

**Dissertation**

---

**On the Use of Constraints in Program Mutation  
and Its Applicability to Testing**

---

Simona Alina NICA

Graz, 2013

*Institute for Software Technology  
Graz University of Technology*



Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Second reviewer: Univ.-Prof. Dr. techn. Nina Yevtushenko



---

*"Giving, You shall receive! "*  
*Nicolae Steinhardt*



# Abstract (English)

One important aspect in the software development life cycle is software testing. By software testing, the application is executed with the goal to detect early stage development errors, i.e., detect failures through test case execution. In his book [125], Myers states that half from the time invested in the process of software development is later invested in software testing. One can apply testing at different stages from the development process of a software application.

The history of testing dates long time ago. It officially starts in 1947, when the researcher Grace Murray Hopper introduces for the first time the notion of *bug*, due to problems encountered while working on a Harvard Mark II computer. Nevertheless, for a long period of time there was no clear difference between debugging and testing. It was only in 1957 when software testing started to distinguish itself from debugging. Myers is the first who defines the clear separation of testing from debugging, introducing the definition for software testing, which is still used nowadays [125].

The research work herein concentrates on software testing, more particularly on mutation testing. Mutation testing is a fault injection testing technique, where faults are deliberately injected into the original software application to measure the effectiveness of the available test suite. These faults are introduced by means of different mutation operators, producing thus several mutated versions for an application. The test suite pool is run against all the mutated version and a metric, i.e., mutation score, is computed to evaluate the effectiveness. Two major drawbacks appear in mutation testing: *time complexity*, due to the high computation time needed to run all the mutants, and *equivalent mutants*. Equivalent mutants are versions of the original software application which semantically behave the same with the original program, but syntactically they differ. This is one important problem, which we aim to overcome in our research.

The work presented here aims to come up with a reliable solution for the automated testing process. We make use of a constrained based approach for test case generation. We extend the work, taking

*Abstract (English)*

---

benefit from the debugging process, which is the next phase after a fault has been detected, i.e., after the process of software testing. We can thus model the research idea as a constraint satisfaction problem, deriving a more effective solution.

# Abstract (German)

Ein wichtiger Aspekt im Software-Entwicklungs-Lebenszyklus ist Software Testen. Durch Software Testen wird die Anwendung mit dem Ziel ausgeführt, frühe Entwicklungsphase-Fehler zu erkennen, d.h., Fehler durch Testfallausführung zu erkennen. In seinem Buch erklärt Myers [125], wie die Hälfte der in den Software-Entwicklungsprozess investierten Zeit später in Software-Testen investiert wird. Man kann Tests in verschiedenen Stadien des Entwicklungsprozesses einer Software-Anwendung einsetzen.

Die Geschichte des Testens begann vor langer Zeit. Es war im Jahr 1947, als die Forscherin Grace Murray Hopper zum ersten Mal den Begriff *Bug* einführte, auf Grund von Problemen, die während der Arbeit an einem Harvard Mark II-Computer auftraten. Dennoch gab es für eine lange Zeit keinen klaren Unterschied zwischen Debuggen und Testen. Erst mit dem Jahr 1957 begann Software-Testen sich von Debugging zu unterscheiden. Myers ist der erste, der die klare Trennung zwischen Testen und Debugging definiert, durch die Einführung der Definition für Software-Testen, die noch heute verwendet wird Myers [125].

Die Forschungsarbeiten hierin konzentrieren sich auf Software-Testen, insbesondere auf die Mutationstest - Methode. Mutation-Testen ist eine Fehlerinjektion-Testtechnik, wo Fehler absichtlich in die originale Software-Anwendung injiziert werden, damit man die Wirksamkeit des verfügbaren Testpakets messen kann. Diese Fehler werden mittels unterschiedlicher Mutationsoperatoren eingeführt und damit werden mehrere mutierte Versionen für eine Anwendung erzeugt. Der Test-Suite Pool wird gegen die gesamten mutierten Versionen ausgeführt und eine Metrik, d.h., der Mutation Score, wird berechnet, um die Wirksamkeit zu bewerten. Zwei wesentliche Nachteile treten bei Mutation Testen auf: *Komplexität*, aufgrund der hohen Rechenzeit, die benötigt wird, um alle Mutanten laufen zu lassen, und *äquivalente Mutanten*. Äquivalente Mutanten sind Versionen des Originalprogramm, die sich semantisch gleich wie das ursprüngliche Programm verhalten, sich aber syntaktisch unterschei-

den. Dies ist ein wichtiges Problem, das wir in unserer Forschung überwinden wollen.

Die hier vorgestellte Arbeit zielt darauf ab, eine zuverlässige Lösung für den automatisierten Testprozess zu erhalten. Wir nutzen einen Constraint-basierten Ansatz für die Erzeugung von Testfällen. Wir erweitern die Arbeit, wobei wir vom Debugging-Prozess profitieren, der die nächste Phase ist, nachdem ein Fehler erkannt wurde, d.h., nach dem Prozess des Software Testen. So können wir das Forschungskonzept als Constraint Satisfaction Problem modellieren, sodass eine effektivere Lösung abgeleitet wird.



# Acknowledgment

I would like to justify my exhaustive acknowledgments, by giving you a small parable (parable of Rabbi Eizik from Krakow), which, I believe, in a way it defines my experience over the last four years. I hope you will have the patience to receive it.

*"Rabbi Bunam used to tell young men who came to him for the first time the story of Rabbi Eizik, son of Rabbi Yekel of Krakow.*

*After many years of great poverty which had never shaken his faith in God, he dreamed someone bade him look for a treasure in Prague, under the bridge which leads to the kings palace. When the dream recurred a third time, Rabbi Eizik prepared for the journey and set out for Prague. But the bridge was guarded day and night and he did not dare to start digging. Nevertheless he went to the bridge every morning and kept walking around it until evening. Finally, the captain of the guards, who had been watching him, asked in a kindly way whether he was looking for something or waiting for somebody. Rabbi Eizik told him of the dream which had brought him here from a faraway country. The captain laughed: "And so to please the dream, you poor fellow wore out your shoes to come here! As for having faith in dreams, if I had had it, I should have had to get going when a dream once told me to go to Krakow and dig for treasure under the stove in the room of a Jew Eizik, son of Yekel! Eizik, son of Yekel! I can just imagine what it would be like, how I should have to try every house over there, where one half of the Jews are named Eizik and the other Yekel!" And he laughed again. Rabbi Eizik bowed, traveled home, dug up the treasure from under the stove, and built the House of Prayer which is called "Reb Eizik Reb Yekels Shul".*

*"Take this story to heart, and make what it says your own: There is something you cannot find anywhere in the world, not even at the Zaddiks, and there is, nevertheless, a place where you can find it", Rabbi Bunam used to add."*

An important conclusion towards the above parable: *the "solution", the "answer", it's right on the corner, but to discover it you need the salutary intervention of an "outsider": a stranger to your country, to your religion, to your language, to your culture. The immediacy of the "treasure" reveals itself as the benefit of "foreignness", of a temporary alienation from yourself and from your familiar environment!*(Andrei Plesu). It was a laborious experience of "far away", a journey towards myself that helped me discover a small part of the "solution".

When I think of all the people who stood by me and help me to become the person I am today, first of all I need to thank God with all my heart, for carrying me so wonderful through life, for blessing

## *Acknowledgment*

---

me with an extraordinary family, for all the people He has sent on the way, all my dear friends, but also for all my back friends. The last four years were a permanent grown-up process, a continuous advancement, both professionally and privately, and each of the people I met brought a beautiful contribution.

I would like to thank my supervisor, Professor Franz Wotawa. He is a model for me, in what concerns the academic and the family life. Thank you for all your confidence, all your help and all the patience you showed me. I would also like to express my deep appreciation for the members of my defense committee, Professor Nina Yevtushenko and Professor Denis Helic. Thank You for the time and patience you have invested in reviewing my thesis and my examination. Thank you, Nina, for the TAROT talks. Also, I would like to express my deep appreciation for Professor Mircea Grosu, CEO at CS Romania, for seeding in me the passion for software testing. I would like to thank Arabella, for her unconditional help. All my consideration for my collaborators, and, especially, for the people who founded my research: the competence network **SoftNet Austria** ([www.soft-net.at](http://www.soft-net.at)), the **Austrian Federal Ministry of Economy, Family and Youth**(BMWFJ), the province of Styria, the **Steirische Wirtschaftsförderungsgesellschaft mbH**.(SFG), the city of Vienna in terms of the center for innovation and technology (ZIT), and the **Austrian Science Fund**(FWF).

I am expressing all my admiration for my wonderful parents, Marioara and Iosif, for all their love and support. Thank you for teaching me that everything is possible, that there are no mountains too high, thank you for all your valuable lessons, for raising me so beautiful. I want to thank, with all my heart, my brother and also my best friend, Mihai! Thank you for always being close to me, thank you for your trust, for having an infinite patience with me and for your unconditional support. I am so grateful to Iulia, my sister-in-law, for all her confidence and support. You are really a wonderful sister for me! Valentin, my second best friend and, for the last two years, my office colleague, thank you! I would like to thank Aniela, for being like my grandmother, for all her love and her kindness. A special thought for my dear-beloved grandmother, Maria. Thank you for inspiring me our religion, thank you for raising in me the beautifulness of the Romanian spirit and our sensible traditions. I would like to thank Regina, for all her help. All my gratitude for my dear friends Carolina, Simona, for being like a second sister, and Catalin, Ionela, Johannes, Anca, Crina and Dana!

Simona Nica  
Graz, 2013

---

### **Statutory Declaration**

*I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.*

Graz, \_\_\_\_\_

Place, Date

\_\_\_\_\_  
Signature

### **Eidesstattliche Erklärung**

*Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.*

Graz, am \_\_\_\_\_

Ort, Datum

\_\_\_\_\_  
Unterschrift



# Contents

<b>Abstract (English)</b>	<b>iii</b>
<b>Abstract (German)</b>	<b>v</b>
<b>Acknowledgment</b>	<b>vii</b>
<b>List of Figures</b>	<b>5</b>
<b>List of Tables</b>	<b>7</b>
<b>1. Introduction</b>	<b>9</b>
1.1. Motivation . . . . .	9
1.2. Problem Statement . . . . .	11
1.3. Contribution . . . . .	12
<b>2. Software Testing</b>	<b>15</b>
2.1. Basic Definitions . . . . .	19
2.2. Software Testing Techniques . . . . .	21
2.2.1. Black Box Testing . . . . .	21
Equivalence Class Partitioning . . . . .	22
Boundary Value Analysis . . . . .	23
State Transition Testing . . . . .	25
Cause-Effect Graphing and Decision Table Technique . . . . .	26

Use Case Testing . . . . .	28
Other Black Box Testing Techniques . . . . .	30
2.2.2. White Box Testing . . . . .	30
Statement Coverage . . . . .	31
Branch Coverage . . . . .	31
Test of Conditions . . . . .	32
Path Coverage . . . . .	32
2.2.3. Gray Box Testing . . . . .	33
2.2.4. Fault Injection . . . . .	34
2.3. Conclusions . . . . .	35
<b>3. Mutation Based Testing</b>	<b>37</b>
3.1. Introduction . . . . .	38
3.2. Mutation Testing Background . . . . .	39
3.2.1. Mutation Testing Strategies . . . . .	42
3.2.2. Mutation Testing Tools . . . . .	47
<b>4. An Efficient Test Suite</b>	<b>51</b>
4.1. Early Stage of the Research . . . . .	52
4.1.1. MuJava . . . . .	52
4.1.2. Mutation of EJB Components . . . . .	53
4.2. An Industry Setting . . . . .	55
4.2.1. Environment Configuration . . . . .	57
4.2.2. Applied Mutation Tools . . . . .	58
4.2.3. Research Strategy . . . . .	59
Time . . . . .	60
Mutation Results . . . . .	60
Encountered Issues . . . . .	61
4.2.4. Conclusion . . . . .	64
4.3. Mutation Based Test Case Generation . . . . .	65
4.3.1. Introduction . . . . .	65
4.3.2. Related Approaches . . . . .	70
4.3.3. Generating Distinguishing Test Cases . . . . .	73
Experimental results . . . . .	84

4.3.4. Future Steps . . . . .	89
4.4. Equivalent Mutant Detection . . . . .	89
4.4.1. Related Work . . . . .	90
4.4.2. Constraint Satisfaction Problem . . . . .	91
4.4.3. Algorithm . . . . .	93
4.4.4. <b>EqMutDetect</b> Tool . . . . .	95
4.4.5. Experimental Results . . . . .	99
4.5. Conclusions And Open Issues . . . . .	102
<b>5. Mutation Testing from An End User Perspective</b>	<b>105</b>
5.1. End User Testing Methods . . . . .	106
5.2. End User Mutation Testing . . . . .	112
5.2.1. Algorithms And Experimental Results . . . . .	113
5.3. Conclusion . . . . .	115
<b>6. Conclusions</b>	<b>117</b>
<b>Bibliography</b>	<b>121</b>





# List of Figures

1.1. First Bug . . . . .	10
2.1. V Software Development Model . . . . .	17
2.2. Original Program . . . . .	21
2.3. Mutated Program . . . . .	21
2.4. State Transitions for an ATM PIN Entering Operation . . . . .	27
2.5. Cause Effect Diagram for Money Withdraw . . . . .	28
2.6. ATM Use Case Example . . . . .	29
2.7. GCDG Program . . . . .	31
2.8. Statement Control Flow Graph . . . . .	31
2.9. Black Box Testing Vs. White Box Testing . . . . .	33
3.1. Mutation Testing Process . . . . .	40
4.1. EJB Mutation Analysis . . . . .	56
4.2. Mutation Score . . . . .	57
4.3. A binary search algorithm comprising a bug in Line 2 . . . . .	68
4.4. Two iteration unrolling for the program from Figure 4.3 . . . . .	75
4.5. Partial SSA representation corresponding to the program from Figure 4.4 . . . . .	77
4.6. The Constraint Representation of the Program From Figure 4.5 . . . . .	79
4.7. Equivalent Mutant . . . . .	83
4.8. Program for computing the maximum of two numbers . . . . .	93

*List of Figures*

---

4.9. Original Program and Its ROR Mutant . . . . .	98
4.10. $CS_{PMi}$ of the Original Program and Its Mutant . . . . .	99
4.11. CSP Solution . . . . .	100
5.1. Spreadsheet for average grade computation, taken from [25] . . . . .	109
5.2. Spreadsheet for feet to meter conversion . . . . .	113
5.3. Graph representation of spreadsheet from 5.2 . . . . .	113

# List of Tables

2.1. Input Partitions . . . . .	23
2.2. Output Partitions . . . . .	23
2.3. Minimum Test Suite For Equivalence Partitioning . . . . .	24
2.4. ATM Decision Table . . . . .	27
3.1. Mothra Mutation Operators . . . . .	42
4.1. MuJava Method Level Mutation Operators . . . . .	53
4.2. MuJava Class Level Mutation Operators . . . . .	54
4.3. Overview environment configuration . . . . .	57
4.4. Eclipse JUnit Test Results . . . . .	61
4.5. Mutation Testing Information per Mutation Testing Tool . . . . .	62
4.6. Success Rate . . . . .	62
4.7. Experimental Results . . . . .	86
4.8. TCAS Experimental Results . . . . .	87
4.9. EqMut Detection Results . . . . .	101
5.1. Spreadsheet Mutation Operators . . . . .	114



# Chapter 1

## Introduction

*"A clever person solves a problem. A wise person avoids it."* - Albert Einstein

### 1.1. Motivation

An important phase in the development life cycle of an application is the testing process. Through testing, the application is executed with the goal to detect early stage development errors, i.e., detect failures through test case execution. Testing can be done in different stages of the development process.

It is essential to understand that testing means identifying defects (through one or several failing test cases), and debugging is locating and fixing the identified defects. It was only when problems to the Harvard University Mark II Aiken Relay Calculator, revealed a moth blocked in one of the computer relays, that the researcher Grace Murray Hopper introduced officially, for the first time, the notion *bug*. Debugging can be seen as a consequence of testing. Therefore, deriving an efficient testing phase is of high importance. For example, in February 2008, because of a system update, a software error occurred at the automated baggage sorting system from the Heathrow Airport; thousands of people were not able to check in their baggage. Due to a non treated exception for a floating-point error, when a 64-bit integer was converted to a 16-bit signed integer, in June 1996, the first flight of the European Space Agency's Ariane 5 rocket failed immediately after launching. The estimated loss was of a half billion US dollars. In October 1999, NASA lost 125 million US dollars due to

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

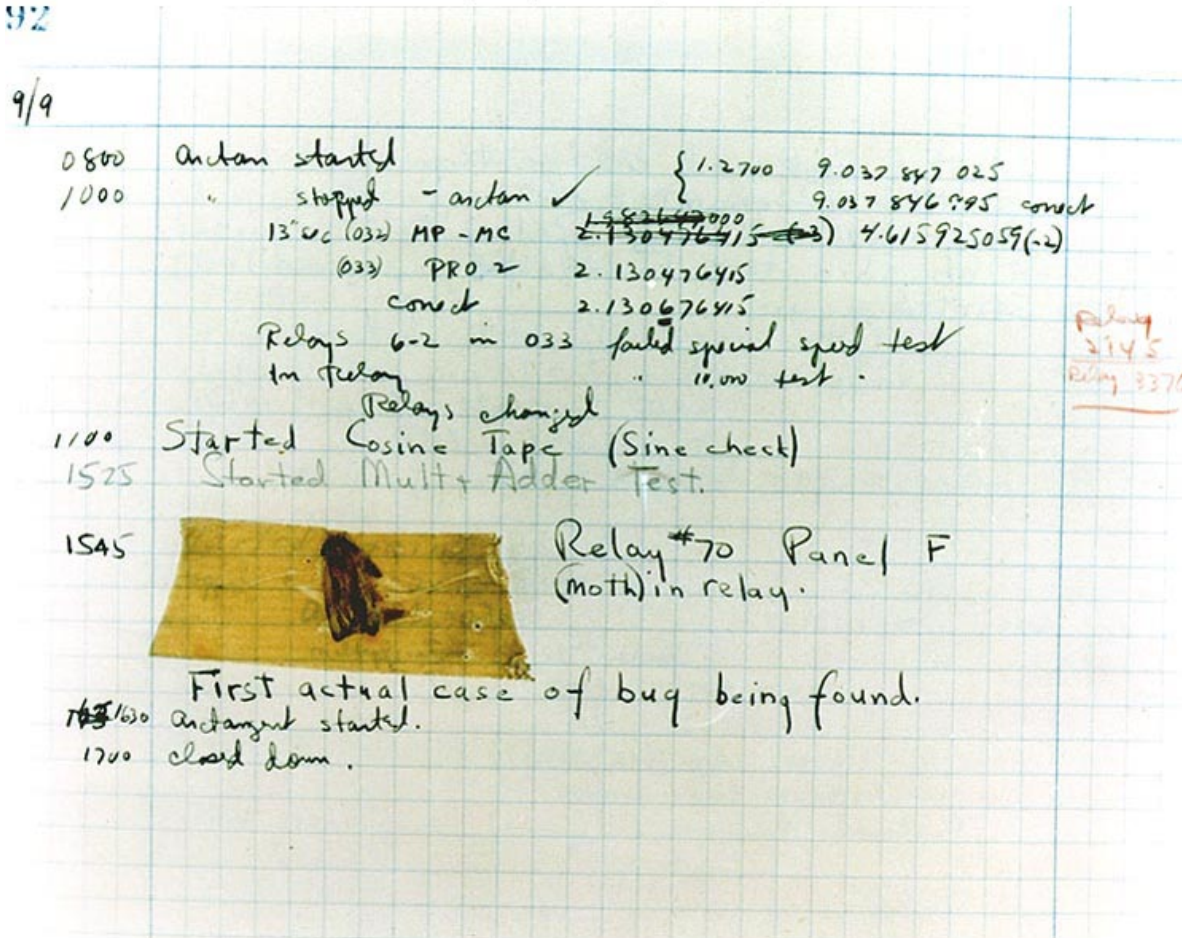


Figure 1.1.: First Bug

computations performed in yards and not in meters, by the spacecraft software. As a consequence, an interplanetary weather satellite was lost in space. Besides the tremendous financial losses, even more critical are the human losses that are induced by different types of software failures. In 1985 and 1986 people died because of the Therac-25 Medical Accelerator. Therac-25 was a machine used as radiation therapy in cancer diagnostics. Due to code reuse, from a previous version, and, therefore, not treated arithmetic overflows, the machine caused an overdoses of unfiltered and unshielded X-rays, producing thus radiation poisoning. In February 1991, 28 US Army soldiers die in Saudi Arabia - Dhahran, due to a barrack hit from an Iraqi Scud. This happened due to a software error of the system clock inside the Patriot defense system. It was a Patriot missile battery which has functioned constantly for more than 100 hours, and the internal clock of the system has deal with an accuracy loss of one third of a second. The defense system failed to detect the Scud missile. These are between the most common software errors. But we can track more and more similar errors, in different activity areas. Therefore special attention must be given to the software testing phase - especially when considering critical embedded systems; it is a complex process, involving a through planning, training and execution. It has been proven that even today more that 50 percent from the software activity is taken by the modification phase [125]. Manual testing is tedious and error prone, therefore a fair solution is automatic testing. Testing can be done considering the tester/programmer's experience, considering the software structure or the functional specifications, at every level from the development life cycle of an application.

The work presented here aims to come up with a reliable solution for the automated testing process, by considering a fault injection approach. We make use of a constrained based approach for test case generation. We extend the work, taking benefit from the debugging process, which, as already presented in the beginning, is a continuation of the testing process.

## 1.2. Problem Statement

Given a program **P** or a set of specification *Spec* of a system *S*, a test suite **TS** and a set of properties **Prop** for the system, which must hold over the program **P**, correspondingly over the set of specifications *Spec*, we intend to

1. Check if **TS** is complete through fault injection ,i.e., assess the quality of the available test suite;
2. If **TS** is not complete:

- Verify if we deal with dead code or not treated exceptions/parts of the code, or
  - Enlarge the test suite, with new test scenario, through a constraint based approach;
3. Extend the testing procedure to debugging and help in the diagnosis process;
  4. Detect semantically equivalent versions of the software, which can result from the fault injection strategy.

### 1.3. Contribution

There are a lot of research studies which are based on mutation testing, when searching for a method to assess the test set quality [86]. The work presented here has as main goal the improvement of the testing process, by using mutation testing for the test case reduction, and also generation, process. Mutation testing is combined together with the constraint satisfaction problem, to allow for an early detections of those equivalent programs, models of the original available source. The technique and the algorithms will be described in detail, and an automated tool is proposed. An automated process can significantly reduce the manual, both human and resource, effort.

The current thesis was constructed having as basis the below listed publications:

- *Challenges in Applying Mutation Analysis on EJB-based Business Applications* [134];
- *Does testing help to reduce the number of potentially faulty statement in debugging?* [127];
- *Using Distinguishing Tests to Reduce the Number of Fault Candidates* [128]
- *Improving the Mutation Score by Means of Distinguishing Test Cases* [132]
- *On the Improvement of the Mutation Score Using Distinguishing Test Cases* [131]
- *Constraint-Based Debugging Combining Mutations And Distinguishing Test Cases* [202]
- *Is Mutation Testing Scalable for Real-World Software Projects? - A Case Study on Eclipse* [135]
- *Detecting Equivalent Mutants by Means of Constraint Systems* [133]
- *On the Use of Mutations And Testing For Debugging* [129]
- *EqMutDetect A Tool for Equivalent Mutant Detection in Embedded Systems* [136]



- *Using Constraints for Equivalent Mutant Detection* [137]

The thesis is organized as follows. In Chapter 2 we describe the available software testing techniques. In 2.1, we give an overview of the basic technical notions we use. We describe the basis of the mutation testing technique in Chapter 3. The main goal of our research is towards test case generation. In Chapter 4 we propose a test case generation method, by making use of the information we receive from the debugging process - Section 4.3, and then by eliminating the equivalent mutants - Section 4.4. A second direction of our research, from an end user point of view, is detailed in Chapter 5. Chapter 6 will conclude the research work, giving an overall overview on the improvements, but also on the still opened questions and future steps to follow.



# Chapter 2

## Software Testing

*“Testing shows the presence, not the absence of bugs” - Edsger W. Dijkstra.*

Software can be found in almost every system of our society, starting from a simple life routines, e.g., a remote control, a microwave, a music station, and going to a person’s daily activities, e.g., a cell phone, an ATM, to more complex and critical systems, e.g., embedded software: traffic control systems, avionics, railway transport systems, automotive industry, etc. Therefore testing the software is, in many situations, of vital importance. A software product is normally developed accordingly to a software development life cycle. Many models of software development are available in the literature and the choice of one model depends on the organizations involved in the process. A model encapsulates all the phases of the product, from its early concept until maturity. Each model has its advantages and its drawbacks:

1. In the **general mode**, each development phase depends on the previous one.
  - a) *Requirements* design phase. Project managers work together with the stake holders, e.g., the end users, and set up a well defined list of requirements the product should fulfill, i.e., who are the end users of the system, what is the input, what information must come at the output, etc. The goal is to obtain the behavioral application.
  - b) The *design* phase comes right after the requirements definition step. During this phase, one or more software architects determine how the system will actually work, i.e., communication between modules, create the model of the application (UML diagrams).

- c) After the design is complete, the *implementation* takes place.
  - d) Testing represents verifying that what was described in the requirements phase was correctly implemented.
2. But the most common model is the **waterfall model**. Usually this is recommended for small applications. The difference from the above presented model is that after each phase is ended, there comes also a review process in order to check if the software is growing accordingly to its trajectory. This model is quite intuitive and easy to use, but, as each phase must be completed and checked, problems might appear when a change in the software is needed. Therefore this suits well for systems with a reduced risk.
  3. **V Model**. This is similar to the waterfall model (sequential execution of paths). But here the testing phase is conducted before the implementation takes place, i.e., to each development phase it corresponds a testing phase. Also this model is quite easy to use and because of each testing step, it can be more successful than the above presented model. But this model, too, is not flexible and suffers from the same drawback as the waterfall model.
  4. **Incremental Model** can be seen as a multiple waterfall model. Each phase from the development life cycle is done in an incremental manner. In this model, after the development phase is completed, regression testing comes next, making thus easier the testing process.
  5. **Spiral Model**. This model combines the phases from the waterfall model with the prototypes of a product, i.e., with the iterative development of the software product. The spiral model is designed for large and complex software projects.

In the literature there are many models for the development life cycle of an application, e.g., iterative model, agile software development model, dual Vee model, rapid application development (RAD model), extreme programming (XP), etc. Throughout this research thesis, we will always refer to the V model for the software development life cycle. As we can observe from the above model description, different development phases exist. Each development phase has associated a specific software testing phase:

- Unit testing - deals with testing the implementation.
- Component(Module) Testing - test the detailed design of each software component / module.
- Integration testing - test the application with respect to the modules interaction, with respect to the specifications.

