
94                          } catch (InvocationTargetException | IllegalAccessException |
                                IllegalArgumentException e) {
95                              System.out.println(testCaseClass.getName() + '.' + m.
                                    getName());
96                              System.out.println("INVOKE-EXCEPTION");
97                          } catch (Error e) {
98                              System.err.println("ERROR");
99                          } catch (Exception e) {
100                             System.out.println("EXCEPTION");
101                         } catch (Throwable t) {
102                             System.out.println("THROWABLE");
103                         }
104                     }
105                 }
106             }
107         } //END OF VarexJ particular if
108     }
109 }
```