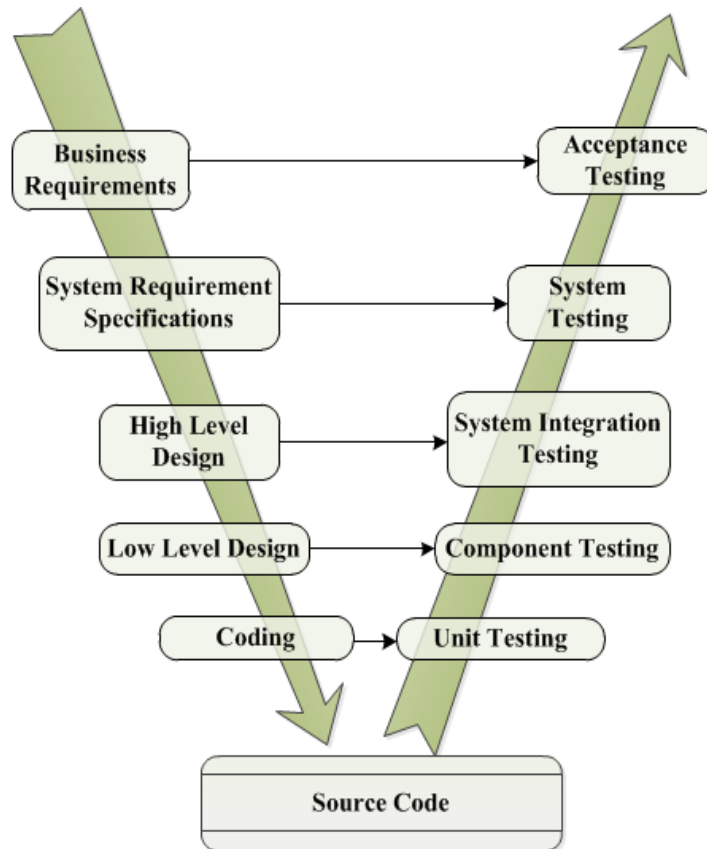


Figure 2.1.: V Software Development Model

- System testing - test the design of the entire software system, according to the defined requirements.
- Acceptance testing - test the software requirements, according to what was established together with the customer, in the contract.

A brief representation of this model is depicted in Figure 2.1.

When starting the testing process, the test design technique plays an important role. This is due to the fact that we will never be able to produce exhaustive testing, and therefore we must create, as much as possible, a minimum set of test cases with the highest impact in detecting faults. Depending on the exact method used for testing, we can have the following software testing design methods:

- Manual testing: the testing strategy is manually conducted, i.e., searching for failures is manu-

ally run.

- Automated testing: tests are created, executed, interpreted (test results) automatically through a software tool, i.e., it represents the automation of the manual testing process.
- Black box testing: the source code is not available to the tester. The testing process is guided only by the software's specifications.
- White box testing: also called glass-box testing, has the property that the tester has access to the complete source code of the program.
- Gray box testing: information about the structure of the program are known but no information about the actual source code, e.g., pre- and post-conditions for a method.

All the last three testing methods, can be either manually or through an automated process conducted.

If we consider the type of testing we design, i.e., the reason a user conducts a testing process, and also the software applicability, we can do:

- Functional testing: black box testing process. Tests are designed according to the specifications, i.e., test the functions by providing the input and checking the output; both input and output must be compliant with the specifications of the software.
- Regression testing: testing procedure in which only the modified software is re-tested, i.e., fixing a bug will not trigger new errors.
- Usability testing: it is usually conducted by the users, i.e., how will the users be able to use the software application. This is a black box testing technique.
- Performance testing: verifies the stability and response times of a system.
- Load testing: assumes to test a system, by causing a high demand on it; for example simulate the concurrent access to the application of multiple users.
- Conformance testing: test if the system is compliant with the defined standards for that system. For example, the telecommunication industry has some protocol standards it must fulfill.
- Security testing: tests that the information from a software system will be protected, i.e., authentication information, confidentiality of the users, tests authorized operations, etc.
- GUI testing: responsible for testing the graphical user interface, with respect to the specifications.

- Installation testing: it is oriented more towards the quality assurance area. Checks if there are available all the features for installing and running the product.
- Acceptance testing: test in order to establish whether the software implements the desired requirements.

In the selection of the techniques we theoretically introduced, but also the definitions, we make use of the notions and methods described by the authors in [171] and [125].

### 2.1. Basic Definitions

In this section we will present the definitions of all the notions we will use throughout the current work. We base our definitions on [127, 125].

**Definition 1 Verification** *represents the process which checks if at a give phase of the development life cycle, the software is compliant with what is supposed to do, i.e., answers the question if the software product was **right** developed.*

**Definition 2 Validation** *is the process conducted when the development has been finished and it ensures that the software was developed according to the initial specified requirements, i.e., answers the question if the **right** software product was developed.*

**Definition 3 Fault.** *A fault or a bug is defined as a defect in the source source, i.e., a certain software doe not succeed to achieve its functionality.*

**Definition 4 Error** *An error appears as a consequence of a fault, and this is due to the incoherence from the source code when compared to the specifications.*

**Definition 5 Failure.** *A failure appears when a requirement is not fulfilled, as a difference between the expected behavior and the actual obtained result. The source of a failure is the error, which is triggered by a fault.*

**Definition 6 Testing.** *Represents the process through which a error is revealed in a program, by executing the program against a well defined set of test cases. According to [125], testing is "the process of executing a program with the intent of finding errors".*

**Definition 7 Debugging.** *The concept refers to localizing and correcting faults revealed through software testing. An important observation: testing is not debugging and debugging is not verification.*

**Definition 8 Test goal.** *A software test goal refers to the testing state that a software system must fulfill, i.e., verification, validation, testing traceability.*

**Definition 9 Oracle.** *An oracle is a procedure used in software testing to establish whether a test passed or it has failed. For the given inputs, the outputs resulted from testing are compared with the outputs computed by the oracle, for those given inputs. Oracles can be represented by specifications, documentation, the human judgment, heuristic oracles, etc.*

**Definition 10 Test Case** *We define a test case for a program  $\Pi$  as a set  $(I, O)$  where  $I$  is the input variable environment specifying the values of all input variables used in  $\Pi$ , and  $O$  the output variable environment. If no output variable environment is specified, we set  $O$  to  $\emptyset$ .*

A *failing test case* is a test case for which the output environment computed from the program  $P$ , executed over input  $I$ , is not consistent with the expected output. Otherwise, the test case is said to be a *passing test case*.

A *test suite*  $TS$  for a program  $\Pi$  is a set of test cases for  $\Pi$ .

**Definition 11 Distinguishing Test Case** *Given programs  $\Pi_1$  and  $\Pi_2$ . We say that a test case  $(I, O)$  is a distinguishing test case, if the output variable environments computed by  $\Pi_1$  and  $\Pi_2$  using the same input  $I$  are different.*

**Definition 12 Test Script.** *We define the test script as an automated test case, which will produce a report when executed against the software.*

**Definition 13 Mutation Operator.** *According to [7], a mutation operator is "a rule that specifies syntactic variations of strings generated from a grammar".*

**Definition 14 Mutant** *Given a program  $\Pi$  and a statement  $S_\Pi \in \Pi$ . Let  $S'_\Pi$  be a statement that results from  $S_\Pi$  when applying changes like modifying an operator or a variable. Then, program  $M$  is the mutant of program  $\Pi$  with respect to statement  $S_\Pi$ , obtained after replacing  $S_\Pi$  with  $S'_\Pi$ . Thus, the mutant results after there are applied one or several mutation operators.*



For the original program in Figure 2.2, a mutant is the program from Figure 2.3, which is obtained by replacing the relational *EQUALS* operator with *GREATER OR EQUALS* relational operator.

**Definition 15 Equivalent Mutant** Given a program  $\Pi$ , and one of its mutants  $M$ . We say that  $M$  is an equivalent mutant if semantically  $M$  behaves exactly like  $\Pi$ . If we consider the distinguishing test case definition, then for an equivalent mutant we will not detect a test case, able to point out the difference between the original program  $\Pi$  and its corresponding mutant,  $M$ .

```
int a, b;
int compute;
if (a == b)
    compute = a;
return compute;
```

Figure 2.2.: Original Program

```
int a, b;
int compute;
if (a >= b)
    compute = a;
return compute;
```

Figure 2.3.: Mutated Program

## 2.2. Software Testing Techniques

### 2.2.1. Black Box Testing

The black box testing design technique is also known as *input/output driven* or *data driven* testing [125]. The test data will be derived using only the specifications of the software, and the tester will treat the software as a black box, where only the inputs and the expected behavior are known, but not the internal structure of the software, i.e., the tester will do not have access to the software implementation. In black box testing the accent relies on the software quality of the system. Usually, tests developed under this design technique can be reused. But when trying to detect all the errors of a software product through black box testing, we have, in most cases, an exhaustive testing process, unless a well defined fault model is available. If we were to use only black box testing, practically we would have to verify every input condition. In most of the situations, this is quite problematic and impossible. Consider, for example, software where database transactions are involved, i.e., a railway train reservation system. In these situations, the current transaction will always depend on what it was done in the previous ones. In our case, it is equivalent to testing all transactions, valid or not valid. Therefore, as we said earlier we must use a finite, not too big, number of test scenarios through which

we could detect the maximum of errors. Mainly, in black box testing, we search for errors with respect to behavior, initialization, missing functions, access to a database. Black box testing can be applied at the following testing levels: unit testing, integration testing, system testing and acceptance testing.

### Equivalence Class Partitioning

**Principle:** Organize the input data of a module in partitions, gathered from the requirements, then generate test cases to cover the resulted partitions. The partitions subsets represent the equivalence classes.

According to [125], the equivalence class partitioning method must satisfy two properties:

- Must reduce at least by one the total number of test cases that need to be developed, in order to fulfill a certain testing criteria;
- Must cover a set of other possible tests.

When establishing the equivalence classes, each input condition is split in subclasses. Then there are determined the condition and the valid and invalid classes of equivalence. Tests must be designed for both the valid and invalid equivalence classes. By invalid classes we understand invalid input data, i.e., error prone. After establishing the equivalence classes, test cases are created to cover as many as possible classes of equivalence, until all the valid and invalid equivalence classes were covered.

### Example

For a given university lecture, a student obtains one grade for the laboratory and one for the written examination (grades are positive integer numbers). The final grade will be the sum of the two grades, where the examination grade is in the range 0..70 and the lab in the range 0..30. The grade must be in the range A..C, where A is excellent, B is satisfactory and C is insufficient. Let us denote by *SumGrade*, the final score a student obtains when summing the two grades and by *FinGrade*, the resulted grade range. Then, we say that:

- If  $\text{SumGrade} \geq 85$ , then *FinGrade* is A
- If  $\text{SumGrade} < 85$  and  $\text{SumGrade} \geq 65$ , then *FinGrade* is B
- If  $\text{SumGrade} < 65$ , then *FinGrade* is C.

Whenever an error appears, e.g., the grade is not in the above defined limits, then an error message must be triggered.

Valid Input Partitions	Invalid Input Partitions
$0 \leq \text{ExamGrade} \leq 70$	$\text{ExamGrade} > 70$
$0 \leq \text{LabGrade} \leq 30$	$\text{ExamGrade} < 0$
	$\text{LabGrade} > 30$
	$\text{LabGrade} < 0$
	ExamGrade is real number/ExamGrade is string
	LabGrade is real number/LabGrade is string

Table 2.1.: Input Partitions

Valid Output Partitions	Invalid Output Partitions
A: FinGrade = A when: $85 \leq \text{SumGrade} \leq 100 \leq \text{ExamGrade} \leq 70$	FinGrade = D
B: FinGrade = B when: $65 \leq \text{SumGrade} < 85$	FinGrade = A- or FinGrade = A+
C: FinGrade = C when: $0 \leq \text{SumGrade} < 65$	FinGrade = null
ErrMsg when: if $\text{SumGrade} > 100$ or $\text{SumGrade} < 0$	

Table 2.2.: Output Partitions

Let *ExamGrade* be the examination grade and *LabGrade* the laboratory grade. The identified input partitions can be observed in Table 2.1. The first column from the table depicts the valid input partitions and the second, the invalid ones.

Let *ErrMsg* be the error message, obtained when the final grade is outside the range, and  $\text{SumGrade} = \text{LabGrade} + \text{ExamGrade}$ . The identified output partitions are described in Table 2.2. Similar to the input partitions table, the first column denotes the valid output partitions and in the second one we show the invalid output partitions.

Totally, we can depict 17 equivalence classes. We can generate a test case for each equivalence class, resulting thus 17 test cases, or we can try to obtain a smaller, minimum, number of test cases with which to cover all the equivalence classes, i.e., a test case might cover one, two or more equivalence classes, so that we have a minimum test suite. In Table 2.3 we can see the minimum set of test cases we need in order to cover all the equivalence class partitions.

### Boundary Value Analysis

In boundary value analysis, the test cases should explore those values situated at the border of the input domains, but also outside, i.e., above and beneath the border. Boundary analysis applies not only over the input domains, but also over the outputs, i.e., test cases should try to simulate values

Test	ExamGrade	LabGrade	SumGrade	EQP ExamGrade	EQP LabGrade	EQP SumGrade	EQPFinGrade	FinGrade
1	65	25	90	$0 \leq \text{ExamGrade} \leq 70$	$0 \leq \text{LabGrade} \leq 30$	$85 \leq \text{FinGrade} \leq 100$	-	A
2				$0 \leq \text{ExamGrade} \leq 70$	$0 \leq \text{LabGrade} \leq 30$	$65 \leq \text{FinGrade} \leq 85$	-	B
3				$0 \leq \text{ExamGrade} \leq 70$	$0 \leq \text{LabGrade} \leq 30$	$0 \leq \text{FinGrade} \leq 65$	-	C
4							-	ErrMSG
5							-	ErrMSG
6	a	b	-	aaa	bbb	-	-	ErrMSG
7					$\text{FinGrade} < 0$	-	D	ErrMSG
8					$\text{FinGrade} > 100$	-	A-	ErrMSG
9					-	-	null	ErrMSG

Table 2.3.: Minimum Test Suite For Equivalence Partitioning

outside the expected limits. If we take into consideration the equivalence partitioning class, in boundary value analysis, test cases must consider the limits of the input and output equivalence classes. Designing boundary value analysis test cases depends much on the tester's experience and creativity. Nevertheless, some common situations involve:

1. For the input domain; if there is available a set of values, tests need to be done for the first and the last element of the domain, and also tests with invalid input values, i.e., just beneath the first element and above the last element;
2. In what concerns the output information, try to apply the same procedure, whenever this is possible.

If we consider the equivalence classes identified in the example from the previous sub-section, we need to write tests for each input and output classes of equivalence, both valid and invalid values. For example, for the input domain, we must have a test case that verifies the boundaries and the case when *ExamGrade* is less than 0 or above 70. The same states for *LabGrade* (smaller than 0 and greater than 30): *ExamGrade* == 0, *ExamGrade* == 70, *ExamGrade* == -1 and *ExamGrade* == 71

A recommended testing practice is to combine this technique with the equivalence class partitioning technique, as it might help in revealing faults [171].

### State Transition Testing

The state transition testing method is based on the concept of state diagrams. In this type of testing, we start from an initial state and, depending on the triggering events, we take different state transitions, reaching the end state. For this type of testing, one should be able to model the system under test as a finite state machine (state diagram), with inputs and outputs, where the nodes represent the states and the connection between nodes depict the transitions. A final state machine, according to Beizer's definition, is an "abstract machine (e.g., program, logic circuit, car's transmission) for which the number of states and input symbols are both finite and fixed. A finite state machine consists of states (nodes), transitions (links), inputs (link weights), and outputs (link weights)" [12].

In state transition, we can work either with abstract models or with more detailed ones, depending on the level of abstractization or elaborateness of the system under test. As critical or important the system is, as much the level of detail will increase. This is one important asset of this strategy. As already written above, four characteristics define a state transition:

1. The system states;
2. The transitions among the states;
3. The transitions trigger events;
4. The resulted actions.

Testing works according to a well defined testing criteria. A transition must execute, at least once, the specified behavior. For an easy identification of the test situations, the state machine can be translated into a tree of transitions. This procedure can be successfully used in system testing; a good example towards this method, is testing a graphical user interface, which can be easily represented as a finite state machine. In deriving the test suite, the tester must know, in advance, the initial state of the system, the inputs for testing, the expected output and the final state. A complete testing is done when we have reached, at least once, every state of the finite state machine and we have executed, at least once, each transition; also, every transition, which broke the specifications, must be verified. The technique is specific to functional testing.

Let us consider a money withdraw example with the PIN entering operation, from Figure 2.4. Mainly this contains: card insertion, PIN entering, PIN OK or Wrong PIN. In order to derive the test cases, we must take into account the following scenarios:

1. Normal PIN entering operation: PIN is correctly typed from the beginning;
2. Wrong PIN entered three time; this results in blocking the card, i.e., "eat the card";
3. PIN entered correctly at the second /third attempt;
4. Request account information.

Each transition, but also each state, might represent a test condition. In state transition, the testing all functions must be properly planned, i.e., executed. The completion criteria in state transition testing requires that each state is reached at least once and each transition is executed at least once, and also every property, which breaks the specification, is verified. This technique is suitable for deriving system testing.

### **Cause-Effect Graphing and Decision Table Technique**

This technique extends the previous one, by taking into account the input dependencies and their influence on the outputs, i.e., the relation between a cause and its effect over the system is represented

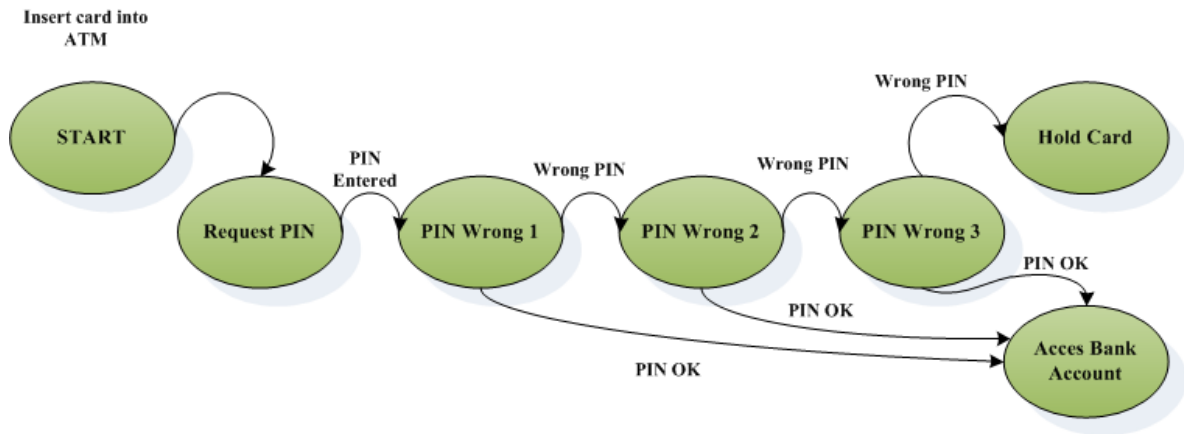


Figure 2.4.: State Transitions for an ATM PIN Entering Operation

Cause	No.1	No.2	No.3	No.4	No.5
Valid Card	0	1	1	1	
Correct PIN	-	0	0	1	1
PIN typed wrong 3 times	-	0	1	-	-
Amount available	-	-	-	0	1
<b>Effect</b>					
Reject card	1	0	0	0	0
Enter new PIN	0	1	0	0	0
Retain card	0	0	1	0	0
New amount	0	0	0	1	0
Retrieve money	0	0	0	0	1

Table 2.4.: ATM Decision Table

as a cause effect graph. Specifications can be used for cause identification. A cause is a condition and each condition is connected with the others by means of logical operators, e.g., AND, OR. The results of these conditions represent the effect. If we consider the example from the previous section, we can identify as causes (conditions) the validity of the card and of the PIN code, and also the sum to be taken from the ATM. As an effect we have an invalid card, i.e., a rejection operation, an incorrect PIN entered or wrong PIN format or an withdrawal operation, for the needed sum.

Figure 2.5, depicts this situation. In order to derive the test cases, the graph can be displayed as a decision table, with inputs and effects. A column from this table will be a test case. For our example, we can observe Table 2.4. The test criteria requires that, at least once, each column is considered.

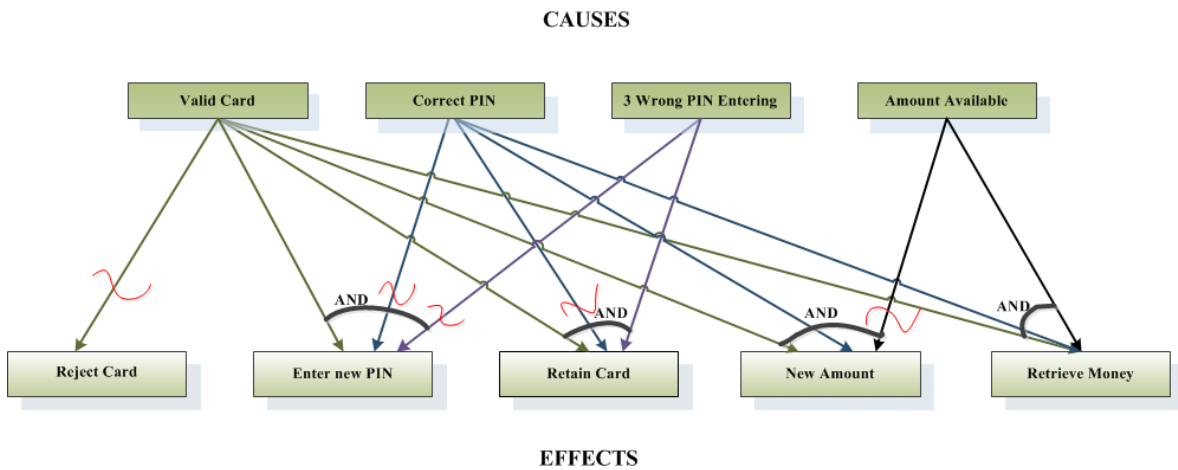


Figure 2.5.: Cause Effect Diagram for Money Withdraw

### Use Case Testing

In order to define the use case testing, we define what an use case is. According to *Wikipedia*, "A use case is a description of a systems behavior as it responds to a request that originates from outside of that system (the user)." The end user depicts how he/she will communicate with the software system, by defining a sequence of interactions. Such an interaction, which an end user takes to achieve a final goal, is called a **use case** of the system. For a sequence, several paths to follow might be selected. Each such a path represents a *use case scenario* [171]. In use case testing, pre and post conditions must be taken into consideration. For example, consider the case of an ATM and suppose a person wants to make a money withdrawal. One first pre condition is to make sure that the person uses a valid card. An immediate post condition is the pin request operation. An extract of an use case diagram can be observed in Figure 2.6. This represents a use case scenarios. In case of use case software testing, the test cases will result from the different test scenarios, but for each use case test it must exist a start point, the pre-conditions, different conditions, expected results and the post-conditions. We assume a normal operation, where the user enters the PIN correctly.

This testing method is applied with success in acceptance testing, but also in system testing. One stopping criteria, in case of use case testing, is when at least all possible sequences have been tested at least once.



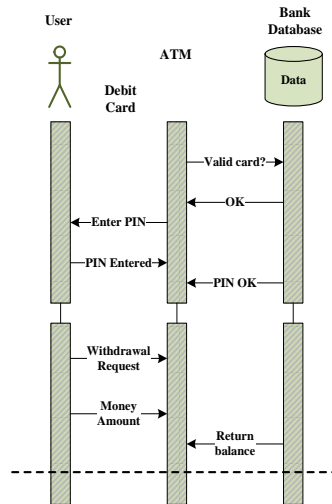


Figure 2.6.: ATM Use Case Example

### **Other Black Box Testing Techniques**

The techniques shortly described above are among the most widely use black box testing techniques. There are , still, some other black box testing strategies. Consider, for example, smoke testing or random testing. Smoke testing deals more with that type of testing which takes care that the most vital functionalities of a software work properly; it does not concentrate on the smaller details. According to the Institute of Electrical and Electronics Engineers, i.e., IEEE, a "daily build and smoke test is among industry best practices" [119]. Random testing, a type of functional testing, is not considered as a reliable software testing method. It consists in generating test cases, based on random generation algorithms, e.g., using as a criteria different coverage measures. Nevertheless, the authors in [53] evaluate this technique and demonstrate that it can help in discovering subtle errors with a small effort.

### **2.2.2. White Box Testing**

Unlike black box testing, in white box testing the tester will have access to the internal structure (design or implementation), e.g., software implementation of the program [171, 125]. As this design technique deals with the effective structure of the program, it is also known as structural testing. When using this design test technique, all the source code must be executed, but the obtained results must be compared against the requirements or the specifications. Due to the fact that all the source code must be executed and, therefore, covered, it can be a more precise design technique. There are some drawbacks the user must take care of. For complex tests, it is mandatory a strong knowledge of the implementation and also of programming. Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation. Also, in case changes in the implementation are often, then test maintenance can become difficult. In this testing strategy, there can also be a code inspection, i.e., code review, process conducted. This demands strong programming skills. Most complex form of white box testing is path coverage. But covering all the paths in a real word application is not a realistic situation, as forcing the execution of a specific path is not easy to trigger. One solution is code development following different coding rules.

In most of the situations, white box testing is applied to the unit testing level, but it can be also used for integration and system testing.

```

1. public int compute(int a, int b) {
2.   int result = 0;
3.   int newA = a;
4.   int newB = b;
5.   if (newA == 0)
6.     result = newB;
7.   if (newA != 0) {
8.     while (newB != 0) {
9.       if (newA > newB) {
10.        newA = newA - newB;
11.        result = newA;
12.      } if (newA <= newB) {
13.        newB = newB - newA; }
14.      result = newA; }
15.   return result;
16. }

```

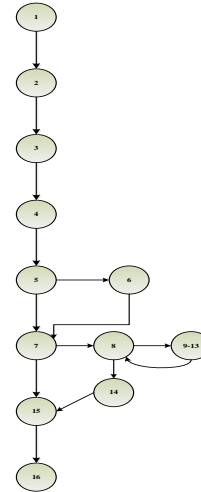


Figure 2.8.: Statement Control Flow Graph

Figure 2.7.: GCDG Program

### Statement Coverage

When using the statement coverage, the main idea involves to test each statement of the system under test. For example, consider the program in Figure 2.7, which computes the greatest common divisor, and its associated control flow graph in Figure 2.8. A complete testing would involve testing all statements, all paths of the control flow graph. A completion testing criteria, in statement coverage, is defined as the percentage of covered statements from the total number of statements. The advantage in statement coverage is that it can reveal dead code. Nevertheless, in statement coverage the empty part of an *IF-THEN-ELSE* instruction is not taken into account.

### Branch Coverage

Generating test cases to support the branch coverage criteria deals with the execution of every branch from a program, at least once, e.g., lines 5-6 or line 7 to 13 in Figure 2.7. In branch coverage, both *IF*

...*THEN* and *ELSE* branches must be considered. This is an advantage, when compared to statement coverage. In this case, the completion criteria for testing deals with the percentage of covered branches from the total number of branches. If we compare the number of test situations that must be designed in branch and statement coverage, a tester will need to develop more test scenarios, in case branch coverage is desired. Also, branch coverage testing helps in detecting missing statements in *ELSE* branches.

### Test of Conditions

Branch coverage only evaluates the *TRUE* or *FALSE* value of the condition. But, let us consider the case of a compound condition, e.g., *if (a <= b) AND (b > 5)*. In condition coverage, each part of the compound condition must be tested with both its false and true values. If we consider  $a = 3$  and  $b = 5$ , then first part of the condition,  $a <= b$  is *true* and second part holds *false*. For  $a = b = 6$  we have the value of true for the second part of the condition. We now need to test also for the false value of the first part of the condition, i.e.,  $a = 7, b = 3$ . In 1979, Myers introduced the concept of multiple condition coverage [125]. In multiple condition coverage all true and false combinations must be tested, i.e., for our small example, for a, b and the compound condition, we must consider *true, true, true, false, false, true, false, false*. Not all the time, all combinations will be possible. Also, unlike for statement and condition coverage, a smaller number of tests is required.

### Path Coverage

Path coverage is seen as the most reliable white box testing technique. It requires to execute each feasible path from a program, which is almost not possible. Although it could help reduce the number of test cases, this is not a realistic solution.

Other white box testing techniques involve loop testing, i.e., make sure that a complete coverage of the loops is conducted. Also the data flow could be another white box testing option. This involves that for each definition of a variable, there must be found a path that will make use of that variable. Such a pair is called DU pair, i.e., definition-association pair. In end user development this represents the testing support for the What You See is What You Test Strategy [159]. Random testing can be successfully applied also to white box testing strategies, i.e., run a program considering different random chosen input values. Nevertheless, the problem of choosing the correct values remains, and also of choosing the best criteria to stop testing.

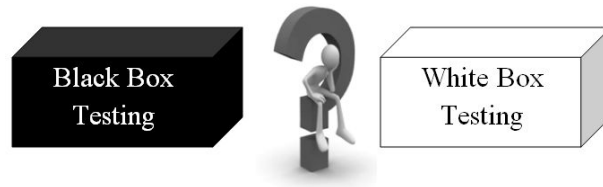


Figure 2.9.: Black Box Testing Vs. White Box Testing

### 2.2.3. Gray Box Testing

There are situations when selecting only black box or white box testing does not represent a feasible approach 2.9. "Test the gray box" [88] represents one of the 293 lessons learned in software testing. An intelligent solution, which combines black box and white box testing strategies, can be gray box testing. In the literature, sometimes gray box testing is denoted as translucent testing [89], due to the fact that the software tester does not have a wide knowledge about the internal behavior of the software system. Several gray box techniques can be applied, e.g., regression testing, matrix testing, orthogonal array testing, pattern testing.

The authors in [89] illustrate a model based testing approach, which combines the parametrized unit testing capabilities of *Pex* [157] with the model based testing environment offered by *Spec Explorer* [26]. *Pex* automatically generates unit test cases, i.e., white box testing technique, which have a high code coverage, for .NET source code; *Spec Explorer* makes use of a model checker to interpret the specifications and to automatically generate model based test cases, i.e., black box testing.

Gray box testing is more suited for testing Web applications. Nevertheless, there is work towards other areas, i.e., real time testing, which combines the benefits of gray box testing, i.e., black box and white box testing, with regression testing and mutation testing [40]. Also, the embedded software domain benefits from the advantages of gray box testing [39].

All the three techniques described so far, black box, white box and gray box, have their own advantages and disadvantages. Deciding upon which of the three software testing techniques is more suited for a software system, depends on different constraints, i.e., the goal of software testing, the resources allocated to the testing process, e.g., time, money, human factor involved, etc. This is the responsibility of the software testing team.

### 2.2.4. Fault Injection

Software fault injection [185] is a common technique used in software testing as a modality to verify the robustness and also the tolerance to different fault categories, of a software application. The technique assumes the insertion of certain bugs into the application, injection of faults, by means of different operators. Among the common used fault injection techniques, we mention:

- *Mutation testing*. It is the oldest fault injection technique. Mutation testing was introduced for the first time in 1971 [47], and since then it has gained more and more attention, achieving today a maturity level. Different types of software systems might use the benefits of mutation testing, since it can be applied with success to different levels of testing and for different programming environments. The technique evaluates the quality of the test cases, by producing small changes, i.e., mutations, at the source code or byte code level, resulting thus new mutated versions of the system under test. The available test suite will then be run against all these mutants and a metric will be recorded to establish the test suite efficiency, against the inserted mutations. Mutation testing can be applied both with white box, e.g., unit testing [79], and black box testing, e.g., mutation of state charts, specifications, code contracts [94, 57]. Different operators were depicted, corresponding to different languages, in order to emphasize different fault types, e.g., arithmetic faults, polymorphism faults, inheritance errors, etc.
- *PIE Technique*, i.e., Propagation-Infection-Execution. According to [183], the technique assumes there exist three particularities that can be used with success to verify the testability degree of a software application:
  - The probability for a given part of the code to be executed;
  - The probability for that part of source code to alter the data behavior;
  - The probability for that modified data behavior to influence the output.
- *EPA Technique - Extended Propagation Analysis*. In EPA, both hardware and software artificial fault injections are conducted and the tolerance degree of the software application is tested. This technique considers only the output values of the application, i.e., correctness is not taken into consideration [184, 182].
- *AVA technique - Adaptive Vulnerability Analysis*. This technique is an extension of the EPA technique, and makes use of assertions in order to detect security properties violations and system survivability [72]. This technique has not achieved its maturity phase and therefore its application domain is somehow restricted.

In the next Chapter, we will present a detailed description, with regard to the mutation testing technique, its advancements and areas of applicability, but also some of the common problems encountered when using this technique.

## **2.3. Conclusions**

Each software testing technique is subject to a design strategy and the wise selection of one technique or another is taken by the software testing team, depending on many factors. Different automated software testing tools were design to encapsulate the introduced strategies, in accordance with the available programming environments, with the usability of the application domain, with the type of software testing that is desired, etc.





# Chapter 3

## Mutation Based Testing

*"A test that reveals a bug has succeeded, not failed."* - Boris Beizer

A natural question, which arises in the software testing domain, is how do we evaluate the test process, i.e., how do we assess the quality of the exiting test suite, what is a good test scenario. One fast answer is that good is strongly connected to the type of testing that is derived, e.g., domain testing, functional testing, stress testing, user testing, regression testing, and so on. Therefore *good* concerns the purpose of the software testing process. A common strategy used to evaluate the software testing process is to achieve code coverage by means of different coverage measures, e.g., requirements coverage, code coverage, test case coverage.

In [9], the authors compare the benefits of using coverage metrics, for measuring the test suite effectiveness, with the advantages of performing mutation testing. As we briefly described in 2.2.4, mutation testing is a fault based testing technique, i.e., changes are made to the original software application, by means of different operators, resulting thus in mutated variants of the original application. The effectiveness is determined using a defined metric, i.e., the mutation score, which is computes the percentage of detected changes from the total number of changes. In his book, [193], Whittaker introduces a software testing fault model, which can be successfully integrated with any type of software application. The author states that two features are essential for the software tester: to have a strong knowledge of the *software application environment* and to be able to proper understand the *capabilities* of the application. According the the author, the software must fulfill four important char-

acteristics, that are subject to an attack, i.e., a test case, and that are essential for the software testing model:

1. The software inputs, which come from the *environment*;
2. The software computes output values, which are later sent to the *environment*;
3. The information of the software application is kept internally, in different *data structures*;
4. With the stored information and the input values, the application runs operations and computations.

The software tester must take care that none of the above characteristics will be broken; if at least one is broken, then the software system will be subject to a failure. The first two features, i.e., input and output values, belong to black box testing area, described previously in Section 2.2.1, whereas the last two are subject to the white box testing domain (See Section 2.2.2).

### 3.1. Introduction

The history of mutation testing goes back to 1970. As a fundamental basis, mutation testing relies on the two concepts: the coupling effect and the competent programmer hypothesis. It was initially designed as a white box testing strategy.

In [47], the authors introduce the concept of the *coupling effect*, which states that simple faults are the basis of more complex faults. According to the authors:

*”Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.”*

The authors derive experiments which prove that the test data used to cover those simple errors, is of significant importance. In order to establish when enough testing has been done, the authors introduce a new concept, *mutation testing* : having as input a program P and its associated test data, i.e., whose adequacy is to be evaluated, the user runs the available tests over the program P. If no error appears, it might be the case that the available test data can not detect the error, i.e., a non adequate test suite is available. Therefore different versions of program P are created, called mutations. These mutated versions differ from the original program through one single inserted error. If the test suite is

run over these mutated programs, two possible solutions might appear: to obtain different results or to have the same results as with the original program. In case the results differ, the authors define the term **dead mutant**, which signifies that the change, i.e., the inserted error, was detected. Otherwise, the authors state the change is **alive**. Two possible explanations exist: the test is not sensitive enough, so that it can detect the inserted change, or the original program is equivalent to the mutated one. The authors conclude that performing a systematic testing method which is able to identify between different versions of the same program, can help the data test generation process of that program.

Regarding the second concept, i.e., the Competent Programmer Hypothesis, it was introduced also in 1978, by the authors in [47]. According to the authors, software developers have the tendency to write software that is *almost correct*. Therefore, the faults that might appear are usually small ones and they can be easily corrected through minor syntactical changes of the code. In [5], the authors present such an example.

These represent the starting point in mutation testing, which were later further defined by Offut [138]. The author extended the coupling effect to the *Mutation Coupling Effect Hypothesis*, which demonstrates that:

*“Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants”* [139].

## 3.2. Mutation Testing Background

Since 1970, the research in mutation testing has evolved considerably. According to the information that resides on the mutation testing repository [84], today there are available more than 400 research papers on mutation testing. Through the diagram in Figure 3.1, we illustrate the basis of mutation testing. In the early stages of mutation testing, only simple errors were taken into account, i.e., simple mutations were derived, each mutated versions differing only through one single change from the original version. As already described previously, in mutation testing we always have as inputs the test suite, i.e.,  $TS$ , the original program  $P$  and a set of mutation operators. A pre-condition is that the test suite must be executed without any error. If  $TS$  executes with success, then according to the types of error against which we want to test the software, we create, corresponding to each mutation operator, the mutated versions of the original program  $P$ , i.e., we create the mutants  $M_1, M_2, \dots, M_n$ .  $TS$  will

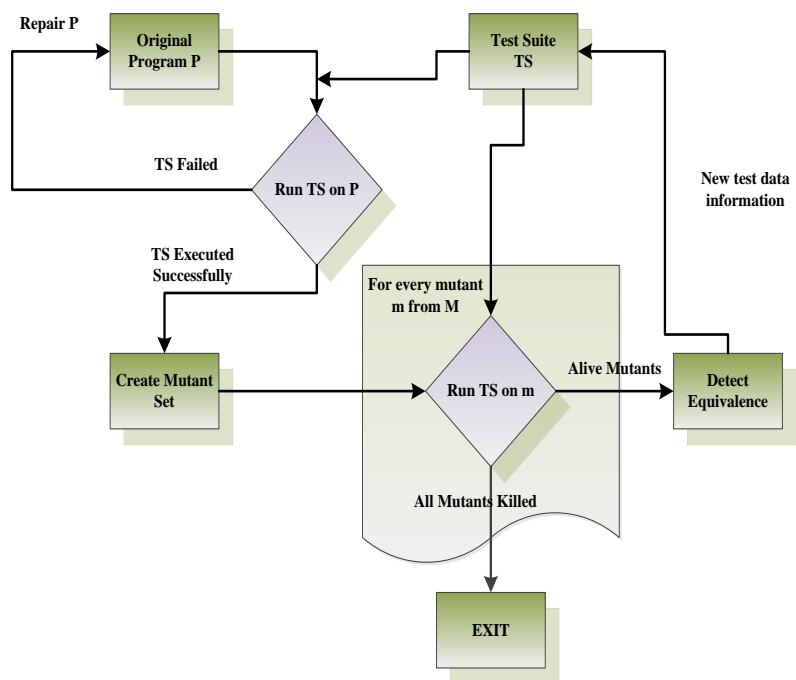


Figure 3.1.: Mutation Testing Process

be executed once more, over the created mutants and the result will be recorded. As defined earlier, when a test scenario from TS succeeds to detect the inserted error, then we say that mutant  $M_i$  was killed, otherwise  $M_i$  is still alive. One known problem in mutation testing is the equivalent mutant problem, i.e., when we do not manage to detect an injected error, due to a semantic equivalence between the mutant and the original program. Therefore, equivalence detection is an intensively investigated topic in mutation testing and different techniques were derived. Nevertheless, it is still an undecidable problem.

In order to keep an evidence on the percentage of changes that are identified by the test suite, the *mutation score MS* metric was introduced:

$$MS = \frac{\text{NumberOfKilledMutants}}{\text{TotalNumberOfMutants} - \text{NumberOfEquivalentMutants}}$$

Another problem in mutation testing is the increased complexity, due to significant amount of time needed to run all the mutants. Therefore, several techniques were developed, in order to reduce the cost of mutation testing.

In traditional mutation testing this set represents the model for the traditional mutation operators. We clarify that in mutation testing, the research literature describes two sets of mutation operators [106]:

1. Traditional mutation operators, which model method level specific behavior, i.e., change arithmetic, relational operators, etc;
2. Class mutation operators, i.e., object oriented, which modify object oriented behavior, i.e., inheritance, access type, polymorphism, etc.

The first mutation testing tool was developed by the authors in [24]. They have implemented a prototype for running mutation testing in FORTRAN. The first set of mutation operators was the Mothra set; it contains twenty-two traditional mutation operators, described in Table 3.1 [90]. Mothra is a mutation environment used for unit testing Fortran software programs, by means of mutation testing. Mothra derives, for each mutated version, a mutant descriptor record in order to describe the injected changes in the source. Besides mutants creation, Mothra is also able to generate test data.

Although achieving complete mutation testing is difficult, this technique is more reliable in revealing faults than any other criteria. In the end, a mutation system, according to Ammann and Offutt [7], can be seen as a complete *"language system, in which programs are parsed, modified and executed"*.

Operator	Description
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source constant replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

Table 3.1.: Mothra Mutation Operators

Therefore, taking into account the problems we encounter in mutation testing, there were designed different approaches to build this type of systems.

### 3.2.1. Mutation Testing Strategies

In what follows we will discuss some of the widely used techniques in mutation testing for reducing the number of generated mutants. If we aim to go beyond the research domain in mutation testing, i.e., benefit from the advantages of mutation testing in industry, then cost reduction techniques are essential. Therefore, several strategies were derived.

The authors in [144], organize these techniques in three categories:

1. *do fewer*: tries to minimize the number of mutant programs that is to be run, with the smallest loss of information. Among the widely known do fewer techniques we enumerate: selective mutation, mutant sampling, mutant clustering, high order mutation.

*Selective mutation* takes into consideration only those mutations that are really different than the others, by removing them from the mutation operators set. Mathur introduced the term of constraint mutation, to depict this constraint in selecting the number of significant mutation operators [195]. Later, this idea was implemented by Offutt, who demonstrated that from the set of 22 mutation operators, described in 3.1, only five of them produce significant mutations: ABS, AOR, LCR, ROR and UOI, achieving a mutation score of 99,5% [140]. Selective mutation has been continuously developed. Recent research towards this direction tries to combine the mutation operators reduction with the cost for detecting the equivalent mutants [124]. Another recent direction in selective mutation is to find a minimum set of mutation operators, through a statical analysis procedure, which considers different subsets of the original set, instead of taking only one small number of mutants [126].

Another *do-fewer* approach is the *sampling* method. Initially, for this technique, mutation testing was applied only to subsets of the mutants set, which were randomly selected [23], e.g., generate the entire set of mutants, then select a random sample subset from the entire set, and remove the rest. This research idea was later extended in the doctoral research of [194], by selecting a sample of a certain percentage from the total number of mutants. The author states that choosing a sample of 10% will reduce the test suite effectiveness with only 16%. Another sampling approach makes use of the Bayesian sequential probability ratio test, in order to derive the sampling [163]. According to their approach, the sampling size is not previously established, but mutants are selected until it is gathered sufficient information which can help in achieving the adequate sampling size.

*Mutant Clustering* does not select random samples from the entire set of mutants, but makes use of the clustering algorithms. After mutant generation, a clustering algorithm - for example, K-means algorithm, agglomerative clustering - is used to organize the mutants into clusters corresponding to the killable test case. From every cluster there is chosen a subset of mutants and used further in the mutation testing process. The idea was first published in [74]. In his thesis, the author demonstrates that using this strategy for selecting a smaller number of mutants, does not alter the value of the mutation score. A recent research towards mutation clustering is conducted by the authors in [83]. They derive a procedure for statically clustering the mutants before test case generation, i.e., generate mutants after the test was generated by taking into account the author's clustering strategy.

When we consider mutant classification, in mutation testing we can encounter first order mu-

tants, i.e., *FOMs*, and high order mutants, i.e., *HOMs*. A FOM results when only one mutation operator is applied and HOMs can appear as a result of two or more mutations, in the same time, i.e., the cardinality is given by the number of inserted faults, which a mutant contains. In [149], the authors run an experiment to evaluate and compare the costs of using FOMs and HOMs. They conclude that mutation testing with FOMs can be more effective, but when using second order HOMs, i.e., two inserted faults in the same time, the number of equivalent mutants will reduce significantly, offering thus a better strategy in mutation testing. The idea of using HOMs to reduce the number of mutants was initially described in [85]. There exist several strategies to combine the first order mutants: *RandomMix*, *DifferentOperators*, *LastToFirst* [152]. A recent direction in obtaining HOMs is by means of genetic algorithms [98]. Along their research, the authors suggest that mutations should be semantically derived and not syntactically. They analyze the relation between the two directions using a multi objective Pareto optimal genetic programming method [151]. Using HOMs raises a challenge towards the above described competent programmer hypothesis. Current research proves that a bug can be fixed only through several changes done in several locations of a program [155]. The authors demonstrate that 90% from the faults discovered after release, are complex faults.

2. *do faster*: aims to split the computational capacity over several computers, by keeping the state information during several runs. *Schema Based Analysis* is one common method for conducting the do faster approach [178, 180, 179]. The technique sums all the mutants into a *metaprogram*, i.e., a metamutant, by means of a program schema (a program schema is a template). This metamutant will be compiled only once. The term mutants schema was defined to encapsulate the metamutant and a metaprocedure set. The metamutant is obtained from the original program and it is used for all mutants representation. It will then be compiled and executed against the available test set. During the execution, this metamutant is instantiated to run as any program from  $N$ , i.e., template instantiation, where  $N$  is a program neighborhood of the original program  $P$ . Mutated statements will have a generic representation. Let us consider, for example, the program from Figure 2.7. Assume we apply the Arithmetic operator replacement in line 10:  $newA = newA - newB$ ; We will have the following mutants:



$newA = newA + newB$

$newA = newA * newB$

$newA = newA / newB$

$newA = newA \% newB$

$newA = newA$

$newA = newB$

The generic representation will be :  $newA = newA \text{ ArithOper } newB$ , where by *ArithOper* we will denote the abstract entity. Along their research towards schema based analysis, the authors observe a performance improvement of more than 300%. Also, unlike for other mutation techniques, this technique allows testing to use the same compiler as the program under test.

Other strategies exist, but they have not been so intensively exploited, e.g., distribute the computational costs on different machines, by means of evolving architectures like network computers [206], vector processors systems [113], concurrent mutants execution in SIMD machines [192]. Recent advancements for reducing the compilation costs, include the use of a Bytecode Translation technique [145, 105] or Aspect Oriented Mutation [11]. In Bytecode Translation technique mutants are generated using the compiled object information of the program to be mutated, thus resulting bytecode mutants, i.e., there is no need for a further compilation of the mutants. Aspect Oriented Mutation uses aspect patches for retaining the methods output; two program executions are required for an aspect patch: first save the results and the original program context and afterwards generate and execute the mutants.

3. *do smarter*: focuses on generating and running a mutant as fast as possible. Several strategies are known: weak mutation, strong mutation, firm mutation. *Strong mutation* depicts the traditional mutation testing process. An extension of strong mutation, in order to implement smarter mutation testing, is *weak mutation*. The notion of weak mutation testing was first introduced in 1982 [73]. According to first principles of this technique, a program is made of a set of components: variable reference, variable assignments, arithmetic, relational and boolean expressions. Unlike traditional mutation testing, where the mutants are analyzed after the entire program execution, in weak mutation a mutant will be inspected immediately after the execution of the modified component of the program, e.g., modified statement. Although many studies were conducted towards weak mutation, its efficiency still represents an open research direction. In their work [141], the authors carry a research study and compare strong mutation against weak mutation. They define four component execution types: *EXpression-WEAK*, i.e., EX-WEAK

first execution of an expression, *Statement-WEAK*, i.e., ST-WEAK first execution of a statement, *Basic Block-WEAK / 1*, i.e., BB-WEAK / 1 first execution of a basic block, and *Basic Block-WEAK / N*, i.e., BB-WEAK / N execution of a basic block after N iterations in a loop. One major advantage is the significant reduction of the program execution time. Nevertheless, the efficiency for the adequate test cases in weak mutation testing is decreasing. The authors conduct the research by recording the strength of the test data, for both situations, i.e., they compute a strong mutation score, *SMS*, and a weak mutation score, *WMS*. They implement a weak mutation system, *Look At Expected Output Not After Return But During Operation*, i.e., Leonardo, in order to derive the experiments. The system makes use of the 22 Mothra mutation operators (described above) and it was applied on simple subroutines, e.g., the bubble sort algorithm, Euclid's algorithm, etc. A significant problem during the experiment was the equivalent mutants detection due to the huge amount of time required to manually identify them. From the conducted experiments, the authors conclude that weak mutation can be successfully used for non critical systems, as a reliable cost reduction technique instead of strong mutation. Critical applications should use a combination between the two techniques.

*Firm Mutation* appears as a mix technique, conceived to overcome the drawbacks from strong and weak mutation testing [196]. The authors consider there are available different timing state parameters, for the program execution, in accordance with the time when the change was derived:

- a)  $t_{change}$ : the behavior when the change is derived;
- b)  $t_{undo}$ : the behavior when the change is turned back and the outputs are compared;
- c) the exact information that is compared.

The authors state that in weak mutation  $t_{change}$  and  $t_{undo}$  appear right before, respectively right after the single execution of a component, while in strong mutation before and after the entire execution. Therefore we can face the case when an introduced error will remain only for a certain number of executions, but not along the entire execution of a program. This behavior is defined as firm mutation and it implements a combination of strong and weak mutation. Several strategies are suggested for  $t_{undo}$  selection point. According to the authors, firm mutation does have certain advantages: a better transparency than the other two techniques, mutation operators selection is not so strict, it is less expensive. Bu some drawbacks do exist, e.g., no solid basis for the proper depiction of  $t_{undo}$ .

### 3.2.2. Mutation Testing Tools

Mutation Testing initially started as a unit testing technique, i.e., white box testing. Along the years, different mutation testing algorithms and tools were developed, i.e., there exist more than 30 mutation testing tools. In what follows, we will offer a brief description on the current available software tools in mutation testing. We present both commercial and open source tools, organizing the information in accordance with the corresponding programming environment for which mutation testing was applied.

1. *PIMS* [22]. It is an early stage mutation tool for Fortran. PIMS accepts the test cases from the user, executes them over the mutants and computes how many mutants were killed. Several other Fortran mutation testing tools were developed afterwards, but the most known of them is *Mothra*.
2. *Mothra* [90]. This is a set of tools, for deriving mutation analysis and testing in Fortran.
3. *Proteum* [45]. First tool for C source code mutation unit testing. The tool supports interface mutation, i.e., interface mutation derives mutation in order to consider components integration errors [46]. Proteum was constantly updated, evolving to a set of tools [110], which can support mutation testing for finite state machines, i.e., *Proteum/FSM* [58], for specifications based state charts, i.e., *Proteum/ST*, for Petri Nets, i.e., *Proteum/PN*, for AspectJ programs, *Proteum/AJ* [61]. The tools are available for download, via email [111]. There were developed specific mutation operators for each of these tools, e.g., there exist 9 finite state machine mutation operators, 17 state chart feature operators, for Petri Nets there are implemented 11 mutation operators, etc. Other tools were derived for implementing C code based mutation testing [6, 65].
4. A commercial tool, combining both C and C++ is *Insure++* [150] from Parasoft. The tool performs memory analysis and error detection, by instrumenting the source code. It has incorporated a tool for analyzing the total code coverage and one for analyzing the memory usage. A particularity of this tool, is that it works to create equivalent mutants, instead of trying to reveal error prone ones. The tool has patented the Source Code Instrumentation technology.
5. Among the commercial C based mutation testing tools we enumerate *PlexTest*, which conducts selective mutation [80], *Certitude*[172]; the tool combines mutation testing techniques with static analysis to measure the quality of HDL design.
6. For Java, there is an impressive number of open source mutation testing tools:
  - *Jester*: It is an open source testing tool, that runs the traditional mutation testing strategy [78, 123]. Due to its not so friendly environment configuration, an integration with the

Eclipse Framework was implemented, making Jester available as an Eclipse Plugin [100]. But the tool has also a command line interface. It supports only traditional mutation operators. Jester generates and compiles mutants and then runs them using the available test cases. The tool is subject to performance problems.

- *JavaMut*: The tool does not have fast performances, i.e., each mutant is separately compiled. However, a download link is not available on the Web [31].
- *MuJava*: Many research studies rely on this tool. It implements both traditional and class level mutation operators. For mutant generation, the tool combines traditional generation of mutants, i.e., by directly changing the source code, with two cost reduction techniques: byte code translation and mutant schema [145, 105]. An automated mutation testing plugin was released for an easy integration with the Eclipse framework, i.e., Muclipse [170]. Moreover, in Muclipse also weak mutation was implemented.
- *Jumble*: The tool is open source and it can be downloaded directly from the Internet [177]. It was designed for an industrial usage, implementing byte code translation, in order to reduce the costs from mutation testing [79]. Nevertheless, the tool has just a simplified set of traditional mutation operators.
- *RI/Response Injection*: The tool concentrates on the object oriented aspect of a program. It mutates method output values, primitive types [14]. It can not mutate objects that are sent as parameters to a method. Currently, there is not available any download information.
- *MuTMuT*: It is a mutation testing tool for multithread code, i.e., *Mutation Testing of MUltiThreaded code*. The tool reuses the information collected when performing the test execution on the original program to be verified [66].
- *Javalanche* : It is an open source framework for Java mutation testing, which claims to overcome the equivalent mutant problem [166, 167, 69] and also to treat the efficiency problem from mutation testing. Javalanche uses dynamic invariants, assessing an impact for the created mutations in order to check invariant violations.
- *PIT*: Is a tool which makes use of the code coverage and byte code information [33]. PIT integrates both JUnit and TestNG, it has a command line interface, and several PIT plugins were developed: for Eclipse, IntelliJ, Gradle build automation environment [34]. It implements traditional mutation operators and works on byte code level.

- *Judy*: Implements a novel approach in mutation testing, i.e., the *FAMTA* (fast aspect-oriented mutation testing) algorithm [108, 109]. During mutant generation, a collection of mutants, but also meta-mutants are created. Meta-mutants allow mutant generation while the source code is modified. The tool can compile with Java versions greater than 1.4, up to 1.6, and it benefits from a fully automated mutation testing process; it has its own predefined set of 16 mutation operators.
- *Testooj*: It is an open source, easy to use, Java testing tool, developed by the Alarcos Research Group of the Castilla-La Mancha university [181]. It supports a command line interface and also a jar for an easy integration with the Eclipse framework. The tool integrates MuJava, for running mutation based unit testing. Test cases are generated based on regular expressions.

But also commercial tools, which claim to solve the problems of existing open source tools, are available, e.g., *BACTERIO Mutation Test System* is a tool developed during a doctoral program [28, 112]. The tool supports JUnit test cases, but also UISpec4J, i.e., library for Java and Swing GUI testing [121]. Bacterio encapsulates both mutant reduction strategies - selective mutation, mutant sampling, and also techniques for reducing the cost execution - mutant schema, parallel execution, byte code translation.

- An interesting mutation testing framework is *ExMan*. It is a generic automated mutation analysis tool, which derives a comparison between different quality assurance strategies, based on model checking, static analysis and testing [17]. ExMan allows integration for different mutation generator tools, i.e., in order to mutate Java or C source code; also, the tool is available for download [18].

There are available other mutation testing tools for Java, but we have chosen to enumerate the ones that were often used throughout the mutation testing research history.

Mutation testing goes further and other programming environments benefit from mutation testing, to assess the quality of the testing process:

7. For database mutation testing, some of the most known tools include *SQLMutation* [176], *JDAMA* [209]. *SQLMutation* generates mutants for SQL based application, by mutating SQL queries. It is an open source tool and can be run online, from the Web interface, or directly integrated in Web Service based applications. *JDAMA* is a mutation testing tool used to check the JDBC interface communication between a database and a Java based environment; it functions

on byte code level.

8. The author of *Jester* (performs Java based mutations), also implemented two other tools:
  - *Pester*, for mutating Python source code with PyUnit unit test,
  - *Nester*, for C# source code mutation.
9. *CREAM* is another tool for mutation testing C# code [50].
10. *MutateMe* is a tool for running mutation testing on PHP applications. It is an open source project, available on GitHub [19].
11. For mutation testing AspectJ source code, the authors in [44] have developed the *AjMutator* environment.
12. Tools were developed also for mutation testing Ruby source code [162]. *SESAME* (Software Environment for Software Analysis by Mutation Effects) is a multi language mutation unit testing framework: it can be applicable to assembly and procedural languages, like C and Pascal, but also to a data flow language, i.e., Lustre.

As we can observe, mutation testing has not only been used for determining the test suite efficiency, but also it has been applied on the specifications [94], or as a reliable method for test case generation from specifications [51]. Another interesting applicability is in regression testing [52], where mutation has been used as a test case prioritization strategy. Also, for test case reduction, the authors in [143] develop a Ping-Pong strategy, based on mutation testing. Another test size reduction strategy is proposed by the authors in [153]. They use the mutation score to select the test cases, but taking care to preserve the quality of the test pool. Based on the computed mutation score, the authors apply a greedy algorithm to select only those test cases that kill the majority of the mutants. They apply their approach on three open source projects, i.e., *Jester*, *JTopas* and *JFreeChart*, in order to demonstrate its efficiency.

# Chapter 4

## An Efficient Test Suite

*"A good threat is worth a thousand tests."* - Boris Beizer

The research presented here is based on the work published in [134, 129, 135, 136, 137]. As described in Chapter 3, one applicability of mutation testing, beside assessing the quality of the existing test suite, is to contribute to the test case generation process. Through our research, we aim to offer an industry reliable automated test case generation solution, combining the benefits of mutation testing, with constraint based programming and software debugging.

The goal of our research is to derive an efficient software testing process, improving the initial test suite. We contribute to mutation test case generation, based on mutation testing and equivalent mutants elimination. Starting from the evaluation of an available test suite, by means of mutation testing, we identify the non treated situations and try to minimize the test case generation procedure. We do this through the use of the constraint satisfaction paradigm and distinguishing test cases procedure, reducing thus the equivalent mutants that result from the mutation process. In the first experiments we conducted, we have started by evaluating how good the testing process was derived in an enterprise environment. The initial results encouraged us to go further on with the approach and overcome the drawbacks we encountered. In this situation, the next step was to improve the testing pool, thus developing new test scenarios able to detect a misbehavior or to treat uncovered source code. We made use of the obtained results and continued our research towards a new direction. Additionally to measuring the quality of the test suite, we have extended the research to test case generation based on

distinguishing test cases and the constraint representation of mutants [204, 130, 201]. Hence, the focus of our research is **test case generation**, and, as an extension to our approach, we strive to eliminate a high percentage of the **equivalent mutants**.

## 4.1. Early Stage of the Research

Our research started initially as an applied project in industry, in collaboration with *Siemens AG Austria*. The goal was to depict a reliable method for evaluating the test suite quality. As mutation testing proved to be one reliable strategy in fault detection, but also in assessing the quality of a test suite, we have chosen to configure the project environment in order to apply mutation testing [9, 7].

The initial research environment assumed an Enterprise JavaBeans (*EJB*) server-side component architecture [122], which was run and deployed over the JBoss Application Server [37]. As EJB architectures are distributed Java applications, we have chosen to run the research with MuJava mutation testing tool [105]. MuJava implements the two steps of the mutation analysis process: automatically generates mutants and then runs the test sets over the mutant source files.

### 4.1.1. MuJava

Along our entire research activity, we decided to work with MuJava. We have implemented an extended version of the tool [134], in order to support compilation with standard Java development kit 1.5. Also, integration with JUnit test cases was implemented. Thus, it can be derived an evaluation of the test suites (by comparing the mutation scores), in order to establish a prioritization of the test cases (run first the most important tests in the testing process). Due to these new add-ons, the tool supports a command line version, therefore an easy integration, e.g., into a daily build process, is now possible.

MuJava implements a combination of the schema based approach with selective mutation. Two sets of mutation operators were designed, in order to derive mutations:

- method level mutation operators. These are traditional mutation operators and they produce traditional mutations, e.g., arithmetic replacements, deletion, insertion. Totally, the tool implements the 12 traditional operators from Table 4.1 [107].



Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Table 4.1.: MuJava Method Level Mutation Operators

- class level mutation operators. There are 29 class level mutation operators, which mutate object oriented features, e.g., inheritance, polymorphism, encapsulation, Java specific language properties, like changing a method access type. They can be observed in Table 4.2.

#### 4.1.2. Mutation of EJB Components

Enterprise JavaBeans components are server side components which contain the business logic of an enterprise application, mainly used for developing large and distributed applications. Usually, an enterprise application contains a significant amount of data that is accessed concurrently by clients, i.e., users. Therefore the enterprise beans manage different tasks concerning the communication between the client and the server, e.g., interaction with the client, information retrieval for the client maintaining, keep data information on a database and communicate with the server.

As mutation involves modifying the EJB source code, for each change of the business logic a new deploy is needed on the server, i.e., the users who interact with the server should use the updated business logic. These type of systems must be robust and should present a good scalability. In order to derive better methods for testing server side components, we aimed to establish an indicator of the test process quality by means of mutation testing. The experiment was run in collaboration with Siemens AG Austria.

We lead the research using three EJB based applications and an internal framework from Siemens AG Austria. During the research we used JBOSS application server as a deploy server [37]. On each

Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHI	Hiding Variable Insertion
	IHD	Hiding Variable Deletion
	IOD	Overriding Method Deletion
	IOP	Overriding Method Calling Position Change
	IOR	Overriding Method Rename
	ISI	<b>super</b> Keyword Insertion
	ISD	<b>super</b> Keyword Deletion
	IPC	Explicit Call of a Parent's Constructor Deletion
Polymorphism	PNC	New Method Call with Child Class Type
	PMD	Parameter Variable Declaration with Parent Class Type
	PPD	Parameter Variable Declaration with Child Class Type
	PCI	Type Cast Operator Insertion
	PCD	Type Cast Operator Deletion
	PCC	Cast Type Change
	PRV	Reference Assignment with Other Comparable Variable
	OMR	Overloading Method Contents Replace
	OMD	Overloading Method Deletion
	OAC	Arguments of Overloading Method Call Change
Java Features	JTI	<b>this</b> Keyword Insertion
	JTD	<b>this</b> Keyword Deletion
	JSI	<b>static</b> Modifier Insertion
	JSD	<b>static</b> Modifier Deletion
	JID	Member Variable Initialization Deletion
	JDC	Java Supported Default Constructor's Creation
	EOA	Reference Assignment and Content Assignment Replacement
	EOC	Reference Comparison and Content Comparison Replacement
	EAM	Accessor Method Change
	EMM	Modifier Method Change

Table 4.2.: MuJava Class Level Mutation Operators

Java bean class, from the three projects, there were applied traditional and also class level mutation operators. In order to determine the set that may detect the majority of faults, for the class level operators we run several experiments choosing different sets of mutation operators. The following strategy was followed:

1. Development of the EJB test applications, followed by project configuration in order to achieve the Siemens software configuration.
2. Generate mutants for each test application. For the newly developed test applications, mutation was conducted on a selected number of bean files, working with a reduced test suite.

- For each application, test cases are created and run over the original version.
  - Traditional mutation operators and class level operators are applied to the original Java bean source code implementation. Mutants are created, corresponding to the chosen mutation operators set.
3. For each created mutant, do a re-deploy of the application and then execute the test set against the mutated deployment file. It is a fully automated process.
  4. Record the mutation score for each experiment.

Figure 4.1 depicts the procedure described above. In Figure 4.2 we can observe the chart with the actual mutation score compared with the line code coverage, measured with Cobertura [32].

This strategy is prone to some limitations, due to the significant number of generated mutants. For a large number of mutants the complete process of redeploying each mutant and running the test suites can lead to an increasing time and to high costs of execution. Also, the application server may suffer from bottlenecks. A solution for reducing the cost of mutation analysis is to establish an effective set of mutation operators and select a well determined pool of test sets that will reveal the majority of a set of faults. A second solution is to inject faults only for the business methods that interfere in the communication between the client and the server, i.e., mutate only specific parts of the source code.

Due to the huge amount of data and time invested into this research experiment, but, also, due to the lack of financial resources, the project was temporarily closed.

## 4.2. An Industry Setting

We are still trying to answer our question : Is Mutation Testing Scalable for Real-World Software Projects?

We aim to assess the costs of applying mutation testing on a real-life software system. We have been investigated the following aspects in order to answer the question whether mutation testing is worth the effort for a real life software project or not:

- The time required for mutation testing,
- The results of mutation testing compared to coverage analysis,
- The issues encountered in setting up and running the selected mutation testing tools.

In what follows, we briefly present the working environment configuration.

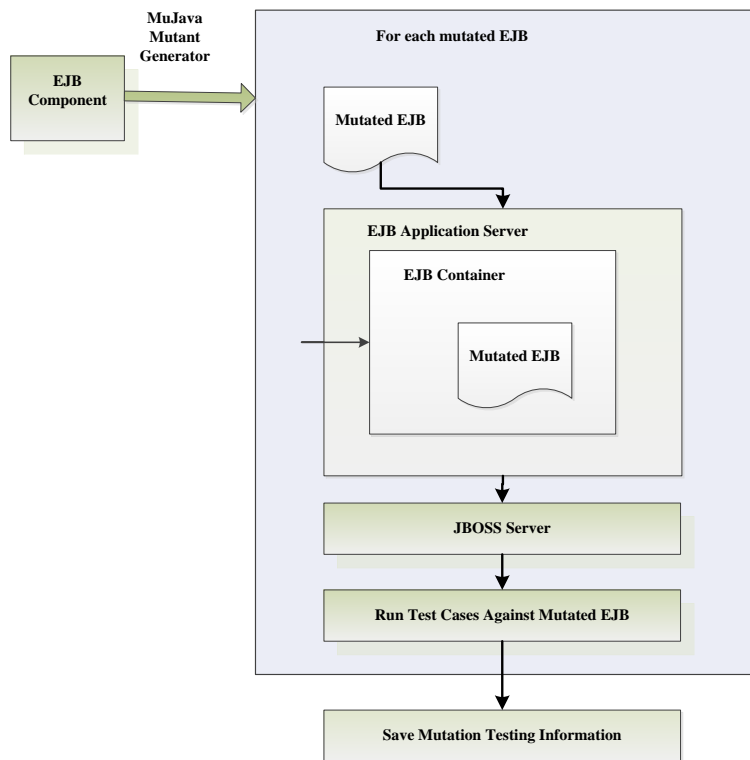


Figure 4.1.: EJB Mutation Analysis

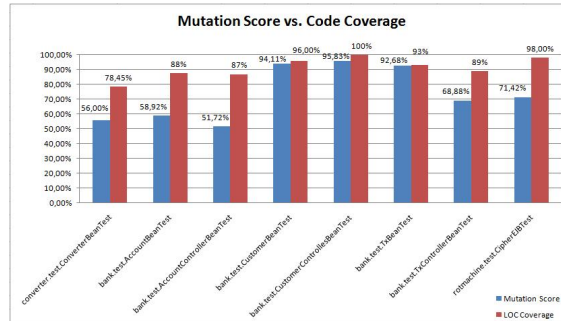


Figure 4.2.: Mutation Score

### 4.2.1. Environment Configuration

We aimed to use mutation testing on Eclipse [75], a widely known and large open source project that shows many parallels to commercial and industrial software projects, especially those developed on the basis of the Eclipse application framework. We retrieved the source code Eclipse Release Build 3.0, from the Eclipse repository [75].

In Table 4.3, the versions and configuration parameters of the tools and test objects used throughout this section are described. All the work presented herein was conducted using the virtual environment Oracle VM Virtual Box. The virtual machine is configured to run on Windows XP SP2 operating system, on an Intel Core 1.73 GHz with 2 GB of RAM. For the Java Virtual Machine, we compile and run all the files involved in the research with version 1.6, update 24. We have chosen to work within a virtual environment, in order to offer a fast portability and also an easy management for our research. Our goal is to derive a fully automated process, for all the Eclipse plugins, which will run over a predefined period of time, on different architectures.

Tool / System	Version	Location/Comment
Eclipse	3.0	[75]
MuJava	3	[105]
Jumble	1.1.0	[177]
Javalanche	0.3.6	[81]
Clover	3.0.2	[10]
EclEmma	1.5.1	[54]

Table 4.3.: Overview environment configuration

We apply three of the most widely used mutation tools: MuJava [205], Jumble [79] and Javalanche [166]. We run the mutation testing technique and then compare the results with the code coverage information provided by Clover [10] and EcEmma [54].

#### 4.2.2. Applied Mutation Tools

For computing the mutation score metric, we take into account, throughout the research, the following mutation tools:

1. **MuJava:** MuJava is a Java based mutation tool, which was originally developed by Offutt, Ma, and Kwon [104]. Its main three characteristics are:
  - Generation of mutants for a given program.
  - Analysis of the generated mutants.
  - Running of provided test cases.

Due to the newly implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration into the testing or debugging process. Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. It works with both types of mutation operators:

- Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
- Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

MuJava was not designed to work with JUnit test cases, nor to compile with Java versions greater than 1.4; i.e., Java development kit 1.5 or 1.6.

As we selected MuJava for deriving mutation testing, we have implemented several add-ons to support JUnit test cases and partial mutation of Java source files compiled with JDK 1.5 and 1.6. We take into account both the traditional mutation operators, i.e., the method level, and the class level ones. MuJava comes with a graphical user interface.

2. **Jumble** It is a class level mutation tool. Moreover, this tool supports JUnit 3 and, recently, it was updated to work with JUnit 4. Similar to MuJava, just one mutation is possible at a time,

over the source code under test. First, the tool runs all the tests on the original, unmodified, source file and checks whether they pass or not, recording the time necessary for each test. Then, it mutates the file according to different mutations operators and runs the tests again. The process is done when all the mutations have been tested. Unlike MuJava, Jumble is able to mutate constants.

3. **Javalanche** This mutation testing tool should resolve two major problems in mutation testing: efficiency and equivalent mutants problem. Javalanche works on byte code and can mutate very large programs. The authors resolve the problem of equivalent mutants by assessing the impact of mutations over the dynamic invariants [167]. According to the authors of the tool, Javalanche has a unique feature. The tool is able to rate the mutations in accordance with their impact on the behavior of program functions, i.e., the greater the impact of an undetected mutation is, the lower the possibility of an equivalent mutant.

We have chosen to carry the research using the above described tools, taken into account their usage inside the experiments conducted in the mutation testing area.

### 4.2.3. Research Strategy

In this section, we present the first results of our research, by taking into consideration the three aspects that we follow in our research work: time, mutation testing results, usage of the JUnit tests provided on the Eclipse repository, and finally we describe the issues encountered in setting up and running the different mutation testing tools. We have taken the following steps:

1. Check-out the project from the Eclipse repository;
2. Run the plugin test cases (the ones associated to the checked out project);
3. Download and install the coverage and mutation tools;
4. Set all the necessary class paths for each tool;
5. Run the tools over the original project and record the results. This step is the one that consumes most of the time, i.e., approximately 1 month and a half in case of our chosen plugin project. This is mainly due to the different compilation exceptions encountered (for the compilation and tools running tasks one human resource was allocated).

As the research procedure is the same for each of the Eclipse plugins, we run the first research steps with the Eclipse Java development tools Core project. The JDT [82] provides the tool plugins that implement the Java IDE, which supports the development of any Java application, including Eclipse plugins.

The JDT Core project, *org.eclipse.jdt.core*, has associated three test projects:

1. *org.eclipse.jdt.core.tests.builder*
2. *org.eclipse.jdt.core.tests.compiler*
3. *org.eclipse.jdt.core.tests.model*

## Time

Concerning the time necessary to derive this research, we have to take into account:

- The time necessary to configure the tools; the effort estimated was of approximately one week;
- The mutants generation time; for the selected plugin, it took us between 6 to 8 hours, i.e., a full working day;
- The time needed to run the test cases against the set of mutants. This is the most significant one, as we have a huge number of mutants.

## Mutation Results

For each test project from Table 4.4, we computed the total number of initial test cases  $\mathbf{No}_{TC}$ , the initial time  $\mathbf{T}_{orig}$ , in minutes, needed to run the tests, and the success rate  $\mathbf{S}_{rate}$  which tells us the percentage of tests that initially passed.

In Table 4.5, we show the detailed mutation testing information for one of the three test projects, *org.eclipse.jdt.core.tests.compiler.regression*. We record the number of generated mutations  $\mathbf{No}_{Mut}$ , the necessary time for generating all the mutations,  $\mathbf{T}_{Mut}$ , the mutation score  $\mathbf{MS}$  and the total time for running the tests over the mutants, i.e.,  $\mathbf{T}_{TC_{Mut}}$ . MuJava generated 123 class mutants and almost 31 000 method mutants, in approximately 360 minute, i.e., 6 hours. We estimated the total time for running all the generated mutants; we did not run all the method level mutants, due to the increased time complexity. The average mutation score recorded was around 65%. The computed mutation



score, for MuJava, is the average of all the mutation scores computed for each run of the plugin, in accordance with the selected mutants.

As it can be observed from Table 4.5, we were not able to obtain any mutation points for Jumble and Javalanche. By  $T_{NoTC}$  we denote the total number of test cases from a specific test project. In Table 4.6, we record, the success rate for the three plugin projects, after running all the test cases from each project, using the coverage tools. Concerning the types of code coverage recorded by the tools we have selected for our research, we know that:

- *Clover* measures statement, branch and method coverage;
- *EclEMMA* computes class, method, statement and basic block code coverage.

In the research conducted so far, we have reported the mutation score to the statement coverage level. Nevertheless, further code coverage measures might be taken into account.

### Encountered Issues

Up to now we were not able to generate mutants, for the JTD Core project, with Jumble or Javalanche. The main problem we have encountered was to run the test cases as plugins test, using our three mutation tools. Besides time consuming, the generation of mutants proved to be also very complex.

Concerning the first mutation tool, MuJava, there are some limitations we have to take into consideration:

- MuJava is not able to generate any mutants in case of constants (it does not mutate constant values);

Test Project	NoTC	T <sub>orig</sub>	S <sub>rate</sub>
org.eclipse.jdt.core.tests.builder	79	161.312	100.00 %
org.eclipse.jdt.core.tests.compiler	2542	16.203	100.00%
org.eclipse.jdt.core.tests.regression	2622	387.735	100.00%
org.eclipse.jdt.core.tests.eval	350	65.562	100.00%
org.eclipse.jdt.core.tests.dom	1584	136.437	100.00%
org.eclipse.jdt.core.tests.formatter	486	21.109	100.00%
org.eclipse.jdt.core.tests.model	2084	293.782	100.00%

Table 4.4.: Eclipse JUnit Test Results

Tool	NoMut	T <sub>Mut</sub>	MS	T <sub>TCMut</sub>
MuJava	123/30947	174.69 min/185.7min	app.65%	est. 2 months
Jumble	-	-	-	-
Javalanche	-	-	-	-

Table 4.5.: Mutation Testing Information per Mutation Testing Tool

Project	T <sub>NoTC</sub>	Clover	EclEMMA
org.eclipse.jdt.core.tests.builder	79	100.00%	100.00%
org.eclipse.jdt.core.tests.compiler	16287	100.00%	100.00%
org.eclipse.jdt.core.tests.model	8639	99.97%	100.00%

Table 4.6.: Success Rate

- Also, missing statements are another limitation of the tool. We are not able to generate mutants, by statement deletion, nor insertion;
- In case of multiple bugs in one statement, the MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program (this limitation is however easy to overcome);
- In order to support execution of JUnit tests, the nullary constructor has to be added to each test class file. Also, the private methods *setUp()* and *tearDown()* must have public access;
- The last problem, regarding mutation, is that in some cases equivalent mutants were generated.

As we can observe in Table 4.5, the majority of mutants generated with MuJava was produced by the method mutation operators. From the large pool of generated traditional mutants, the three most commonly encountered were:

1. AOIS, i.e., Arithmetic Operator Insertion, with 13654 mutants,
2. LOI, i.e., Logical Operator Insertion, with 4698 mutants, and
3. ROR, i.e., Relational Operator Replacement, with 3980 mutants.

Javalanche is of real interest in our approach, as it should deal with the equivalent mutant problem. This would allow us to reduce the high number of generated mutants and thus reduce the effort. Therefore, we further hope to run the research and, together with the people involved in Javalanche development, come with a solution.

We have to mention that, on small and simple projects, i.e., no more than 200 lines of code and

which have the test sets located in the same project as the mutated classes, we were able to configure and successfully make use of Jumble and Javalanche.

In what follows, we briefly describe the experience recorded for configuring and running the mutation tools we have used for this stage of our research. We denote by **Tool<sub>Config</sub>**, i.e., tool configuration, the knowledge accumulated when configuring all the paths; by **Mut<sub>Gen</sub>** we present the mutants generation step and by **Running<sub>TC</sub>** the observations when running the mutants.

### 1. *MuJava*

- **Tool<sub>Config</sub>**: The graphical user interface, but also the command line version, are intuitive and easy to use.
- **Mut<sub>Gen</sub>**: The tool must have access to the class files corresponding to each file to be mutated; also, the user can select which mutation operators to apply, both from the set of traditional mutants and also from the class level ones.
- **Running<sub>TC</sub>**: MuJava requires the tests to have the nullary constructor. Also, the methods `setUp()` and `tearDown()` must have public access (default is protected). This was time consuming, as we had to update all the test classes with the nullary constructor and the public access for the two methods.

Both for generating the mutants and then running them, it took us a lot of time. A solution may be the integration into build script (like Apache Ant), which is to be run on a monthly basis without any user interaction.

### 2. *Jumble*

- **Tool<sub>Config</sub>**: A `readme.txt` file is available, where the steps to take are quite easy to follow. Nevertheless, after following the instructions and setting the classpath, we were not able to derive a running configuration.
- **Mut<sub>Gen</sub>**: We were not able to get any mutants, due to execution errors.
- **Running<sub>TC</sub>**: Not reachable.

### 3. *Javalanche*

- **Tool<sub>Config</sub>**: The `javalanche.xml` file has to be copied to the current user directory, where it resides the project to be mutated. Then, it follows an easy configuration phase, i.e., in the xml configuration file, change the paths towards the installation folder and the working project directory.

- **Mut<sub>Gen</sub>**: Javalanche instruments the byte code and then needs to take control over the test execution. For test execution, Javalanche relies on JUnit test suites. If a test suite is not supplied, Javalanche just mutates the code, but it can not take over the control of the test execution.
- **Running<sub>TC</sub>**: We did not manage to reach this step.

Regarding the two coverage tools we have used, we found it easy to setup and integrate them into a daily build script (e.g., Apache Ant), but also as an Eclipse plugin (both Clover and EclEmma are available, for integration, as Eclipse plugins).

#### 4.2.4. Conclusion

Mutation testing is an efficient method to detect errors inside the software projects. Unfortunately, the available open source mutation testing tools we have used so far in our research, have proven to take a lot of time in order to derive all the configuration settings. Although mutation testing can assist in revealing many errors, not all of them represent real actual software failures. The problems, mostly encountered with this technique, are the complexity required to derive the process (as higher the number of generated mutants is, as higher the computation time), but also the equivalent mutant problem.

Each of the above described mutation tools requires different configuration settings. The time effort that we have invested just in configuring each tool and then deriving the entire mutation testing technique was of several months. From the results obtained during the research work, we state that mutation can be regarded as a good software quality metric, but special attention should be given to the drawbacks presented above and, also, to the total amount of time. Meanwhile, setting up the configuration and then running the code coverage tools has proven to be easy to handle. Based on other previous works, we compare the results given by code coverage with the ones obtained from the mutation testing process, and we conclude that, even by 100% code coverage, it is often the case that the test suite is poor with respect to the inserted mutations, i.e., low mutation score.

What distinguishes our work from the previous ones, e.g., [8, 109], is the fact that we take a huge, well known and widely used software project, i.e., Eclipse, and start to record different software metrics. The most important of them is the mutation score metric. For Eclipse we can track the faults database and, therefore, derive a realistic and practical report of the mutation testing technique,

together with other quality software metrics, in order to depict real software bugs. One of the work we report to is the research conducted by Zeller [154].

### 4.3. Mutation Based Test Case Generation

The work presented in the following section was published in the [129] journal paper.

#### 4.3.1. Introduction

Debugging, i.e., detecting, locating, and correcting a bug, in a program is considered a hard and time consuming task. This holds especially in the case of software maintenance where the programmer has little knowledge of the program's structure and behavior. Today's research activities mainly focus on the fault detection part of debugging. Automated verification and testing methods based on models of the system and specification knowledge have been proposed. Little effort has been spent in automated fault localization and even less in fault correction. There has been also no research activity bringing together testing and fault localization and correction, except for the fact that test cases are used for debugging. However, to the best of our knowledge there is no work that analyzes the impact of test suites on the obtained debugging results.

We aim to contribute to the test case generation problem in order to improve the results obtained in automated debugging, based on a model of the program. In particular we show how test cases can be generated to distinguish potential diagnosis candidates. A potential diagnosis candidate, or diagnosis candidate for short, is a statement that can explain why test cases fail. A diagnosis candidate does not need to be the real bug. But the real bug should be included in the list of diagnosis candidates delivered by an automated debugger; we focus on method level debugging or on small programs (up to 1000 LOC). This drawback is due to the fact that in our approach we use the full semantic of the program to encode the debugging problem.

We now consider the following code snippet to illustrate our combined debugging and testing approach. We use this small program to avoid introducing too much technical overhead and to focus on the underlying idea.

1. `i = 2 * x;`
2. `j = 2 * y;`
3. `o1 = i + j;`
4. `o2 = i * i;`

We cannot say anything about the correctness of such a code fragment without any additional specification knowledge. Let us assume that we also have the following test case specifying expected outputs for the given inputs:  $x = 1$ ,  $y = 2$ ,  $o1 = 8$ ,  $o2 = 4$ . Obviously, the program computes the outputs  $o1 = 6$  and  $o2 = 4$ , which contradicts the given test case. Therefore, we know that there is a bug in the program and we have to localize and correct it. At this stage we might use different approaches for computing potential fault locations. If we use the data and control dependencies of the program, we might traverse the dependencies from the faulty outputs to the inputs backward. In our example, we are able to identify statements 1, 2, and 3 as potential candidates.

A different way to locate bugs is to consider statements as equations and to introduce correctness assumptions. If the test case together with the assumptions and the equations are consistent, the assumptions stating incorrectness of statements can be used as potential diagnosis candidates. Consider, for example, Statement 1 to be faulty and all other statements to be correct. As a consequence, Statement 1 does not determine a value for variable  $i$ . However, from Statement 4 and the test case we can conclude that  $i$  has to be 2 (if assuming only positive integers). Hence, we are able to compute a value for  $o1$  again, which contradicts the given test case. Therefore, the assumption that Statement 1 is a diagnosis candidate cannot be correct. Note that even when assuming that  $i$  might be -2, we are able to derive a contradiction. It is also worth noting that the described approach for generating diagnosis candidates can be fully automated.

We are able to apply this technique for making and checking correctness assumptions for all statements and finally obtain statements 2 and 3 as diagnosis candidates. For larger programs we might receive a lot of potential diagnosis candidates and the question of how to reduce their number becomes very important. One solution is to ask the user about the expected value of intermediate variables, like  $i$  or  $j$ , for specific test cases. Such an approach requires more or less executing the program stepwise. Moreover, especially in the case of software maintenance where a programmer is not very familiar with the program answering questions about values of intermediate variables, this can hardly be done. Therefore, we suggest an approach that computes test cases, which allow to distinguish the behavior

of diagnosis candidates. More specifically, we are searching for inputs that reveal a different behavior of diagnoses candidates. Such test cases are called distinguishing test cases [201]. In case no such distinguishing test case can be computed, the diagnosis candidates are, from the perspective of their input output behavior, equally good.

What prevents us from applying the approach of distinguishing test cases, in order to differentiate diagnosis candidates, is the fact that the fault localization approaches only give us information about the incorrectness and correctness of some statements, but not about the correct behavior of potentially faulty statements. Hence, computing test cases is hardly possible. In order to solve this problem we borrow the idea of mutation or genetic-based debugging [189, 43]. Mutants, i.e., variants of the original program, are computed and tested against a test suite. The mutants that pass all test cases are potential diagnosis candidates. Computing mutants for all statements and testing them against the test suite is very time consuming and some techniques for focusing on relevant parts of the program have been suggested. In our case we are able to use the diagnosis candidates for focusing on relevant parts of the program. Hence, when finding a mutant for a diagnosis candidate that passes all test cases, we do not only localize the bug but also state a potential correction.

For our example program we might obtain two mutations  $m_1, m_2$  for statements 2 and 3, e.g., :  $m_1 = 2. \quad j = 3 * y$  and  $m_2 = 3. \quad o1 = i + j + 2$ . Obviously there are more mutations available, but now, for illustrating the distinguishing test cases, we use only these two. A distinguishing test case for these mutants is  $x = 1, y = 1$ . Mutant  $m_1$  computes the value 5 and mutant  $m_2$  the value 6 for the output variable  $o1$ . If we know the correct value of  $o1$ , we are able to distinguish the two mutants. From this example we conclude that we are able to distinguish diagnosis candidates using distinguishing test cases. In what follows we will provide experimental evidence that the approach is feasible and that it leads to a reduction of diagnosis candidates when applied to general programs.

We introduce and discuss the approach and tackle the research question regarding the approach's usefulness with some exceptions. The programs used for the evaluation are small programs and they mainly implement algebraic computations. Moreover, we do not handle object-oriented constructs. However, we do not claim to answer the research question completely. We claim that the approach can be used for typical programs comprising language constructs like conditionals, assignments, and loops. The structures of the used programs are similar to those of larger programs or, at least, we do not see why any big differences should exist. Another argument is that the approach is mainly for debugging at the level of methods comprising a smaller amount of statements, where our technique is definitely feasible.

We briefly recall the basic definitions, introduced in Chapter 2.1. This includes the definition of test cases and test suites, the debugging problem, as well as the definition of mutations. The direction of our research deals with debugging based on models of programs, which are written in a programming language. We have conducted the research under the assumption of an imperative, sequential assignment language  $\mathcal{L}$  with syntax and semantics similar to Java, ignoring all object-oriented constructs and method calls. We further restrict the data domain of the language to integers and booleans. In Figure 4.3 we present an example program, which serves as a running example. The program implements the binary search algorithm - the algorithm retrieves the index of the searched element from a sorted array. This program comprises a bug in Line 2, where `boolean condComplex = (length > 0) && (result < 0);` would be the correct statement.

```
1. int result = -1;
2. boolean condComplex = (length == 0) && (result < 0);
   // correct version is: length > 0
3. while (condComplex) {
4.   if (length == 1) {
5.     if (values[start] == value)
6.       result = start;
       else
7.     length = 0;
   } else {
8.   int m = (start + length) / 2;
9.   if (value < values[m])
10.    length = length / 2;
       else {
11.    length = length - length / 2;
12.    start = m;
       } }
13. condComplex = (length == 0) && (result < 0);
   }
14. finalResult = result;
```

Figure 4.3.: A binary search algorithm comprising a bug in Line 2

In order to state the debugging problem, we assume a program  $\Pi \in \mathcal{L}$  that does not behave as expected. In the context of this current section, such a program  $\Pi$  is faulty when there exist input values from which the program computes output values differing from the expected values. The input and correct output values are provided to the program by means of a test case. For defining test cases we introduce variable environments (or environments for short). An environment is a set of pairs  $(x, v)$



where  $x$  is a variable and  $v$  its value. In an environment there is only one pair for a variable. We are now able to define test cases formally as follows:

**Definition 16 [Test Case]** A test case for a program  $\Pi \in \mathcal{L}$  is a tuple  $(I, O)$  where  $I$  is the input variable environment specifying the values of all input variables used in  $\Pi$ , and  $O$  the output variable environment, not necessarily specifying values for all output variables.

For example a (failing) test case for the program from Figure 4.3 is  $I_\Pi : \{a = 2; b = 1\}$  and  $O_\Pi : \{result = 2\}$ .

A test case is a *failing test case* if and only if the output environment computed from the program  $\Pi$ , when executed on input  $I$ , is not consistent with the expected environment  $O$ , i.e., when  $\text{exec}(\Pi, I) \not\subseteq O$ . Otherwise, we say that the test case is a *passing test case*. If a test case is a failing (passing) test case, we also say that the program fails (passes) the test case. For the program from Figure 4.3 the test case  $(I_\Pi, O_\Pi)$  is a failing test case. For input  $I_\Pi$  the program will return  $result = 3$  which contradicts the expected output  $O_\Pi : \{result = 2\}$ .

**Definition 17 [Test Suite]** A test suite  $TS$  for a program  $\Pi \in \mathcal{L}$  is a finite set of test cases of  $\Pi$ .

A program is said to be correct with respect to  $TS$  if and only if the program passes all test cases. Otherwise, we say that the program is incorrect or faulty.

We are now able to state the debugging problem.

**Definition 18 [Debugging Problem]** Let  $\Pi \in \mathcal{L}$  be a program and  $TS$  its test suite. If  $T \in TS$  is a failing test case of  $\Pi$ , then  $(\Pi, T)$  is a debugging problem.

A solution to the debugging problem is the identification and correction of a part of the program responsible for the detected misbehavior. We call such a program part an explanation. There are many approaches that are capable of returning explanations including [189, 203, 115, 3, 114, 27, 130] among others. In this joint research, we make use of a constraint based debugging approach [27, 130]. In particular, the approach makes use of the program's constraint representation to compute possible fault candidates. So, debugging is reduced to solving the corresponding constraint satisfaction problem (CSP).

**Definition 19 [Constraint Satisfaction Problem (CSP)]** A constraint satisfaction problem is a tuple  $(V, D, CO)$ , where  $V$  is a set of variables defined over a set of domains  $D$  connected to each other by

a set of arithmetic and boolean relations, called constraints  $CO$ . A solution for a CSP represents a valid instantiation of the variables  $V$  with values from  $D$  such that none of the constraints from  $CO$  is violated.

Note that the variables used in a CSP are not necessarily variables used in a program. We discuss the representation of programs as a CSP in the next section. Afterwards we introduce an algorithm for computing bug candidates from debugging problems. This algorithm uses only statements as potential explanations for a failing test cases. No information regarding how to correct the program is given. Hence, we have to extend the approach to deliver also repair suggestions. This is done by mutating program fragments. In the current context, we briefly recall the definitions already introduced in Section 2.1.

**Definition 20 [Mutant]** Given a program  $\Pi$  and a statement  $S_\Pi \in \Pi$ . Further let  $S'_\Pi$  be a statement that results from  $S_\Pi$  when applying changes like modifying the operator or a variable. We call the program  $\Pi'$ , which we obtain when replacing  $S_\Pi$  with  $S'_\Pi$ , a mutant of program  $\Pi$  with respect to statement  $S_\Pi$ .

Another important issue in the theory of program mutation is the identification of a test case able to outline the semantic difference between a program and its mutant. We call such a test case a *distinguishing test case*.

**Definition 21 [Distinguishing Test Case]** Given a program  $\Pi \in \mathcal{L}$  and one of its mutant  $\Pi'$ , a distinguishing test case for program  $\Pi$  and its mutant  $\Pi'$  is a tuple  $(I, \theta)$  such that for the input value  $I$  the output value of program  $\Pi$  differs from the output value of program  $\Pi'$ .

### 4.3.2. Related Approaches

This joint research work is mainly based on model-based diagnosis (MBD)[156] and its application to debugging, dating back to the early 1990s. Console et al. [38] described how to apply MBD for diagnosing logic programs. They also showed that their approach improves earlier approaches in the same domain [168]. Bond and Pagurek [16], and Bond [15] improved Console et al.'s paper. The work on debugging of logic programs more recently led to diagnosis of knowledge bases [59, 60].

Beat Liver [103] described the first application of MBD for ordinary sequential programs. In contrast to other MBD based debugging approaches, Liver relies on a specification of the modules comprising a program because these specifications are used as model. Therefore, Liver's approach is

rather limited. In order to avoid this limitation and to make MBD applicable in all cases where the source code is available, Friedrich and colleagues [63], and later Wotawa [197], introduced the techniques behind a debugger for VHDL programs. The debugger converts programs into logical models, which are used directly for locating bugs. In [63] the authors use a model where control and data dependencies are represented. Wotawa and colleagues [197], represent the structure of a program and the behavior of the statements as logical equations that are used by a diagnosis engine to compute diagnosis candidates. Peischl and Wotawa [203] overcome some limitations of [197], e.g., considering only combinatorial circuits represented as VHDL programs; they discuss and compare the obtained results.

Following the approach of representing programs as logical formulas Mayer et al. [114, 116, 117, 118] introduced extensions to handle object-oriented programs. In their approach basically a constraint network representation and a solver based only on value propagation are used for debugging. Because of the underlying model, the approach initially tended to produce too many diagnosis candidates. Improvements, like the introduction of abstractions [116, 114], help to reduce ambiguities and thus to minimize the overall number of computed diagnoses, making the approach useful even in case of object-oriented programs.

Also, we make use of model-based diagnosis, where we use a constraint representation of a program together with a general constraint solver for fault detection. The use of a general constraint solver instead of a value propagator helps to reduce ambiguity. Tests have shown that our constraint approach is as good as the approach used by Mayer [114], where abstract interpretation is used. The closest work to the research described in this section are [27, 130, 200, 204]. However, instead of focusing only on constraint-based debugging, we present an approach for debugging based on constraints, mutations and the generation of distinguishing test cases that allows for a methodical and algorithmic reduction of fault candidates (based on available knowledge, i.e., the source code, and a test suite). Moreover, our approach allows generating test suites based on the ability of distinguishing fault candidates.

Program mutations is mainly used for testing [24],[146, 142],[140]. However, recently, program mutation was also used for debugging. In [189] and more recently [43] the authors describe the application of mutations and genetic programming to program debugging. In order to avoid computing too many mutants, the authors use focusing techniques based on dependences and spectrum-based methods [87, 3], respectively. The use of mutations is similar to our work. The difference is that we are using constraints together with mutations and generation of distinguishing tests, for reducing the

size of the fault candidates set.

Gotlieb and colleagues [68] presented an approach that makes use of constraints for test case generation. The approach presented in this paper shares the idea of using constraints and a constraint solver for test case generation. We focus however, on computing tests that distinguish between fault candidates (instead of computing arbitrary tests). Collavizza and Rueher [35] published further work in the context of program verification. Contrary to their work, we are interested in locating the fault instead of only detecting it. It is also worth mentioning that there are many other works on testing within the MBD community, e.g., [169, 120, 56]. In [169] the authors also describe a method for computing distinguishing test cases, but there the focus is more on hardware and not on software like in our case.

Other more recent approaches of debugging include delta debugging [207], which can be applied to test case minimization as well as to fault localization, e.g., [70], spectrum-based debugging [87, 3, 4], and slicing based methods like [97, 13, 208, 199, 127]. Spectrum-based debugging makes use of a test suite to identify the most likely fault candidates. The idea is to check whether a statement is executed in a passing or failing run. In case a statement is only called in failing test cases, it is most likely to be faulty. When a statement is only executed in passing test cases, the statement cannot contribute to the faulty behavior and thus can be excluded from the list of fault candidates. For statements that are executed in failing and passing program runs a coefficient is computed based on the information about how often a statement is executed in passing or failing runs. There are many coefficient definitions available, e.g., Ochiai used by Abreu et al. [3, 4]. Empirical results indicate that spectrum-based debugging generally gives good ranking results that allow focusing the users' attention to a smaller portion of the source code to be examined in order to correct the bug. However, the outcome depends on the structure of the program, and in the worst case, i.e., programs where only data flow is present, spectrum-based debugging does not give any useful ranking (because all statements are equally likely to be faulty in this case).

Slicing-based approaches make use of the data and control flow for debugging. The idea is to trace back from wrongly computed variables along the data and control flow graph, ignoring those statements that do not have an influence on the computation of the value. Weiser [190, 191] pointed out that this approach is very close to the approach used by programmers for fault localization. The static approach for slicing [190, 191] does not make use of test cases when computing a slice. Hence, the statements that are element of such a computed slice are most likely a super set of the statements necessary to compute a faulty value for a specific test case. In order to get rid of the problem Korel

and Laski [92] invented the concept of dynamic slicing to be used for debugging [93]. There are also several improvements given in literature, e.g., [48, 208]. Interestingly there is a close relationship between slicing and MBD. Wotawa [198] proved this relationship for the static slicing case. In [127] dynamic slices together with some MBD techniques are used to compute possible root causes of a misbehavior together with their probabilities for being faulty.

### 4.3.3. Generating Distinguishing Test Cases

Before converting a program  $\Pi \in \mathcal{L}$  into its corresponding constraint representation we have to apply some intermediate transformation steps. These transformations are necessary for removing its imperative behavior, i.e., making it a declarative one, as required by the constraint programming paradigm.

Our three step algorithm for converting a program and encoding its debugging problem into a CSP is as follows:

1. *Loop elimination*  $\Pi_{LF} = LR(\Pi)$ : We define loop-elimination as a recursive function where  $n \geq 0$  is the number of iterations:

$$LF(\text{while } C \{B\}, n) = \begin{cases} \text{if } C \{B \text{ } LF(\text{while } C \{B\}, n-1)\} & \text{if } n > 0 \\ \text{if } C \{ \text{too\_many\_iterations} = \text{true}; \} & \text{otherwise} \end{cases}$$

We replace each loop-structure by a number of nested if-statements, i.e., number of iterations. The number of iterations  $n$ , is given by the test case. We furthermore make use of a variable `too_many_iterations` that handles the situation where a test case exceeds  $n$ . The variable `too_many_iterations` has to be declared and initialized with `false`. This can be easily done by assuming that the first line of a converted program always is `boolean too_many_iterations = false; .` It is worth looking at the consequences when not using the variable `too_many_iterations`. We illustrate this by means of a simple example. Consider the following program:

```

1. x = inp + 3;
2. while (x > 5 ) {
3.     x = x - 1;
4. }
```

This example implements a function that sets  $x$  to  $inp$  if  $inp \leq 5$  and to 5, otherwise. The unrolled version of the program assuming  $n = 0$ , i.e., a program run without iterations, and not using the `too_many_iterations` variable would be the following:

```
1. x = inp + 3;
```

When considering the only remaining statement as equation, all values fulfilling this equation are test cases. Therefore,  $inp=4$  and  $x=7$  would be a test case. Unfortunately, this is not a valid test case for the original program. Consequently, there must be an indicator in case a solution exceeds the pre-defined number of iterations. In our conversion this is handled with the special variable `too_many_iterations`. Note that we do not include `too_many_iterations` in the SSA conversion process, this being a necessary precondition for step 2 of our algorithm.

For the two iterations version of the program from Figure 4.3 have a look at Figure 4.4.

2. *SSA conversion*  $\Pi_{SSA} = SSA(\Pi_{LF})$ : The static single assignment (SSA) form is an intermediate representation of a program with the property that no two left-side variables share the same name. This property of the SSA form allows for an easy conversion into a CSP. It is beyond our scope to detail the program-to-SSA conversion. However, to be self-contained we only explain the necessary rules needed for converting our running example into its SSA representation. For more details regarding the SSA-conversion see, for example, [200].

- We convert *assignments* by adding an index to a variable each time the variable is defined, i.e., occurs at the left side of an assignment. If a variable is re-defined, we increase its unique index by one such that the SSA-form property holds. The index of a referenced variable, i.e., a variable occurring at the right side of an assignment, equals to the index of the last definition of the variable.
- We split the conversion of *conditional structures* into three steps: (1) the entry condition is saved in an auxiliary variable, (2) each assignment statement is converted following the above rule, and (3) for each conditional statement and variable defined in the sub-block of the statement, we introduce an evaluation function

$$\Phi(v_{\text{then}}, v_{\text{else}}, \text{cond}) \stackrel{\text{def}}{=} \begin{cases} v_{\text{then}} & \text{if } \text{cond} = \text{true} \\ v_{\text{else}} & \text{otherwise} \end{cases}$$

which returns the statement conditional-exit value, e.g.,  $v_{\text{after}} = \Phi(v_{\text{then}}, v_{\text{else}}, \text{cond})$ .

```

0. boolean too_many_iterations = false;
1. int result = -1;
2. boolean condComplex = (length == 0) && (result < 0);
//   correct is: length > 0
3. if (condComplex) {
4.   if (length == 1) {
5.     if (values[start] == value)
6.       result = start;
       else
7.     length = 0;
   } else {
8.     int m = (start + length) / 2;
9.     if (value < values[m])
10.    length = length / 2;
       else {
11.    length = length - length / 2;
12.    start = m; } }
13. condComplex = (length == 0) && (result < 0);
14. if (condComplex) {
15.   if (length == 1) {
16.     if (values[start] == value)
17.       result = start;
       else
18.     length = 0; }
   else {
19.     int m = (start + length) / 2;
20.     if (value < values[m])
21.       length = length / 2;
       else {
22.       length = length - length / 2;
23.       start = m;}}
24. condComplex = (length == 0) && (result < 0);
25. if (condComplex) {
26.   too_many_iterations = true;}}
27. finalResult = result;

```

Figure 4.4.: Two iteration unrolling for the program from Figure 4.3

- **Arrays** in our case need not to be treated specially because the used constraint solver is able to handle arrays directly.

The way we handle **arrays** is similar to the way we handle assignment statements. For the purpose of explaining the conversion of arrays, we assume an array  $A$  of length  $n > 0$  with elements  $\langle a_1 \dots a_i \dots a_n \rangle$ . The access to elements is assumed to be done using the  $[\ ]$  operator, which maps from  $A$  and a given index  $i$  to the array element  $a_i$ . We now discuss

two cases, i.e., the array is used at the right side of an assignment, and the array occurs at the left side of an assignment.

In a statement of type  $z = A[E]$ , i.e., the array access is found at the right hand of an equation with index  $E$ ,  $A$  is represented in the SSA form as  $A_k$ , where  $k$  is the currently given unique index  $k$  for the array  $A$ .

The more difficult part is handling statements like  $A[m] = x_{\text{expr}}$ , where the array is on the left hand side of an assignment. For this purpose we use an **update array** function [42]:  $\Psi(A, i, \text{exp})$ . The function  $\Psi$  returns a new array  $A'$ , which except for the value at index  $i$ , has the same values as the array  $A$ . For example, if we encounter statement  $A[m] = x_{\text{expr}}$  during conversion, then its SSA form is  $A_{k+1} = \Psi(A_k, m, x_{\text{expr}})$ . We now formally define the function  $\Psi$ . Assume a program fragment  $A[i] = f(\vec{x})$ , where the  $i$ -th element of  $A$  is set to the outcome of function  $f$  given parameters  $\vec{x}$ . This statement only changes the  $i$ -th element but not the others.

```
{ A } // A before the statement
A[i] = f(x)
{ A' } // A after the statement
```

The new value after executing the statement is given as follows:  $A'[i] = f(\vec{x})$  and  $\forall j \in \{1, \dots, n\}, i \neq j: A'[j] = A[j]$ . As a consequence, we assume that the function  $\Psi$  (written as `Psi` in the source code) implements this semantics in order to allow for replacing the original statement with  $A = \Psi(A, i, f(\vec{x}))$ . Note that this semantics might not be correct in general. In particular, in the case of side-effects the converted program behaves differently than the original one.

The SSA representation of the program from Figure 4.4 is given in Figure 4.5.

3. *Constraint conversion*  $CON = CC(\Pi_{SSA})$ : This last step of the conversion process transforms the SSA statements to the corresponding constraints, including also the encoding of the debugging problem. For this purpose we introduce a special Boolean variable  $AB(S)$  for a statement  $S$ , that states the incorrectness of a statement  $S$ . The constraints model of a statement comprises corresponding constraints or-connected with  $AB(S)$ . Let  $S \in \Pi_{SSA}$  and let  $C_S$  be the constraint encoding statement  $S$  in the constraint programming language. Note that  $\phi$  functions cannot be incorrect. Hence, no  $AB$  variable is defined for statements using  $\phi$ . We model  $S$  in  $CON$  as



```

0. boolean too_many_iterations = false;
1. int result_0 = -1;
2. boolean condComplex_1=(length_0==0)&&(result_0<0);
3. boolean cond_0=condComplex_1==true;
4. boolean cond_1=cond_0&&length_0==1;
5. boolean cond_2=cond_1&&values_0[start_0]==value_0;
6. int result_1=start_0;
7. int length_1=0;
8. int result_2= $\phi$ (result_0,result_1,cond_2);
9. int length_2= $\phi$ (length_1,length_0,cond_2);
10. int m_1=(start_0+length_0)/2;
11. boolean cond_3=!cond_1&&value_0<values_0[m_1];
12. int length_3=length_2/2;
13. int length_4=length_2-length_2/2;
14. int start_1=m_1;
15. int length_5= $\phi$ (length_4,length_3,cond_3);
16. int start_2= $\phi$ (start_1,start_0,cond_3);
17. int result_3= $\phi$ (result_0,result_2,cond_1);
18. int length_6= $\phi$ (length_5,length_3,cond_1);
19. int m_2= $\phi$ (m_1,m_0,cond_1);
20. int start_3= $\phi$ (start_2,start_0,cond_1);
21. boolean cond_4=cond_0&&condComplex_1==true;
22. boolean cond_5=cond_4&&length_6==1;
23. boolean cond_6=cond_5&&values_0[start_3]==value7_0;
24. int result_4=start_3;
25. int length_7=0;
26. int result_5= $\phi$ (result_3,result_4,cond_6);
27. int length_8= $\phi$ (length_7,length_6,cond_6);
28. int m_3=(start_3+length_6)/2;
29. boolean cond_7=!cond_5&&value_0<values_0[m_3];
.....
52. int m_7= $\phi$ (m_3,m_6,cond_0);
53. int length_20= $\phi$ (length_15,length_19,cond_0);
54. int length_21= $\phi$ (length_16,length_20,cond_0);
55. int start_9= $\phi$ (start_4,start_8,cond_0);
56. boolean condComplex_6=get_branch(condComplex_3,condComplex_5,cond_0);
57. int finalResult_1=result_9;

```

Figure 4.5.: Partial SSA representation corresponding to the program from Figure 4.4

follows:

$$CON \cup \begin{cases} AB(S) \vee C_S & \text{if } S \text{ does not contain } \phi \\ C_S & \text{otherwise} \end{cases}$$

Hence the CSP representation of a program  $\Pi$  is given by the tuple

$(V_{\pi_{SSA}}, D_{SSA}, CON)$ , where  $V_{\pi_{SSA}}$  represents all variables of the SSA representation  $\Pi_{SSA}$  of pro-

gram  $\Pi$ , defined over the domains  $D_{SSA} = \{Integer, boolean\}$ .

Debugging of a program requires the existence of a failing test case. This means that in addition to the set of constraints  $CON$ , we must add an extra set of constraint encoding a failing test case  $(I, O)$ . For all  $(x, v) \in I$  the constraint  $x_0 = v$  is added to the constraint system. For all  $(y, w) \in O$  the constraint  $y_t = w$  is added where  $t$  is the greatest index of variable  $y$  in the SSA form. Let  $CON_{TC}$  denote the constraints resulting from converting the given test case. Then, the CSP corresponding to the debugging problem of a program  $\Pi$  is now represented by the tuple  $(V_{\pi_{SSA}}, D_{SSA}, CON \cup CON_{TC})$ . If there is more than one failing test cases, then each failing test case will form together with the program a single debugging problem.

In our implementation we model the CSP to represent the debugging problem in the language of the MINION constraint solver [64]. MINION is an out of the box, open source constraint solver. Its syntax requires a little effort in modeling the constraints than other constraint solvers, e.g., it does not support different operators on the same constraint. Because of this drawback sometimes complex constraints have to be split into two or three more simpler constraints. However, because of this characteristic, MINION, unlike other constraint solver toolkits, does not have to perform an intermediate transformation of the input constraint system. For example the MINION representation of statement 1 from Figure 4.5 is shown in Figure 4.6 given by statements 1,2,3 and 4, whereas the MINION constraints corresponding to line 4 of the program from Figure 4.5 are represented by: 9, 10, 11, 12. Statements 15 to 18 from the MINION representation correspond to the  $\Phi$  functions given in statement 7 and 8 from Figure 4.5, and the last SSA statements from Figure 4.5, to, correspond to the Minion statements 111 to 121. The failing test case is given by the lines 125 to 129. As it can be easily noticed the correct output should be 1, which is not the case, because for the current failing statement, the program will always output -1, independent from the input value.

The approach presented above comprises 3 steps. In the first step we compute a set of bug candidates, i.e., program statements that might cause the revealed misbehavior, from the constraint representation of the program  $\Pi \in \mathcal{L}$ . In the second step, we compute the set of mutants for each candidate that lead to a new program passing all previously failing test cases. If no such mutant can be found we remove this candidate from the list of bug candidates. In the third step, we compute distinguishing test cases that distinguish between two randomly selected bug candidates. We execute the third step several times until the number of bug candidates is smaller than a given value or no further reduction is possible. In this section, we explain each of the debugging steps starting from the computation of

### 4.3. Mutation Based Test Case Generation

---

```

1. watched-or({element(ab,1,1), reify(ineq(0,length_0,-1),cond_aux1)})
2. watched-or({element(ab,1,1), reify(ineq(result_0,0,-1),cond_aux2)})
3. watched-or({element(ab,1,1), reify(watchsumgeq(
    [cond_aux1,cond_aux2], 2),condComplex_1)})
.....
9. element(values_0,start_0,auxARRAY1)
10. reify(eq(auxARRAY1,value_0),cond_aux6)
11. watched-or({element(ab,4,1), reify(watchsumgeq(
    [cond_aux6,cond_aux6], 2),cond_aux7)})
12. watched-or({element(ab,4,1), reify(watchsumgeq(
    [cond_1,cond_aux7], 2),cond_2)})
.....
15. watched-or({eq(cond_2,0), eq(result_2,result_1)})
16. watched-or({eq(cond_2,1), eq(result_2,result_0)})
17. watched-or({eq(cond_2,0), eq(length_2,length_0)})
18. watched-or({eq(cond_2,1), eq(length_2,length_1)})
.....
111. watched-or({eq(cond_0,0), eq(m_7,m_6)})
112. watched-or({eq(cond_0,1), eq(m_7,m_3)})
113. watched-or({eq(cond_0,0), eq(length_20,length_19)})
114. watched-or({eq(cond_0,1), eq(length_20,length_15)})
115. watched-or({eq(cond_0,0), eq(length_21,length_20)})
116. watched-or({eq(cond_0,1), eq(length_21,length_16)})
117. watched-or({eq(cond_0,0), eq(start_9,start_8)})
118. watched-or({eq(cond_0,1), eq(start_9,start_4)})
119. watched-or({eq(cond_0,0), eq(condComplex_6,condComplex_5)})
120. watched-or({eq(cond_0,1), eq(condComplex_6,condComplex_3)})
121. watched-or({element(ab,12,1), eq(finalResult_1,result_9)})
.....
125. eq(value_0, 3)
126. lexleq(values_0, [2,3])
127. lexleq([2,3], values_0)
128. eq(start_0, 0)
129. eq(finalResult_1, 1)

```

Figure 4.6.: The Constraint Representation of the Program From Figure 4.5

candidates using the CSP representation to the computation and use of distinguishing test cases.

Let  $CON_{\Pi}$  be the constraint representation of a program  $\Pi$  and  $CON_T$  the constraint representation of a failing test case  $T$ . The debugging problem formulated as a CSP comprises  $CON_{\Pi}$  together with  $CON_T$ . Note that in  $CON_{\Pi}$  assumptions about correctness or incorrectness of statements are given, which are represented by a variable  $AB$  assigned to each statement. The algorithm for computing bug candidates calls the CSP solver using the constraints and asks for a return value of  $AB$  as a solution.

The size of the solution corresponds to the size of the bug, i.e., the number of statements that must be changed together in order to explain the misbehavior. We assume that single statement bugs are more likely than bugs comprising more statements. Hence, we ask the constraint solver for smaller solutions first. If no solution of a particular size is found, the algorithm increases the size of the solutions to be searched for and iterates calling the constraint solver. This is done until either a solution is found or the maximum size of a bug, which is equivalent to the number of statements in  $\Pi$ , is reached.

**Algorithm CSP\_Debugging** ( $CON_{\Pi}, CON_T$ )

*Inputs:* A constraint representation  $CON_{\Pi}$  of a program  $\Pi$ , and a constraint representation  $CON_T$  of a failing test case  $T$ .

*Outputs:* A set of minimal bug candidates.

1. Let  $i$  be 1.
2. While  $i$  smaller or equal to the number of statements in  $\Pi$  do:
  - a) Call the constraint solver using  $CON_{\Pi}, CON_T$  to search for solutions regarding the  $AB$  variables, where only  $i$  statements are allowed simultaneously to be incorrect.
  - b) If the constraint solver returns a non-empty set of solutions, then return this set as result and leave the algorithm.
  - c) Otherwise, let  $i$  be  $i + 1$ .
3. Return the empty set as result.

For example, for the constraint system from Figure 4.6 the constraint solver MINION finds 5 possible explanations for the failing test case  $I : (a_0 = 0, b_0 = -250), O : (result_7 = 0)$  in less than 0.1s; the working environment was Eclipse, using an Intel Pentium Dual Core 2 GHz computer with 4 GB RAM. This result is very satisfactory, especially with respect to computation time. However, further steps might be performed in order to reduce the size of the bug candidates. For this purpose we suggest the use of mutations.

Assume a faulty program  $\Pi$  and a failing test case  $(I, O)$ . Let  $D_{AB}$  be the set of bug candidates obtained when calling **CSP\_Debugging** on the constraint representation of  $\Pi$  and  $(I, O)$ . The following algorithm makes use of program mutations for further restricting  $D_{AB}$ .

**Algorithm Filter\_TestCase** ( $D_{AB}, \Pi, T$ )

*Inputs:* A set of bug candidates  $D_{AB}$ , the faulty program  $\Pi$ , and the failing test case  $T$ .

*Outputs:* A set of mutants  $Mut_{\Pi}$  of program  $\Pi$ .

1. Let  $Mut_{\Pi}$  be the empty set.
2. For all elements  $d \in D_{AB}$  do:
  - a) Generate all mutants of program  $\Pi$  with respect to the statements stored in  $d$  and store them in  $V_{Mut}$ .
  - b) Add to  $Mut_{\Pi}$  every program  $\Pi' \in V_{Mut}$ , which passes the test case  $T$ .
3. Return  $Mut_{\Pi}$ .

The **Filter\_TestCase** algorithm returns for the faulty program  $\Pi$  a set of repair possibilities  $Mut_{\Pi}$ . Due to the usage of the debugging algorithm **CSP\_Debugging**, we compute the repair only for the resulting bug candidates set  $D_{AB}$ . A mutant is part of  $Mut_{\Pi}$ , i.e., a repair, if and only if it is able to pass the failing test case  $T$ . Hence, we expect that the number of bug candidates can be reduced. Moreover, since mutation is only applied for bug candidates we do not need to compute all possible mutations even in the case when they cannot explain the revealed misbehavior.

The number of repair possibilities for a statement of the  $D_{AB}$  set is strongly tied to the capabilities of the used mutation operators and the used mutation tool. Because of this fact this part of the approach is as good as the available capability of the used mutation tool. Note that after applying the **Filter\_TestCase** algorithm, in our experiments we were able to eliminate between 20% and 90% of the bug candidates, because of the inability of the suggested repair to pass the test case. Hence, filtering based on mutations was very successful.

The last step of our algorithm comprises the integration of distinguishing test cases to further reduce the bug candidate set. Let  $Mut_{\Pi}$  be the set of mutants for a program  $\Pi$  obtained after applying the **Filter\_TestCase** algorithm. And let  $CON_{Mut_{\Pi}}$  be the constraint representation of the programs from  $Mut_{\Pi}$ .

**Algorithm TestCase\_Generator**  $Mut_{\Pi}, CON_{Mut_{\Pi}}$

*Inputs:* A set of valid repair possibilities,  $Mut_{\Pi}$ , for a faulty program  $\Pi$  and their constraint representation  $CON_{Mut_{\Pi}}$ .

*Outputs:* A subset of  $Mut_{\Pi}$ .

1. Let *Tested* be empty.

2. If there exist mutants  $\Pi', \Pi'' \in Mut_{\Pi}$  with  $(\Pi', \Pi'') \notin Tested$ , add  $(\Pi', \Pi'')$  to *Tested* and proceed with the algorithm. Otherwise, return  $Mut_{\Pi}$
3. Let  $CON_{\Pi'}$  and  $CON_{\Pi''} \in CON_{Mut_{\Pi}}$  be the constraint representation of programs  $\Pi'$  and  $\Pi''$  respectively.
4. Let  $CON_{TC}$  be the constraints encoding  $Input_{\Pi'} = Input_{\Pi''} = I \wedge Output_{\Pi'} \neq Output_{\Pi''}$
5. Solve the CSP:  $CON_{\Pi'} \cup CON_{\Pi''} \cup CON_{TC}$  using a constraint solver.
6. Let  $O$  be the correct output for the original program  $\Pi$  on input  $I$  (derived from user interaction or specifications).
7. If  $Output_{\Pi'} \neq O$ , then delete  $\Pi'$  from  $Mut_{\Pi}$ .
8. If  $Output_{\Pi''} \neq O$ , then delete  $\Pi''$  from  $Mut_{\Pi}$ .
9. If (CSP has no solution) go to step 1.
10. For all  $\Pi' \in Mut_{\Pi}$  do:
  - a) If  $\Pi'$  fails on generated test case  $(I, O)$  delete  $\Pi'$  from  $Mut_{\Pi}$ .
11. Return  $Mut_{\Pi}$ .

The above algorithm searches for two mutants, distinguished via a test case. The algorithm in the current form is restricted to search for only one pair of such mutants but can be easily changed in order to compute several different pairs where a distinguishing test case is available. The only disadvantage of this algorithm is that Step 6 requires an interaction with an oracle. If no automated oracle is available user interactions are required and prevent the approach from being completely automated. To solve the constraint system resulted at step 5 we use the MINION constraint solver. Another characteristic of this approach is that, for the CSP to be solvable, the name of the variables of the two mutants should differ. This is however an encoding problem which can be easily overcome by encapsulating in the name of each variable the name of the mutant file. When using the above approach for the example from 4.4 we are able to reduce the conflict set to one element, which was also the correct one. For more information regarding distinguishing test cases and their computation using MINION, we refer the interested reader to [201].

To obtain the program's set of mutants relative to the set of fault candidates we relay on the JAVA mutation tool MuJava [105]. MuJava is a Java based mutation tool, which was originally developed by Offut, Ma, and Kwon. Its main three characteristics are:

1. Generation of mutants for a given program.
2. Analysis of the generated mutants.
3. Running of provided test cases.

Due to the new implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration of the tool in the testing or debugging process.

Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. As already presented in the beginning of this Chapter, the tool implements both types of mutation operators:

- Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
- Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

In mutation testing one significant problem is represented by the equivalent mutant problem. A mutant is said to be equivalent with the original program when there is no test case that can detect that modification - the output will always be the same with the output of the original program. Figure 4.7 illustrates this particular situation, in the case of an arithmetic operator replacement (AOR) mutation. These mutants should be detected because, otherwise, there will never be killed. This is a tedious task, which can not be performed manually. In order to derive an automated detection of the equivalent mutants, there were proposed several techniques [142, 71, 167] .

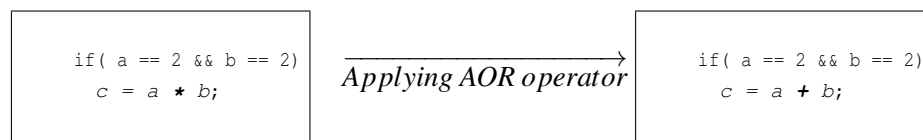


Figure 4.7.: Equivalent Mutant

For our experiments, up to now we have considered only traditional mutation operators. Moreover, we further restrict the mutation operators to mutations on expressions comprising deletion, replacement, and insertion of primitive operators (arithmetic operators, relational operators, conditional operators, etc.). Mutation by deletion of operands or statements was proved to be inefficient [140].

Because of the selected tools there are currently some limitations of our implementation. For example, if the bug is on the left side of an assignment we cannot correct it. Another limitation is due to constants. If the bug occurs in the initialization, MuJava is not able to generate any mutants. Missing statements are another limitation imposed by the currently used tools. Therefore, we currently do not consider bugs because of missing statements. Finally, there is a limitation regarding multiple bugs in one statement. The MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program.

### Experimental results

We tested the approach using a set of programs implementing various numerical operators as well as searching. Moreover, we also applied our approach to the `tcas` program that is often used in the debugging research community, e.g., [114] for comparing different approaches. In each program we manually injected one single fault. All the injected faults can be found at the right side of an assignment statement. With the exception of the `tsa03` program all faults are functional faults, i.e., there are no changes in the program structure, e.g., caused by changing a variable. Furthermore, we used the original bug free version of each faulty program as the test oracle in our experiments. Using the output values of the original bug-free program, we were able to decide which of the mutants should be eliminated after computing the distinguishing test cases. Obviously, in the real life situation someone cannot benefit from the existence of such a reference program or other executable specification. Consequently, when applying our method in a real world setting, the user has to determine the correct output for the given inputs.

The process of mutant generation, program to CSP conversion, and the computation of the conflict set is fully automated. The generation of the distinguishing test cases was performed manually with the help of the MINION constraint solver. Basically, we have to manually construct the used constraint system. However, even this step can and will be fully automated in future releases of our prototype. In order to obtain the experimental results, we applied the following process. For each program we first performed the conversion into its constraint representation. Then we computed the fault candidates. For each fault candidate, i.e., faulty statement, we computed all its possible mutants. We eliminated from the generated set of mutants all mutants which were not able to pass the error revealing test case. In addition, we tested the number of oracle-interactions required to obtain the minimal set of faulty components. By an "oracle-interaction" we mean repeating the **TestCase\_Generator** algorithm until no other distinguishing test case can be generated, i.e., each time we applied the algorithm we



asked the oracle, i.e., the original fault free program in our case, to provide the correct output for the generated test case.

The results of the experimental study are given in Table 4.7 and 4.8, which contain the following information:

- a number of iterations **It**,
- the number of variables **Var<sub>π</sub>**,
- its size given in lines of code **LOC<sub>π</sub>**,
- the number of inputs **Inputs**,
- number of outputs **Outputs**,
- the size of its SSA representation given as lines of code **LOC<sub>SSA</sub>**,
- the number of MINION constraints **|CO|**,
- the number of MINION variables over which the constraint system is defined **Var<sub>CO</sub>**,
- the number of fault candidates **|Diag|**,
- the size of the conflict set resulted after applying *Filter\_TestCase* algorithm, **|Diag<sub>fit</sub>|**,
- the number of calls to the *TestCaseGenerator* algorithm, **#UI**, necessary to obtain the number of fault candidates,
- the number of fault candidates **|Diag<sub>TC</sub>|**.

When analyzing Table 4.7, we observe that in most cases we were able to eliminate more than half of the initial fault candidates. Hence, reducing the diagnosis candidates by eliminating those candidates where no mutant that passes the original test suite can be found, is very effective. The use of distinguishing test cases further reduces the number of fault candidates. Thus in many cases only one diagnosis candidate remains, which was always the correct one.

However, when using larger programs like *tcas* files from Table 4.8, a reduction to one diagnosis candidate was not possible. But even in this case, the approach leads to a reduction of more than 80 percent regarding the computed diagnosis candidates. It is worth noting that a reduction to one candidate is not always possible, as programs can be corrected at different locations. Also, regarding the *tcas* files, it can be seen that in some situations no elimination was done, due to the *limitations* imposed by the mutation tool. For constant assignments, e.g., `ConstVal = 20`, we do not generate

Name	It	Var $\pi$	LOC $\pi$	Inputs	Outputs	LOC <sub>ssa</sub>	CO	Varco	Diag	Diagn	#UI	DiagnC
DivATC_V1	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V2	2	5	21	2	1	32	33	29	5	3	1	1
DivATC_V3	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V4	2	5	21	2	1	32	33	29	4	4	1/2	3(1)/1
GedATC_V1	2	6	35	2	1	49	61	46	2	2	1	1
GedATC_V2	2	6	35	2	1	49	61	46	10	3	1/2/3/4/5	3/3/2/2/1
GedATC_V3	2	6	35	2	1	49	61	46	2	2	1	1
GedATC_V4	2	6	35	2	1	49	61	46	2	2	1	1
MultATC_V1	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V2	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V3	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V4	2	5	16	2	1	26	24	19	5	2	1	1
MultV2ATC_V1	2	6	20	2	1	49	67	46	6	2	1	1
MultV2ATC_V2	2	6	20	2	1	49	67	46	2	1	1	1
MultV2ATC_V3	2	6	20	2	1	49	67	46	6	1	1	1
SumATC_V1	2	5	18	2	1	27	24	20	2	2	1	1
SumATC_V2	2	5	18	2	1	27	24	20	3	2	1	1
SumATC_V3	2	5	18	2	1	27	24	20	5	2	1	1
SumPowers_V1	2	11	36	3	1	72	87	70	16	6	1/2/3/4	4/4/2/2
SumPowers_V2	2	11	36	3	1	72	87	70	11	6	1/2	2/1
SumPowers_V3	2	11	36	3	1	72	87	70	11	1	1	1
Binomial	2	37	189	6	3	481	1329	763	5	3	1	1
BinSearch	2	8	37	4	1	66	102	83	6	3	1	2
Hamming	2	15	77	2	1	176	266	185	9	5	1	3
Data	2	5	40	1	1	45	57	50	7	3	1	1
whiteTest	2	16	94	4	1	110	147	103	8	4	1	3

Table 4.7.: Experimental Results

### 4.3. Mutation Based Test Case Generation

Name	It	Var <sub><math>\pi</math></sub>	LOC <sub><math>\pi</math></sub>	Inputs	Outputs	LOC <sub>SSA</sub>	CO	Var <sub>CO</sub>	Diag	Diag <sub>int</sub>	#UI	Diag <sub>rc</sub>
tcas01	1	48	125	12	1	125	98	132	28	15	1	14
tcas02	1	48	125	12	1	125	98	132	26	-	-	-
tcas03	1	48	125	12	1	125	98	132	27	13	1/2/3/4	13/12/9/9
tcas04	1	48	125	12	1	125	98	132	25	14	1	14
tcas05	1	48	125	12	1	125	98	132	25	14	1	13
tcas06	1	48	125	12	1	125	98	132	25	12	1/2/3/4	12/12/12/11
tcas07	1	48	125	12	1	125	98	132	9	5	1	5
tcas08	1	48	125	12	1	125	98	132	27	13	1/2/3/4	11/11/11/10
tcas09	1	48	125	12	1	125	98	132	11	6	1/2/3/4	6/5/5/4
tcas10	1	48	125	12	1	125	98	132	29	18	1	17
tcas11	1	48	125	12	1	125	98	132	23	13	1	13
tcas12	1	48	125	12	1	125	98	132	23	12	1	12
tcas13	1	48	125	12	1	125	98	132	27	5	0	5
tcas14	1	48	125	12	1	125	98	132	6	2	0	2
tcas15	1	48	125	12	1	125	98	132	24	12	1	11
tcas16	1	48	125	12	1	125	98	132	26	6	0	6
tcas17	1	48	125	12	1	125	98	132	9	2	0	2
tcas18	1	48	125	12	1	125	98	132	9	2	0	2
tcas19	1	48	125	12	1	125	98	132	9	2	0	2
tcas20	1	48	125	12	1	125	98	132	27	13	1/2/3/4	13/12/12/11
tcas21	1	48	125	12	1	125	98	132	27	16	1/2/3/4	16/16/16/15
tcas22	1	48	125	12	1	125	98	132	8	4	1	4
tcas23	1	48	125	12	1	125	98	132	9	5	1	5
tcas24	1	48	125	12	1	125	98	132	24	-	-	-
tcas25	1	48	125	12	1	125	98	132	9	5	1	4
tcas26	1	48	125	12	1	125	98	132	25	14	1	14
tcas27	1	48	125	12	1	125	98	132	25	12	1	12
tcas28	1	48	125	12	1	125	98	132	14	7	1	7
tcas29	1	48	125	12	1	125	98	132	10	5	1	5
tcas30	1	48	125	12	1	125	98	132	13	9	1	9
tcas31	1	48	125	12	1	125	98	132	24	-	-	-
tcas32	1	48	125	12	1	125	98	132	23	13	1	13
tcas33	1	48	125	12	1	125	98	132	9	3	0	3
tcas34	1	48	125	12	1	125	98	132	22	12	1	12
tcas35	1	48	125	12	1	125	98	132	14	8	1	8
tcas36	1	48	125	12	1	125	98	132	2	2	1	2
tcas37	1	48	125	12	1	125	98	132	9	-	-	-
tcas38	1	48	125	12	1	125	98	132	1	-	-	-
tcas39	1	48	125	12	1	125	98	132	9	-	-	-
tcas40	1	48	125	12	1	125	98	132	8	-	-	-
tcas41	1	48	125	12	1	125	98	132	27	17	1	17

Table 4.8.: TCAS Experimental Results

any mutants. Therefore, these statements will always be part of the fault candidates set. Another limitation of the mutation tool is that it does not remove/add statements or variables. Hence, there were cases when the approach failed, because the error was caused by statement insertion, e.g., *tcas02*, or by missing variables, e.g., *tcas37*. In his PhD thesis, Mayer [114] also used the 41 *tcas* files, but compared to his results, e.g., 88.3 percent, we were able to obtain a score of 92.7 percent, after applying the filtering algorithm.

Another factor, which influences the quality of the obtained results, is the way of choosing the mutant pairs for computing distinguishing test cases. Unfortunately, there is no theory available for predicting a certain pair of mutants that when selected produces the best or worst distinguishing test case. Because of that we randomly selected the pair of mutants when carrying out the experimental evaluation. Note that we also observed that after trying out all mutant pairs for the *DivATC\_V4* program the best distinguishing test case would lead to 1 element in the conflict set contrary to 3 given in Table 4.7.

It is also worth noting that computing the diagnosis candidates and the distinguishing test cases using the CSP solver MINION was very fast. In our experiments the necessary time for computing a distinguishing test case never exceeded 0.3 seconds using a Pentium 4 Dual core 2 GHz with 4 GB of RAM computer. Hence, for smaller programs or program parts that can be separately analyzed like methods the proposed approach is definitely feasible.

The computational complexity of the approach has to take into consideration the complexity for generating the mutants, converting a program to its constraint representation and the debugging complexity. From mutants generation perspective, as larger the programs are, as higher the complexity is. This is why, currently, we restrict the approach to small to medium size programs. Converting a program to its constraint representation requires a polynomial time; the programs considered in the experiments are all finite, hence the conversion will always terminate. Regarding the last step, we can state that the debugging complexity is given by the complexity for solving the constraint system. In case of finite domains, after checking all the solutions, the constraint solver will terminate. The computational complexity for solving the constraint system is exponential with the number of used variables.

#### 4.3.4. Future Steps

In this joint research work we presented an approach for restricting the number of potential diagnosis candidates by means of distinguishing test case generation and program mutation. A distinguishing test case for two diagnosis candidates is characterized by a set of inputs that reveal different executions for both diagnosis candidates such that they can be distinguished with respect to their output behavior. Just using the distinguishing test case alone we are not able to decide which diagnosis candidates to remove or if we should eliminate both from the list of candidates. This can only be done after consulting a test oracle, e.g., the user or a formal specification, for the expected output of the distinguishing test case. Candidates where the computed output is not equivalent to the expected one can be eliminated. The advantage of this approach is that only the input-output behavior of a program is used for distinguishing diagnosis candidates. Moreover, the approach computes additional test cases based on their discriminating power for distinguishing diagnosis candidates. Usually, test cases are generated for fulfilling coverage criteria like statement coverage or branch coverage.

Besides the theoretical contribution we extend the first experimental results [127] of the proposed approach. The results indicate that the approach allows a substantial reduction in the diagnosis candidates. For smaller programs we were able to reduce the diagnosis candidates to the real bug. Obviously, this was not always the case. For larger programs more diagnosis candidates remain. This has been somehow expected because programs cannot be usually corrected only by replacing one statement with another. Instead the right repair actions might comprise changes at different positions in the program. In future work we want to extend the experimental study. This includes the use of larger programs as well as example programs comprising multiple faults.

## 4.4. Equivalent Mutant Detection

As we presented along the thesis, one major problem of mutation testing is the *equivalent mutant problem*. In Section 4.2, we focused on running the extended MuJava tool over a well known and widely used software project, i.e., the Eclipse framework [75], and to establish whether or not mutation testing is scalable to real-world software applications. But mutation testing proved to be quite complex. A significant number of mutants were generated, which requires a lot of time for execution. Moreover, during the experiments we observed that we have to deal with equivalent mutants in many cases. As a consequence, we concluded that solving the equivalent mutants problem is an

important issue in mutation testing. First of all, the runtime complexity can significantly decrease, and also, through equivalent mutants detection and deletion, we achieve a more accurate mutation score. Therefore, we focused our attention towards this challenge.

We briefly recall that a mutant is said to be equivalent if there is not such a test case, able to differentiate between the output of the mutant and the output of the original program. When considering the definition of the mutation score, we see that detecting all equivalent mutants is very important. In literature several techniques for equivalent mutant detection exist, e.g., [142, 167]. Since the early stage of the research aimed to apply mutation testing in an industry setting, we have made as research goal the detection and removal of equivalent mutants, for our concrete environment. We strive on giving an answer to this problem and start the basis of a tool, **EqMutDetect**, serving our purpose. We base our approach and our tool implementation on *MuJava* and on constraint solving, for proving the mutants equivalence.

The *equivalent mutant problem* is a decision problem that allows to determine whether a program is behavioral equivalent to its mutant. This problem is obviously equivalent to the program equivalence problem that is well known to be undecidable in general.

#### 4.4.1. Related Work

An approach for mutant detection is described in [71]. The authors use program slicing for solving the equivalent mutant problem. In order to show whether a mutant is equivalent or not, the described technique compares the effect of a program and its mutant on certain variables. [147] introduces a solution most closely to ours, which is also based on constraints. The authors developed a tool that implements a mathematical constraint algorithm. For determining the equivalent mutants, the authors use constraints to recognize infeasible constraints that produce equivalent mutants. The used constraint will describe the circumstances under which a mutant must be detected, i.e., if a test case can detect the mutant, then the constraint system will be true. Otherwise, no test case is able to kill the mutants, and, therefore, an equivalent mutant is detected. The authors state that a mutation is detected only when it satisfies three conditions: reachability, necessity and sufficiency. Unlike in our approach, where we make use of the distinguishing test case concept, the authors propose three strategies to detect equivalent mutants: negation, constraint splitting and constants comparison. For negation, the authors use constraint negation and partial negation of two constraints, e.g., C1 and C2, rewrite one of the two constraints and then compare them. If they are syntactically equal, then the constraint

system is infeasible and a mutant with this infeasible constraint system is equivalent. But constraint negation does not help in determining if the necessity constraint conflicts with the path expression. Therefore, in order to detect conflicts, constraint splitting is introduced. Constant comparison is based on a property common in constraints generated for test cases: both constraints should have the format  $(V \text{ RelOp } K)$ , where  $V$  is a variable,  $\text{RelOp}$  is a relational operator, and  $K$  is a constant. Also the variables in both constraints are required to be the same.

[69] uses the impact on executions and also on the return values. The approach examines the impact of mutations at the coverage level. In particular the approach makes use of observations about the lines of code executed. The coverage from the original execution is compared with the coverage of the mutants. Regarding the impact over the return values, the authors established two types of non-equivalent mutations, which do not affect the coverage level: mutations which affect only the data, but not the output values of a program, and mutations reflected in the return values.

#### 4.4.2. Constraint Satisfaction Problem

For program conversion into the constraint representation, we make use of previous work, conducted in the constraint based testing research area [68]. In order to be self-contained we briefly recall the conversion of sequential programs into their equivalent constraint representation under certain assumptions. For the conversion of programs to their constraint representation we make use of previous work, conducted in the constraint based testing research area [68, 36, 49, 67]. To get a more detailed view in constraints and how the conversion into a constraint representation can be used for fault localization, the interested reader can consult [201]. We take again the small program from Figure 4.8 (a) as an example to demonstrate the conversion.

The first step of the conversion process is to eliminate all loop statements. The idea here is to replace a while statement with a nested conditional statement of a pre-defined nesting depth  $nd$ . Obviously the value of  $nd$  determines whether the original program behaves equivalent to its corresponding loop-free variant. If we choose a value for  $nd$  that is too small for a given test case, then the programs behave in a different way. On the other hand when choosing a very large value for  $nd$  the resulting loop-free program becomes unnecessarily large. Therefore in practice a trade-off for  $nd$  is necessary, together with means for detecting situations where  $nd$  is too small. We ensure this by introducing a fresh boolean variable `loopi` for each loop, that is initialized with `false` and set to `true` whenever nesting depth  $nd$  is not large enough.

In the second step the loop-free program is converted to its static single assignment form (SSA) [20]. In the SSA every variable is defined only once. This can be ensured by mapping each variable  $x$  occurring in a program to a variable  $x_i$  where  $i$  represents an index variable starting from 0. Every time  $x$  is defined, the index  $i$  is increased by 1 and added to the variable. If the variable is referenced after the current index,  $i$  is used until a new re-definition of  $x$  occurs. Beside adding indices there is one additional conversion rule for conditional statements. Let us assume the following program fragment: `if (x > 4) { y = 0; } else { y = 1; }`. After adding indices to the variable let us assume the following situation: `if (x_0 > 4) { y_1 = 0; } else { y_2 = 1; }`. In this case whether to use  $y_1$  or  $y_2$  after the conditional, when referring to  $y$ , is not defined. In order to overcome this problem, conditional statements are replaced with a  $\Phi$  function. The purpose of the  $\Phi$  function is to map the variables defined in either of the branches of a conditional statement to the same variable but with a new index that can be referenced after the statement. For our program fragment this additional rule leads to the following program:

```
y_1 = 0;
y_2 = 1;
y_3 = Phi((x_0 > 4), y_1, y_2);
```

Note that we assume a function `Phi` available in the programming language representing the function  $\Phi$ , which is defined as follows:  $\Phi(C, x_1, x_2) = \begin{cases} x_1 & \text{if } C \text{ evaluates to true} \\ x_2 & \text{otherwise} \end{cases}$

It is also worth noting that after the second step, the program comprises assignment statements only. Therefore in the final step of conversion we only need to map the assignment statements to equations (or constraints). This step has to take care of the underlying constraint solver. In our case we use MINION [64]. Because of space restrictions we do not discuss the mapping to MINION programs. Instead we use mathematical equations written in italics for representing the arrays, and we assume a constraint solver that is able to compute a solution for a set of equations. From here on we also assume that a function `convert( $\Pi, nd$ )` implements the conversion taking a program  $\Pi$  and a maximum nesting depth  $nd$  as inputs. Figure 4.8 (b)-(d) depicts the conversion results step-by-step when applying `convert` to the multiplication program from Figure 4.8 (a) and using  $nd = 1$  as maximum nesting depth.



```
1. int a, b;  
2. int max = a;  
3. if (a <= b)  
4.     max = b;
```

(a) Original program

```
1. int a, b;  
2. int max = a;  
3. if (a < b)  
4.     max = b;
```

(b) Mutant Program

```
1. int a, b;  
2. int max = a;  
3. if (max <= b)  
4.     max = b;
```

(c) Equivalent mutant

Figure 4.8.: Program for computing the maximum of two numbers

#### 4.4.3. Algorithm

We implement an approach that makes use of the constraint representations of a program and its mutant. The idea is to use the constraints for computing distinguishing test cases, i.e., test cases that allow for differentiating two program runs using the same input. We also discuss challenges and give some initial results when applying our approach to smaller programs. The basic idea behind the equivalent mutant detection algorithm is motivated by the concept of distinguishing test cases introduced for program debugging in [127]. In this paper the authors describe how distinguishing test cases can help to further improve fault localization via extending the available test suite. The equivalent mutant problem can be solved as a byproduct of computing distinguishing tests. If there is no test case that distinguishes the program from its mutant, then the program and its mutant have to be equivalent. Otherwise, if such a test case exists, we cannot certify that both the original program and its mutant have a different behavior.

This is due to the fact that the constraint representation of a program, as introduced in the previous section, is only behavioral equivalent to the corresponding program, for test cases not exceeding the number of considered iterations in while statements. This fact is less problematic in pure fault localization where no further test cases are generated, because the number of iterations to be considered for generating the loop-free program is known in advance. This holds because fault localization requires

at least one failing test case, which determines the number of iterations assuming that the program halts.

As a consequence of this observation it is not always possible to determine equivalent mutants when using distinguishing test cases. However, this is not a surprise because of the undecidability of the equivalent mutant problem. So what are the consequences? In case a distinguishing test case can be computed, we have to check whether this test case is a feasible test case or not. Feasible means that the program and its mutant are able to execute the input of the test case and return the expected output, which can be derived from the solution of the corresponding constraint representation. A non-feasible distinguishing test case is a test case that, when executed on the original program and its corresponding mutant, it does not return the expected output (the one computed by the constraint representation). If the obtained distinguishing test case is feasible, the programs are not equivalent. Otherwise, we have to search for a different distinguishing test case. We can do this by adding the information that the previously computed input is not allowed to be computed anymore. Moreover, there is a second problem due to the number of iterations considered during conversion. If the constraint solver returns no solution, this might be due to the chosen nesting depth. Therefore, we have to increase the nesting depth and start searching again. Of course this cannot be done forever. Therefore in practice we limit the nesting depth to a pre-defined maximum value.

The following algorithm implements the underlying basic idea of checking whether a mutant is equivalent to its corresponding program or not.

**Algorithm equalMutantDetection( $\Pi, M, nd, nd_{max}$ )**

*Input:* A program  $\Pi$ , its mutant  $M$ , the initial nesting depth  $nd$ , and the maximum nesting depth  $nd_{max}$ .

*Output:* `true` if  $\Pi$  is equivalent to  $M$  and `false`, otherwise.

1. Convert the program into its constraint representation:  $CON_{\Pi} = \mathbf{convert}(\Pi, nd)$
2. Let  $M'$  be a program obtained from  $M$  by adding the postfix `_M` to all variables.
3. Convert the mutant into a set of constraints:  $CON_M = \mathbf{convert}(M', nd)$
4. Let  $CON$  be  $CON_{\Pi} \cup CON_M$ .
5. For all input variables  $x$  of  $\Pi$ , add the constraint  $x = x\_M$  to  $CON$ .
6. Let  $y^1, \dots, y^k$  be the  $k$  output variables of  $\Pi$ . Add the constraint  $y^1 \neq y^1\_M \vee \dots \vee y^k \neq y^k\_M$  to  $CON$ .

7. Call a constraint solver on  $CON$  and let  $SOL$  be the set of solutions, e.g., mappings of variables to values that satisfy all constraints in  $CON$ .
8. If there exists no solution  $SOL$ , i.e.,  $SOL = \emptyset$ , then  $\Pi$  and  $M$  are potentially equivalent. In this case do the following:
  - a) If  $nd \geq nd_{max}$ , terminate the algorithm and return `true`.
  - b) Otherwise, increase  $nd$  by 1 and go to 1.
9. Otherwise, there exists a (non-empty) solution  $SOL$ . If there is no variable  $loop_j$  with an assigned value of `true` in  $SOL$ , then return `false`. Otherwise, add the information that the inputs computed in  $SOL$  are not valid to  $CON$  and go to 7.

In the **equalMutantDetection** algorithm lines 1–6 are for constructing the constraint system. In Line 5 constraints are added in order to force the input variables of the program and its mutant to be the same. Line 6 is for stating that a distinguishing test case triggers at least one output variable to hold a different value after execution. Note also that the mutant variables are changed before conversion. Line 7 calls the constraint solver. In Line 8 it is handled the case where no distinguishing test case can be found. This either leads to the result that the mutants are equivalent or to an increase of the considered nesting depth and a re-computation. Line 9 is for handling the case where a distinguishing test case is found. There feasibility is checked via testing whether the number of iterations for computing a solution has been exceeded. If the computed test case is not feasible, search starts again using the information already gathered.

Note that the algorithm always terminates because of the introduced  $nd_{max}$  variable. However, it can be the case that the algorithm returns `true` but there is a distinguishing test case requiring the programs to be executed for a longer time. However, if the algorithm returns `false`, the program and its mutant are definitely different with respect to their semantics. This restriction comes from the undecidability of the underlying equivalent mutant problem.

#### 4.4.4. EqMutDetect Tool

In order to verify the practicability of our approach, but also to automate it, we have implemented the **equalMutantDetection** algorithm. We use a predefined value for the nesting depth, leaving the implementation *step 8 b*, for a new release of our tool.

**EqMutDetect** is a mutation based test case generation tool, entirely developed in Java programming

language. One important feature of the tool is the detection and deletion of equivalent mutants. The tool itself comprises three main parts: (1) *Mutants generation from a given program*, (2) *Equivalent mutants detection and reduction* and (3) *Test case generation*. For mutant generation and execution we rely on an enhanced version of MuJava [205, 134]. Further integration with other mutation testing tool is work in progress, i.e., Jumble [79] and Javalanche [166]. We aim to offer a reliable mutation test case generation tool that handles the equivalent mutant problem as good as possible. We also modified MuJava in order to be JDK 1.5 compliant (see [134]).

The currently used mutation tool produces method level and class level mutations. However, in our research experiments, we excluded object oriented programs and generate only method level mutations, i.e., we make use of the method-level operators defined for MuJava, and mutate statements by replacing, deleting and inserting primitive operators, e.g., arithmetic operator, relational operator, etc. However, we do have some limitations, with respect to the mutations we derive:

- Constant values are not mutated;
- Deletion of statements is not possible; mutants cannot be obtained through statement insertion or deletion;
- We perform only First Order Mutation (this limitation is easy to overcome), i.e., each mutant contains only one change when compared with the original program ;
- Regarding the running of test cases, each JUnit test file must have the nullary constructor.

For equivalence detection and mutation based test case generation we have chosen to work with (and integrate) MINION [64] constraint solver. MINION is an open source constraint solver. The syntax of MINION needs some knowledge when writing the constraints. In most of the situations, the complex constraints must be split into two or three more simpler constraints. Unlike other constraint solver toolkits, MINION does not have to perform an intermediate transformation of the input constraint system. MINION receives as input a file describing an instance of a constraint satisfaction problem, and then tries to solve it. Therefore the original Java program must be transformed to its corresponding CSP representation. In order to obtain this CSP representation, we have to:

1. Eliminate the loops from the original program. We do this unrolling them with a bounded number of nested *if* statements.
2. Convert the unrolled program to its equivalent static single assignment form (SSA representation), i.e., the SSA representation is an intermediate representation which says that each variable is assigned exactly once in a program.

3. Convert the obtained SSA representation into the corresponding constraint representation system.

Nevertheless, we are prone to some limitation in what concerns the MINION constraint solver. We are not able to offer support for object-oriented constructs, therefore we use only traditional mutation operators in our case studies. Moreover, it is worth noting that the approach itself has some limitations. In the first step of the conversion to constraints, we remove all loop statements. The nesting depth is predefined and hence, there might be spurious solutions, i.e., solutions that cannot be executed by the program or the mutant. This might happen because of the fact that there are not enough iterations considered during conversion. This problem can be solved in a bounded fashion by checking for spurious solutions. Checking is performed by executing the test case using program  $P$  and a mutant  $M_i$ . If execution is possible and the results are equivalent to the results obtained from the constraint solver, the test case is a valid test case. Otherwise, the number of iterations has to be increased and a new conversion has to be established for obtaining the corresponding constraint representations.

**EqMutDetect** can be called on an entire set of mutants in order to remove all equivalent mutants. In this case the tool returns a new set of mutants where the equivalent mutants are removed. However, when using **EqMutDetect** the user also has the option to work in a step by step manner, selecting a file and a mutant from the list of corresponding mutants. In the following we use the latter option to run our tool demonstration.

### Case study

Figure 4.9 shows the window of **EqMutDetect** where the user has a look at the source code of the original program and the selected mutant. We apply the conversion of both programs to the MINION input file format. As a result we obtain a constraint system. The part corresponding to the mutants equivalence condition will look as presented below:

```
● **VARIABLES**
  DISCRETE GcdATC_resultGcdATC1_0 -250..6000
  DISCRETE GcdATC_newAGcdATC3_0 -250..6000
  DISCRETE GcdATC_newBGcdATC5_0 -250..6000
  .....
  #Mutant
  DISCRETE GcdATC_ROR_2_resultGcdATC1_0 -250..6000
  DISCRETE GcdATC_ROR_2_newAGcdATC3_0 -250..6000
  .....
  **CONSTRAINTS**
  eq(GcdATC_aGcdATC7_0,GcdATC_ROR_2_aGcdATC7_0)
```

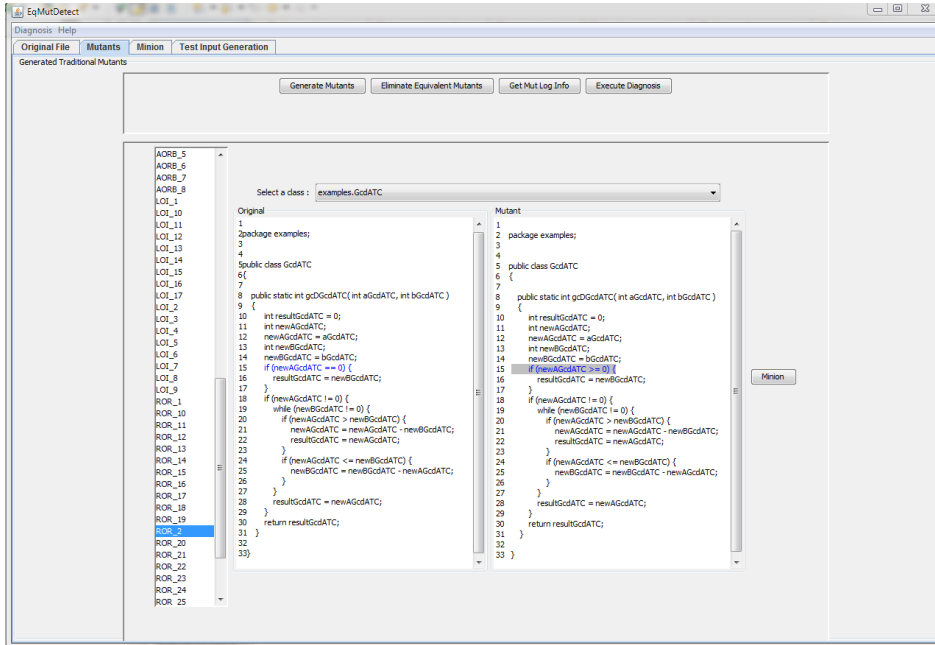
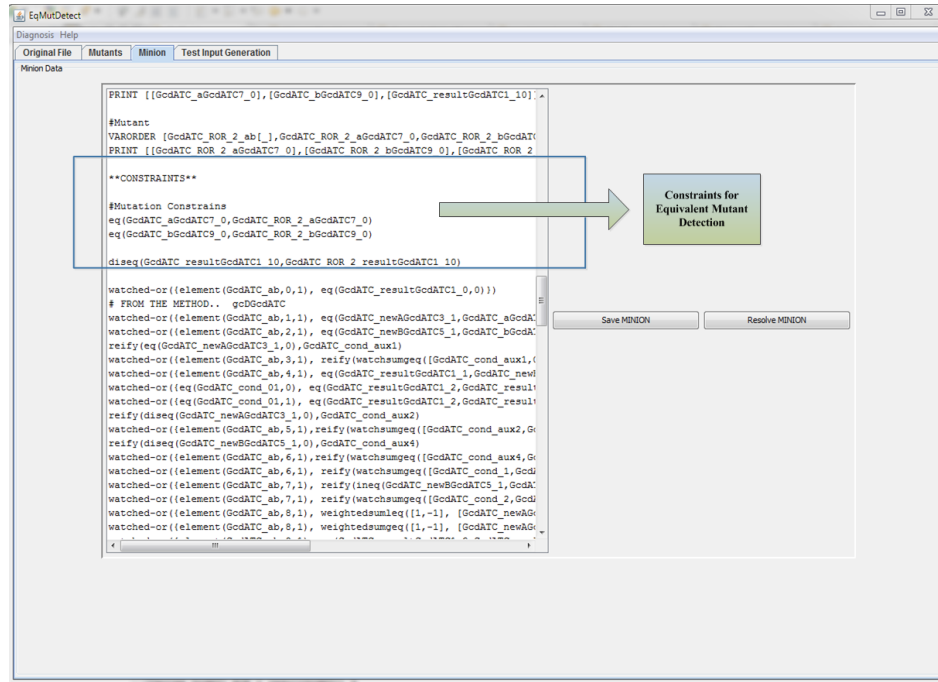


Figure 4.9.: Original Program and Its ROR Mutant

*eq(GcdATC\_bGcdATC9\_0,GcdATC\_ROR\_2\_bGcdATC9\_0)*  
*disseq(GcdATC\_resultGcdATC1\_10,GcdATC\_ROR\_2\_resultGcdATC1\_10)*

watched-or(element(GcdATC\_ab,0,1),  
 eq(GcdATC\_resultGcdATC1\_0,0))  
 watched-or(element(GcdATC\_ab,1,1),  
 eq(GcdATC\_newAGcdATC3\_1,GcdATC\_aGcdATC7\_0))  
 watched-or(element(GcdATC\_ab,2,1),  
 eq(GcdATC\_newBGcdATC5\_1,GcdATC\_bGcdATC9\_0))  
 .....  
 watched-or(eq(GcdATC\_cond\_1,1),  
 eq(GcdATC\_newBGcdATC5\_8,GcdATC\_newBGcdATC5\_1))  
 watched-or(element(GcdATC\_ROR\_2\_ab,0,1),  
 eq(GcdATC\_ROR\_2\_resultGcdATC1\_0,0))  
 watched-or(element(GcdATC\_ROR\_2\_ab,1,1),  
 eq(GcdATC\_ROR\_2\_newAGcdATC3\_1,GcdATC\_ROR\_2\_aGcdATC7\_0))  
 .....  
 watched-or(eq(GcdATC\_ROR\_2\_cond\_1,0),  
 eq(GcdATC\_ROR\_2\_newBGcdATC5\_8,GcdATC\_ROR\_2\_newBGcdATC5\_7))  
 watched-or(eq(GcdATC\_ROR\_2\_cond\_1,1),  
 eq(GcdATC\_ROR\_2\_newBGcdATC5\_8,GcdATC\_ROR\_2\_newBGcdATC5\_1))

Figure 4.10.:  $CS_{PM_i}$  of the Original Program and Its Mutant

In Figure 4.10 we present a snapshot of the system, corresponding to the mutants equivalence condition. We highlight the constraints that we added to the system in order to detect the equivalent mutant. When calling the MINION solver we return that there is no solution. Hence, we conclude that the mutant and the original program are distinct. Therefore, no equivalent mutant is found. Figure 4.11 presents the result obtained from the MINION solver. What we observe is that the constraint solver is not able to find a solution. From this observation we conclude that the mutant and the original program are distinct and no equivalent mutant is found.

#### 4.4.5. Experimental Results

Using the implemented tool **EqMutDetect** we conducted first experiments. For this purpose we applied **EqMutDetect** on some smaller programs varying from 13 to more than 380 lines of code.

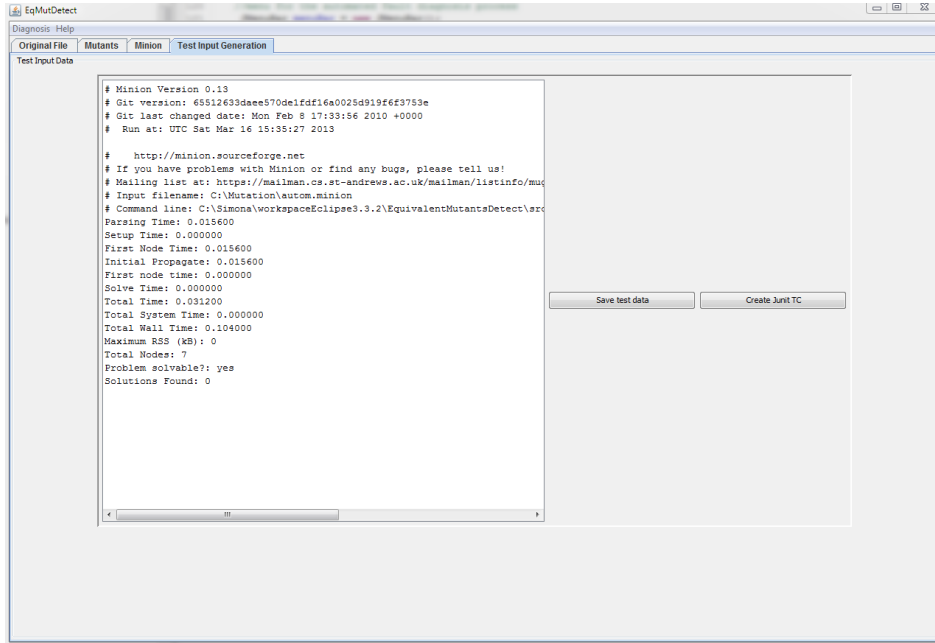


Figure 4.11.: CSP Solution

The used programs include the TCAS files [158], and programs that implement operations on arrays, and mathematical operations. The TCAS files were converted to Java syntax, in order to produce mutations with MuJava. In our experiments the nesting depth for representing loops vary from 2 to 5. For constraint solving we have established a 5 minutes time bound. We executed **EqMutDetect** using the maximum and minimum nesting depth, and observed no difference regarding the obtained solutions. Therefore, we conclude that for the current experiments a small number of iterations is adequate.

In Table 4.9 we summarize the obtained results. **LOC** denotes the lines of code, **No<sub>Mut</sub>** represents the total number of computed mutants, and **Det<sub>EqMut</sub>** is the number of equivalent mutants detected using **EqMutDetect**. We observe from Table 4.9 that in some cases no equivalent mutants were generated, i.e., the constraint solver was always able to find a distinguishing test case. We also encountered the situation where the constraint solver tried to find a solution within the predefined time limit, but it did not manage. In this case we terminated search.

Regarding the quantitative results of our experiments, there are some interesting findings. About 40% of the mutants generated for the *tcas* files were found equivalent when using our approach.



#### 4.4. Equivalent Mutant Detection

Class	LOC	NoMut	DetEqMut
tcas01	125	231	137
tcas02	125	231	139
tcas03	125	231	149
tcas04	125	231	142
tcas05	125	218	149
tcas06	125	236	153
tcas07	125	231	149
tcas08	125	231	149
tcas09	125	231	152
tcas10	125	241	156
tcas11	125	241	149
tcas12	125	231	149
tcas13	125	231	149
tcas14	125	231	147
tcas15	125	218	113
tcas16	125	231	147
tcas17	125	231	139
tcas18	125	231	139
tcas19	125	231	148
tcas20	125	231	149
tcas21	125	241	152
tcas22	125	234	144
tcas23	125	241	139
tcas24	125	234	137
tcas25	125	-	-
...	...	...	...
tcas41	125	229	119
<b>AVG</b>	<b>125</b>	<b>226</b>	<b>137</b>

Class	LOC	NoMut	DetEqMut
C432Order	382	2038	1341
ArrayOperations	101	496	11
BubbleSort	18	62	9
CalculateRectArea	14	18	2
CalculateRectPerimeter	13	21	4
CoffeeMachine	43	91	0
FindEvenOrOddNumber	15	34	3
FindLargestSmallestNumber	19	41	13
GcdATC	35	95	29
NumberFactorial	19	38	12

Table 4.9.: EqMut Detection Results

This happens because we only take into account mutations, which are reachable and have an impact on the return values. Also concerning the tcas files, we found situations when we could not apply our approach, because the mutation engine failed to generate mutants, e.g., for *tcas25*. It is also interesting to note that for the largest program *C432Order* the constraint solver was able to provide solutions within the 5 minutes bound.

## 4.5. Conclusions And Open Issues

Reducing the number of equivalent mutants plays a significant role in determining the efficiency of test suites. The mutation score can be significantly improved when removing all equivalent mutants. In the presented approach we combine constraint representations of programs with mutation testing. In particular we are searching for a distinguishing test case in order to differentiate the program from its mutant. Because of the fact that the underlying problem is undecidable, the approach does not guarantee to find a solution. The influence of certain parameters like the given nesting depth is left for future research. As a byproduct the approach allows for adding new test cases to the test suite. Computed distinguishing test cases can be used to increase the mutation score.

In our current experiments we consider smaller programs. Therefore, the efficiency and scalability of the approach can be disputed. Hence, future research will include improving the experimental bases and using more and larger programs. Moreover, we also want to extend the tool in order to handle object-oriented programs.

We introduced the tool **EqMutDetect** and the experimental results to demonstrate the feasibility of the approach. Eliminating equivalent mutants is necessary in order to improve mutation testing and derive an accurate mutation score. The proposed tool is based on constraint solving and a constraint representation of the program and its mutations. The tool derives new test information, resulting thus an improved test suite. It allows for computing a distinguishing test case that would kill a mutant. In case there is no such test case, the tool identified an equivalent mutant. The distinguishing test cases can be added to the test suite in order to improve its mutation score. The currently implemented **EqMutDetect** can be effectively used for measuring the quality of test suites in software systems, where programs are smaller and usually comprise basic data types like integers or booleans and control structures like loops and conditionals.

Another feature of our tool is that we have integrated an option to derive the diagnosis approach

#### *4.5. Conclusions And Open Issues*

---

described in Section 4.3. That research work was conducted in collaboration with [201, 127]. Thus, we make use of the benefits from debugging in testing.



# Chapter 5

## Mutation Testing from An End User Perspective

*"Anyone who has never made a mistake has never tried anything new."* - Albert Einstein

One of the most encountered programming paradigms nowadays is end user programming (EUP). Common EUP applications are spreadsheets, modelers, database users, etc. In EUP, end users derive applications to achieve their needs, but, usually, they do not have solid notions of software programming techniques, e.g., accounts, teachers, health care personnel, architects, health care users (these write formal specification to generate medical reports), etc. End user testing (EUT) differs from normal software testing, as end users are usually people trained in other areas but software development. Therefore detecting misbehavior is important, as severe damages might appear, e.g., a cut and paste mistake in an excel spreadsheet made one important power generator company, TransAlta, to buy power transmission in excess; this caused higher contract costs than normally [41]. Another important aspect in end user testing is that users will be more confident when considering the correctness of their application. The growing in EUT has lead also to a growing in end user testing and debugging, resulting thus several strategies. The current chapter represents an extension to our research in mutation based test case generation and it gives us a future research direction towards the area of mutation testing. It presents a novel approach, which is currently under a peer-review process. Therefore, this early stage work offers plenty of room for further research, development and improvement.

In what follows we will present a short overview on the current available end user testing strategies, describing some of the the most commonly used techniques in end user testing. Then we will describe the first preliminary algorithms and the future steps to take in our research.

## 5.1. End User Testing Methods

Spreadsheets represent one of the most known end user applications [165]. They are files, which based on input values, compute outputs, by means of formulas, in accordance with certain user requirements. For example, consider a teacher who has to compute the grades for a group of students, based on given scores and rules for computing, i.e., formulas. The authors in [91] propose a classification for the end user testing methods, according to the different modalities through which end users test: by means of oracles, software testing, specifications checking, consistency checking or visualizations. One simple method used for verifying the output values of a program is based on an oracle, which is able to determine whether the output is correct or not. The oracles can be represented by people, specifications, well documented definitions, etc. In End User Engineering, the end users act as the oracle. In [99], it was proven that people are imperfect oracles, as professional programmers tend to be overconfident, while end users tend to exceed the overconfidence peak [148]. Therefore, a solution for end user programmers, to reveal errors, is through software testing. Systematic testing, by means of a well designed testing plan, is not suitable for the end user programmers, who must test in an incremental manner. Therefore end user testing approaches must take into account this aspect. The authors in [160] propose a form based testing strategy for visual programs. These programs can be used by a wide range of people, from software engineers to end users; e.g., the electronic spread sheets system is an example of such a program. Users of this type of programs create cells, manipulate cell information, create formulas, etc. Spreadsheets are among the most widely used applications. Therefore, assessing their reliability is important. The authors prove that testing these systems is different from testing imperative programs. According to the authors, due to their cell dependency property, the testing strategies are based on an adequacy criteria. They define a testing criteria for the formulas and then for the cell dependency. Studies demonstrate that these types of systems are prone to faults involving incorrect formulas, incorrect or missing references. Syntax error faults or references to non existing cells can be detected when the programs are created. The testing strategies for the end user programs are based on code testing, where the oracle is actually the end user. The authors define the end user program as a function that maps the vector of input cells to the vector of output cells,

over a domain  $D$ . Testing these programs assumes initializing the input cells and afterwards executing them. In a spreadsheet system, when a cell is changed, the system automatically updates every input and results are automatically computed. The authors define this as the responsiveness characteristic of a form based visual program. Therefore, this leads to an incremental testing strategy: incrementally altering the values from the input cells, and then validate the values of the corresponding output cells. As control flow depicts the code based test adequacy criteria, in visual form based programs this assumes deriving a cell evaluation order criteria, independent from the engine, i.e., end users have no programming knowledge. In order to derive testing adequacy criteria the users make some assumptions. They use input cells, i.e., input vector of the program, and non input cells, cells that can require validation. According to their assumptions, input cells do not reference other cells, cell references are static, recursion is not allowed and formulas cannot constitute a testing input. An abstract model of such a system includes the form based visual program  $P$  with a cell relation graph CRG, i.e., control flow of the formulas and cell dependencies from  $P$ . A cell is modeled as a formula directed graph, where a node is an expression from a cell, and an edge is the control flow between expressions. For two cells  $A$  and  $B$ , we say that  $B$  is cell-dependent with  $A$ , if the formulas from  $B$  reference  $A$ . In CRG, cell dependencies are edges between the formulas graphs. As exhaustive testing cannot be derived, the authors define three testing adequacy criteria, to establish when enough testing has been conducted:

1. Node and edge adequacy criteria assume execution of each node and edge from the graph. A test  $t$  for node  $N$  from formula graph  $G$  must trigger the execution of the formula corresponding to  $G$  and the path to that formula must include  $N$ . If for each formula graph of  $P$  and each executable node  $N$  in the graph, there is at least one test that triggers  $N$ , then this is a node adequate test suite  $T$ . If for each formula graph of  $P$  and each executable edge between two nodes in the graph  $G$ , there is at least one test that triggers the two nodes, then we have an edge adequate test suite  $T$ .
2. Cell dependence adequacy criteria. From the first criterion, we observe that when nodes and cells are executed, then also interactions between the cells are triggered. Therefore a second criterion involves testing all the interactions between cells. Let us assume two cells,  $C1$  and  $C2$ , belonging to  $P$  and each having a formula graph,  $CG1$  and  $CG2$ . If cell  $C2$  depends on  $C1$ , then a test  $t$  will derive the cell dependence edge between  $CG1$  and  $CG2$  if  $t$  determines the evaluation of  $CG2$ , and the evaluation goes through path  $CG2$  which contains a node whose expression references  $C1$ . If at least one such test exists then there exists a cell dependence

adequate test suite.

3. Data flow adequacy criteria considers the coverage of all the cell dependencies inside the graph CRG and the effect of the control dependencies which appear between expressions. For a cell  $C$ ,  $F$  represents the formula of  $C$ . An expression in  $F$  that sets the value of  $C$  is a definition of  $C$ . An expression in  $F$  that refers another cell  $C1$ , is a use of  $C1$ . The authors define:

- The predicate use, *p-use*, which directly alters the output of a predicate expression;
- The computation use, *c-use*, which does not directly alters the output of a predicate expression, but appears in the predicate expression.

The authors then determine definitions-use associations (du-associations), i.e., definition -c-use association and definition -p-use association, which will reflect the use cases that those definitions of cells will reach, i.e., try to execute every data dependency that could be executed. Such an association maps each expression from a cell formula, which defines the value of the cell, to the expressions from other cell formulas that use the defined cell. The criterion requires that each executable du-association will be triggered by the test data so that the du-association will directly or indirectly affect the monitor of a value that is assumed to be valid by the program. These three adequacy end user testing criteria are later implemented in a prototype tool [159]. The methodology is called WYSIWYT (What You See Is What You Test). This is a white box spreadsheet testing technique, where the end user can incrementally and systematically validate and test the spreadsheet. The tool implementing the data dependency adequacy criteria uses some decision boxes that the end user must validate. The decision box will be automatically checked when the value of a cell is validated against the given inputs. Beside data dependency, the tool also displays a percentage of the testing advancement. In a later paper, [161], the authors empirically evaluate the WYSIWYT prototype, using computer science students as the end users, which have no knowledge on the environment. They divided the users in a group using the WYSIWYT methodology and a group using an ad-hoc strategy. The authors analyze the results from the effectiveness, efficiency and overconfidence point of view. The group using the WYSIWYT strategy was more effective and efficient than the second group, but less overconfident. In Figure 5.1, we present an example, extracted from [25]. The spreadsheet is used for computing the average grade of a student, for a given course. According to the authors, a complete tested cell will be painted in blue; red is used for designing an untested cell, and another color, different than red or blue, for partially tested cells. The check marks depict a validated value against the given input values. Also, in the right upper corner of the figure, there is a progress bar which depicts how much has been tested.



	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	B
2	Farnes, Joan	7,649	92	94	92	92.6	A
3	Green, Matt	2,314	78	80	75	77.4	C
4	Smith, Scott	2,316	84	90	86	86.6	B
5	Thomas, Sue	9,857	89	89	89	93.45	A
6							
7	AVERAGE		86.4	88.8	85.6	87.69	

Figure 5.1.: Spreadsheet for average grade computation, taken from [25]

In [62], the authors, based on the WYSIWYT methodology, derive an automatic test generation process for spreadsheets. The user has the possibility to select a certain combination of cells and then make use of the Help Me Test utility, in order to generate test scenarios. First, the tool determines the set of all valid du-associations, corresponding to the user request. Then it computes the set of input cells and based on all these, it tries to generate a test case, exploring those paths from the data flow that were not yet taken into consideration. After finding the test case, the tool will do a validation on the output cells. In case a spreadsheet modification appears, the already existing test cases can be reused, i.e., regression testing [76]. The strategy uses a specific data structure to keep the information, involving:

1. Cell information (name, value, trace of the recent cell execution);
2. Test case information (input cell, validated cell from a test, reached cell through the test, validated du-associations);
3. Spreadsheet information (table of validation counts, given by the du-associations, the set of impacted tests that can be reused).

When a cell validation is done, the tool goes back through the CRG graph of the given spreadsheet, updates and saves the information according to the data structure presented above. The type of changes can affect input values, intermediate and output ones, and are represented by cell deletion or insertion, formula change. When a change appears, the impacted test cases are saved and re-run. All the described end user testing strategies are based on what you see is what you test concept. A recent report on the state of the art on End User Programming,[91], states that both testing and debugging are based on the WYSIWYT, which currently represents the most reliable error detection method in End User Programming.

Another technique used for verifying end user programs is based on data validation. The authors in [164] propose a new tool for data validation, inside spreadsheets and web forms. Through the implemented set of tools, the users have the possibility to distinguish between valid data, invalid data and data that could be either valid or invalid. Another possibility is to check the output values against a set of well defined specifications. One type of specifications can be represented by the assertions testing strategy described earlier. Another approach relies on storing the end user data sessions in order to later use this information in the testing process [55]. Nevertheless, the WYSIWYT is the most mature end user testing strategy and later research works use this strategy as basis for testing and debugging, e.g., [21, 96], combined with machine learning [95]. Another approach in end user testing is based on metamorphic testing, i.e., MT, which was first introduced by the authors in [29]. According to [29], "MT generates follow-up test cases, making reference to metamorphic relations (MR)", i.e., "relations among multiple executions of the target program", and in the end the test results are verified against the metamorphic relations. As metamorphic relations are used, the oracle problem is no longer taken into consideration. In [29, 30], the authors demonstrate that MT is effective in fault detection. In [102], the authors make use also of the MT technique for testing end user programs. They conclude that metamorphic testing is efficient for EUP, as no software testing knowledge is required, but only the knowledge of the user. According to this, the end user must determine the properties of the system, in conformance with the end user domain specifications. Later, using the determined properties, the MR relations are constructed between the inputs and outputs, obtaining thus follow up test cases. Afterwards, the test results will be checked against the metamorphic relations. In metamorphic testing, there is always needed an extra traditional software testing method, e.g., random testing, in order to compare the metamorphic testing results with the traditional testing outputs. If they broke the metamorphic relations, the authors state that an error is discovered. Along their research paper, the authors teach and experiment this technique to a class of undergraduates and postgraduates students, during three years. They conclude that this is a reliable testing technique for EUT, simple to use, i.e., students understood easily the concept of MT, were able to identify the metamorphic relations without difficulties, different faults were detected by different groups of students, an automation of MT was simple, without a tool support. Another testing alternative is to consider mutation testing. In [1], the authors propose a set of mutation operators for spreadsheets, apply mutation testing over different spreadsheet applications and do an empirical evaluation concerning the fault detection capability towards mutation. Generally, mutation testing is used to compute the effectiveness of a test suite. For this situation the the mutation score metric is defined as the percentage of the spreadsheet mutations detected by the test suite. Mutations represent variants of the initial spreadsheet, obtained

when applying different mutation operators, e.g., arithmetic operators, relational operators, etc, in a spreadsheet cell.

But the End User Engineering is not limited only to the spreadsheet domain. Many of the end user applications reside on the Internet. Consequently, testing was a concern also for this domain; therefore different web testing tools were designed. We enumerate some of the most known:

- *Selenium*: It is a collection of tools which offer support for many platforms, e.g., Java, Javascript, Ruby, PHP, Python, Perl and C#. Selenium was designed as a Firefox plugin, having a friendly user interface, which eases the development of automated tests. It records user actions and then exports them as a reusable script. An interesting feature is that tests can be run in parallel [174].
- *TOSCA Testsuite*: A commercial tool for automated web based testing tool, which implements different testing strategies; i.e., GUI and non GUI testing, load testing, regression testing, data warehouse testing, etc. It computes test efficiency using test coverage information (resulted from test execution) [175].
- *Woodstein*: Software agent, which works both on a computer and a web browser. The tool records actions performed by a user on a web page, then tries to come up with different possible answers, e.g., *why does this happen?*, *Why did that happened?*, *How did that data get that values?*, i.e., it computes diagnosis information for the end user [187, 188, 101, 186].
- *IBM Rational Functional Tester*: Commercial automated functional web testing tool, which performs regression testing. The tool is able to simulate a human end user, by imitating human actions, assessments. It is an easy to use tool, both for professional developers, but also for non professionals, e.g., end users. The tool has also integrated advance debugging feature and SAP(Systems, Applications and Products in Data Processing) automation test support [77].
- *Telerik Test Studio*: Performs different types of testing, e.g., functional testing, load testing, mobile application testing, exploratory testing. It supports ASP.NET, JavaScript, HTML, AJAX and also integration with Microsoft Visual Studio 2008 and 2010, NUnit, XUnit, MbUnit unit test cases. It runs by recording actions from a Web page (only for Internet Explorer) [173].

Testing can be done at every level of the software development life cycle. Important for the end users might be the GUI testing and the requirements/specifications testing (test that the end user application/script does what it is supposed to do). That means the testing process can be conducted from unit testing, integration testing, system testing, acceptance testing and up to user acceptance testing (the user acceptance testing is more closely to our approach).

## 5.2. End User Mutation Testing

End users have different views on the outputs an application should compute, but, nevertheless they might help in discovering faults. If we consider the end user testing process, we state that it can actually be derived from the standard testing process:

- Create unit tests and, for each new functionality, run the unit tests;
- Perform code inspection;
- Based on specifications, perform functional testing;
- If functionality testing is fulfilled, then actual end users can test the application.

End user feedback is extremely important, as they are responsible for deciding whether the application covers its requirements. Therefore a thorough discussion with the EU must exist, in what concerns the technical experience. The end user must be familiar with the tool/application, e.g., organize tool trainings. From the end user point of view, the application is like a black box, therefore documentation is important for the end user, i.e., create check lists, maybe possible/detected defects, in order to draw the attention where critical parts may appear.

We will make use of the definitions presented in Chapter 2.1, and, consequently, extend them to the end user engineering domain, in particular to the spreadsheet domain. For this research direction, we define a program  $P$  as being represented by a spreadsheet, which must contain a set of non empty cells. Each cell can be defined by an expression, i.e., a formula, or it can be referenced by another cell. We can model a spreadsheet as a direct acyclic dependency graph, with vertices and edges.

Directed acyclic graphs are a good solution for expressing the dependencies between the cells of a spreadsheet. An edge will be represented by a path between cells, i.e., vertices of the graph. If a cell expression makes use of another cell, then we say that we have an edge, i.e., we say that cell  $A$  and  $B$  are dependent *if and only if* there is a path from cell  $A$  to  $B$ . Therefore, the inputs are those vertices with no incoming edges and the outputs are defined as the vertices with no outgoing edges. In Figure 5.2 we present a spreadsheet, which computes a feet to meter conversion. The associated directed acyclic graph is depicted in Figure 5.3.

A test case is defined as a pair  $(cells, expectVals)$ , where  $cells$  is the set of input cell values and  $expectVals$  represents the expected value for the output cell(s). A *passing test case* is a test for which the computed output cell values are the same with the expected ones. The definition of the failing test case, presented in section 2.1, will remain unchanged, i.e., a *failing test case* is a test case for which

Inch	Centimeters
2,00	5,08
6,00	15,24
6,50	16,51
10,00	25,40
23,00	58,42

Figure 5.2.: Spreadsheet for feet to meter conversion

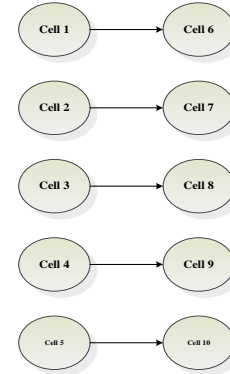


Figure 5.3.: Graph representation of spreadsheet from 5.2

the output environment, i.e., output cells, computed from the program  $P$ , i.e., the given spreadsheet, executed over input  $I$ , i.e., input cells, is not consistent with the expected output.

The authors in [1] have introduced the set of mutation operators, described in Table 5.1, in order to apply and run end user mutation based testing.

### 5.2.1. Algorithms And Experimental Results

In order to demonstrate the applicability of our approach to the end user engineering domain, we will make use of the algorithms described previously, in chapter 4, section 4.3. Also, in what concerns the research towards debugging, we will make use of the work conducted in [2]. Similar to our initial approach, we choose to work, through our experiments, with the MINION constraint solver. The difference between the research from chapter 4 and the current chapter, consists in the fact that we run the research on spreadsheets, therefore the input and output algorithm parameters must be correspondingly adapted. Thus:

- **Algorithm Filter\_TestCase** ( $DiagCells, SP_i, TC$ )

*Inputs:* The set of diagnosis cell candidates  $DiagCells$ , the faulty spreadsheet  $SP_i$ , and the failing test case  $TC$ .

*Outputs:* A set of mutated spreadsheets  $Mut_{SP_i}$  for the faulty spreadsheet  $SP_i$ .

1. Let  $Mut_{SP_i}$  be the empty set.

Operator	Description
ABS	ABSolute Value Insertion
AOR	Arithmetic Operator Replacement
CRP	Constants RePlacement
CRR	Constants for Reference RePlacement
LCR	Logical Connector Replacement
ROR	Relational Operator Replacement
RCR	Reference for Constant Replacement
FDL	Formula DeLetion
FRC	Formula Replacement with Constant
RFR	ReFerence Replacement
UOI	Unary Operator Insertion
CRS	Contiguous Range Shrinking
NRS	Non-Contiguous Range Shrinking
CRS	Contiguous Range Expansion
CRE	Contiguous Range Expansion
NRE	Non-Contiguous Range Expansion
RRR	Range Reference Replacement
RRR	Formula Function Replacement

Table 5.1.: Spreadsheet Mutation Operators

2. For all cells  $cell \in DiagCells$  do:
  - a) Generate all mutants of spreadsheet  $SP\_i$  with respect to the faulty cells stored in  $cell$  and then save them in  $NewSP_{Mut}$ .
  - b) Add to  $Mut_{SP\_i}$  every mutated spreadsheet  $SP\_i' \in NewSP_{Mut}$ , which passes the test case  $TC$ .
3. Return  $Mut_{SP\_i}$ .

The last step of our algorithm comprises the integration of distinguishing test cases to further reduce the bug candidate set. Let  $Mut_{SP\_i}$  be the set of mutants for the spreadsheet  $SP\_i$  obtained after applying the **Filter\_TestCase** algorithm. And let  $CON_{Mut_{SP\_i}}$  be the constraint representation of the programs from  $Mut_{SP\_i}$ . Then, the test case generation algorithm is:

• **Algorithm TestCase\_Generator**  $Mut_{SP\_i}, CON_{Mut_{SP\_i}}$

*Inputs:* A set of valid cells repair options,  $Mut_{SP\_i}$ , for a faulty spreadsheet  $SP\_i$  and their constraint representation  $CON_{Mut_{SP\_i}}$ .

*Outputs:* Subset of  $Mut_{SP\_i}$ .

1. Let *Tested* be empty.

- 
2. If there exists mutants  $SP_{i'}, SP_{i''} \in Mut_{SP_i}$  with  $(SP_{i'}, SP_{i''}) \notin Tested$ , add  $(SP_{i'}, SP_{i''})$  to *Tested*, then proceed with the algorithm. Otherwise, return  $Mut_{SP_i}$
  3. Let  $CON_{SP_{i'}}$  and  $CON_{SP_{i''}} \in CON_{Mut_{SP_i}}$  be the constraint representation of programs  $SP_{i'}$  and  $SP_{i''}$  respectively.
  4. Let  $CON_{TC}$  be the constraints encoding  $Input_{SP_{i'}} = Input_{SP_{i''}} = I \wedge Output_{SP_{i'}} \neq Output_{SP_{i''}}$
  5. Solve the CSP:  $CON_{SP_{i'}} \cup CON_{SP_{i''}} \cup CON_{TC}$  using a constraint solver.
  6. Let  $O$  be the correct output for the original program  $SP_i$  on input  $I$
  7. If  $Output_{SP_{i'}} \neq O$ , then delete  $SP_{i'}$  from  $Mut_{SP_i}$ .
  8. If  $Output_{SP_{i''}} \neq O$ , then delete  $SP_{i''}$  from  $Mut_{SP_i}$ .
  9. If no solution, go to 1
  10. For all  $SP_{i'} \in Mut_{SP_i}$  do:
    - a) If  $SP_{i'}$  fails on generated test case  $(I, O)$  delete  $SP_{i'}$  from  $Mut_{SP_i}$ .
  11. Return  $Mut_{SP_i}$ .

We have manually produced small mutations and then applied the initial algorithms on very simple spreadsheets, i.e., spreadsheets whose cells contained only constants, basic formula operations, e.g., addition, subtraction, multiplication, etc. Due to time constraints, this aspect of our research was not roughly tested, nor implemented.

### 5.3. Conclusion

The area of End User Engineering becomes more and more widespread, therefore deriving an efficient testing tool for non professional developers, as seldom end users are, is an essential aspect in software engineering. As mutation testing has proved his efficiency in revealing errors, but also in specific fault based test case generation, we state that it can, indeed, represent a strong option, being aware, nevertheless, of the known limitations.

Towards end user based mutation testing, a future work and challenging, in the same time, is the development of a tool implementing constraint based mutation testing generation. This will have a useful and practical direction, in what concerns, at least, the spreadsheet end users.





# Chapter 6

## Conclusions

*"It is good to have an end to journey toward;  
but it is the journey that matters, in the end."*

- Ernest Hemingway

Since its beginnings, mutation testing has been an intensively researched topic, and even nowadays there is an important research conducted towards this direction. Its applicability domain is continuously growing. From traditional software programming to end user development, mutation testing can be successfully utilized to assess a test suite adequacy.

Through our research, we have contributed towards proving why mutation testing can be a good measure to verify a test suite effectiveness, not only in the research domain, but also when applied in industry. Nevertheless, limitations do exist and if they are properly taken into consideration, then mutation testing can be successfully configured. One of the most known encountered problems in mutation testing is the equivalent mutants problem. We have proposed a viable solution for equivalent mutants detection and elimination, by means of the constraint programming paradigm and the distinguishing test cases concept. The results we obtained, when applying our algorithms in a traditional software development environment, encouraged us to go beyond traditional programming and extend our research towards end user software engineering. End user software engineering is one of the most widely used programming technique, as both professional and non-professional software developers, i.e., end users, can benefit of it.

We have introduced the **EqMutDetect** tool, for equivalent mutants detection and reduction. The proposed tool runs based on constraint solving and a constraint representation of the program and its mutations. It computes a distinguishing test case that is able to kill a mutant. When no such test case is identified, we say that the tool detected an equivalent mutant. The identified distinguishing test cases can then be added to the test suite in order to improve its mutation score. Thus, our tool contributes to one significant problem in mutation testing and, also, it derives a new and improved test suite, helping in test case generation. The results we have obtained demonstrate the feasibility of the approach. We can resume, based on the derived research experiments, that our approach is useful, but more important, that we have gained important feedback and the experience we have accumulated can further on be used in our future research. Nevertheless, we are aware that its scalability remains an open topic. Our goal was to elaborate a reliable strategy and also implement a tool, so that we could make use of the advantages given by mutation testing, into a real size software application.

Regarding my personal idea on how future research might look like, two challenging directions come into discussion. The first of them, and the one that justifies all the effort I have invested in the last years, is the applicability of mutation testing in a real industry setting. Having some background also as a software test engineer, I find a useful applicability of mutation testing in the automotive industry. Deriving a secure and stable test design procedure is crucial for this industry domain. Hardware fault injection techniques have been successfully used in industry, why not apply this fault injection technique (mutation testing), towards software testing of critical systems? Embedded systems are critical systems in need for solid testing.

Our goal is to depict a reliable automated testing process. By this, I think of an efficient, mutation orientated, test suite. Why the embedded system area? Simple: whether we want it or not, everyone of us has become addicted to this area; from the daily underground railway system which, for some of us, is the fastest way of locomotion, towards personal automobiles and, even more, towards the avionics industry, we all use them as a natural think in our existence, without wondering so often: "is there a problem?. They must be functioning good. "

That is why I strongly believe that with lots of effort and, nevertheless, motivation invested towards our approach, we could be able to certify that mutation testing can be scalable for real-world software projects. We have the basis, we have a good background, we have proven the practicability of our algorithms, we even have access to specific fault taxonomies for these type of systems, so designing the failures these systems should stand up, becomes a matter of software programming.

There is a second future research direction, in the area of End User Testing. This is a novel domain,

---

which is gaining more and more interest. It would be challenging for the not-experienced end user programmer, to check his application against certain types of common known human errors. Think only of a monthly work report: what happens if one month you receive no salary at all, due to an overseen formula error? Considering these two directions, we believe that mutation based test case generation is of valuable interest and help.



# Bibliography

- [1] ABRAHAM, R. AND ERWIG, M. 2009. Mutation operators for spreadsheets. *IEEE Transactions On Software Engineering Journal* 35, 94–108. (Cited on pages 110 and 113.)
- [2] ABREU, R., RIBOIRA, A., AND WOTAWA, F. 2012. Constraint-based debugging of spreadsheets. In *CIbSE*. 1–14. (Cited on page 113.)
- [3] ABREU, R., ZOETEWIJ, P., AND VAN GEMUND, A. J. 2006. On the accuracy of spectrum-based fault localization. In *Proceedings TAIC PART'07*. IEEE, 89–98. (Cited on pages 69, 71, and 72.)
- [4] ABREU, R., ZOETEWIJ, P., AND VAN GEMUND, A. J. 2009. Spectrum-based multiple fault localization. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 88–99. (Cited on page 72.)
- [5] ACREE, A. T., BUDD, T. A., DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1979. Mutation analysis. techreport GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia. (Cited on page 39.)
- [6] AGRAWAL, H., DEMILLO, R. A., HATHAWAY, B., HSU, W., HSU, W., KRAUSER, E. W., MARTIN, R. J., MATHUR, A. P., AND SPAFFORD, E. 1989. Design of Mutant Operators for the C Programming Language. techreport SERC-TR-41-P, Purdue University, West Lafayette, Indiana. (Cited on page 47.)
- [7] AMMANN, P. AND OFFUTT, J. 2008. *Introduction to Software Testing*, 1 ed. Cambridge University Press, New York, NY, USA. (Cited on pages 20, 41, and 52.)

- [8] ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. 2005. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. St Louis, Missouri, 402–411. (Cited on page 64.)
- [9] ANDREWS, J. H., BRIAND, L. C., LABICHE, Y., AND NAMIN, A. S. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 608–624. (Cited on pages 37 and 52.)
- [10] ATLISSIAN. Clover. <http://www.atlassian.com/software/clover/>. (Cited on pages 57 and 58.)
- [11] B. BOGACKI, B. W. (Cited on page 45.)
- [12] BEIZER, B. 1995. *Black-box Testing: Techniques For Functional Testing Of Software And Systems*. San Val, Incorporated. (Cited on page 25.)
- [13] BINKLEY, D. AND HARMAN, M. 2004. A survey of empirical results on program slicing. In *Advances in Software Engineering – Advances in Computers Vol. 62*, M. Zelkowitz, Ed. Academic Press Inc., 106–172. See also [citeseer.ist.psu.edu/661032.html](http://citeseer.ist.psu.edu/661032.html). (Cited on page 72.)
- [14] BOGACKI, B. AND WALTER, B. 2007. Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing. In *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed. IFIP International Federation for Information Processing, vol. 227. Springer US, 273–282. (Cited on page 48.)
- [15] BOND, G. W. 1994. Logic Programs for Consistency-Based Diagnosis. Ph.D. thesis, Carleton University, Faculty of Engineering, Ottawa, Canada. (Cited on page 70.)
- [16] BOND, G. W. AND PAGUREK, B. 1994. A Critical Analysis of “Model-Based Diagnosis Meets Error Diagnosis in Logic Programs”. Tech. Rep. SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada. (Cited on page 70.)
- [17] BRADBURY, J. S., CORDY, J. R., AND DINGEL, J. 2006. ExMAN: A Generic and Customizable Framework for Experimental Mutation Analysis. *Workshop on Mutation Analysis 0*, 4. (Cited on page 49.)
- [18] BRADBURY, J. S. AND DINGEL, J. 2006. ExMAN - An Experimental Mutation Analysis Framework. <http://faculty.uoit.ca/bradbury/exman/>. (Cited on page 49.)
- [19] BRADY, P. 2009. Mutateme. <https://github.com/padraic/packman>. (Cited on page 50.)
- [20] BRANDIS, M. M. AND MÖSSENBÖCK, H. 1994. Single-pass generation of static assignment form for structured languages. *ACM TOPLAS 16(6)*, 1684–1698. (Cited on page 92.)

- [21] BROWN, D., BURNETT, M., ROTHERMEL, G., FUJITA, H., AND NEGORO, F. 2003. Generalizing wysiwyf visual testing to screen transition languages. *Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments*. (Cited on page 110.)
- [22] BUDD, T. AND SAYWARD, F. 1977. Users Guide to the Pilot Mutation System. techreport 114, Department of Computer Science, Yale University. (Cited on page 47.)
- [23] BUDD, T. A. 1980. Mutation Analysis of Program Test Data. Ph.D. thesis, Yale University. (Cited on page 43.)
- [24] BUDD, T. A., DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1980. Theoretical And Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '80. ACM, New York, NY, USA, 220–233. (Cited on pages 41 and 71.)
- [25] BURNETT, M. 2009. What is end-user software engineering and why does it matter? *Proceeding of the 2nd International Symposium on End-User Development (IS-EUD '09)*, 15–29. (Cited on pages 6, 108, and 109.)
- [26] CAMPBELL, C., GRIESKAMP, W., NACHMANSON, L., SCHULTE, W., TILLMANN, N., AND VEANES, M. 2005. Testing concurrent object-oriented systems with spec explorer. In *Proceedings of the 2005 international conference on Formal Methods*. FM'05. Springer-Verlag, Berlin, Heidelberg, 542–547. (Cited on page 33.)
- [27] CEBALLOS, R., GASCA, R. M., VALLE, C. D., AND BORREGO, D. 2006. Diagnosing errors in dbc programs using constraint programming. *Lecture Notes in Computer Science 4177*, 200–210. (Cited on pages 69 and 71.)
- [28] CENTER, A. Q. 2012. BACTERIO Mutation Test System. <http://www.alarcosqualitycenter.com/index.php/products/bacterio>. (Cited on page 49.)
- [29] CHEN, T., CHEUNG, S. C., AND YIU, S. 1998. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology 20*, 4, 105. (Cited on page 110.)
- [30] CHEN, T. Y., TSE, T. H., AND ZHOU, Z. Q. 2003. Fault-based testing without the need of oracles. *Proceedings of Information and Software Technology*. (Cited on page 110.)
- [31] CHEVALLEY, P. AND THÉVENOD-FOSSE, P. 2002. A Mutation Analysis Tool for Java Programs. *International Journal on Software Tools for Technology Transfer 5*, 1 (November), 90–103. (Cited on page 48.)

- [32] COBERTURA. 2005. Download Site. <http://cobertura.sourceforge.net/>. (Cited on page 55.)
- [33] COLES, H. 2011. PIT Mutation Testing. <http://pitest.org/>. (Cited on page 48.)
- [34] COLES, H., JEDYNAK, M., ZAJACZKOWSKI, M., GLOVER, P., AND VICTOOR, A. 2011. PIT Ecosystem. <http://pitest.org/links/>. (Cited on page 48.)
- [35] COLLAVIZZA, H. AND RUEHER, M. 2007. Exploring Different Constraint-Based Modelings for Program Verification. In *In Principles and Practice of Constraint Programming (CP 2007)*. Providence, RI, USA, 49–63. (Cited on page 72.)
- [36] COLLAVIZZA, H., RUEHER, M., AND HENTENRYCK, P. 2010. Cpbpv: a constraint-programming framework for bounded program verification. *Constraints 15*, 238–264. (Cited on page 91.)
- [37] COMMUNITY. 2002. JBoss Application Server. <http://www.jboss.org/jbossas>. (Cited on pages 52 and 53.)
- [38] CONSOLE, L., FRIEDRICH, G., AND DUPRÉ, D. T. 1993. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings 13<sup>th</sup> International Joint Conf. on Artificial Intelligence*. Chambéry, 1494–1499. (Cited on page 70.)
- [39] COULTER, A. 1999. Graybox software testing methodology: embedded software testing technique. In *Digital Avionics Systems Conference, 1999. Proceedings. 18th*. Vol. 2. 10.A.5–1–10.A.5–8 vol.2. (Cited on page 33.)
- [40] COULTER, A. 2000. Graybox software testing in the real world in real-time. *Proceedings of STAREAST 2000*. (Cited on page 33.)
- [41] CULLEN, D. 2007. Errors in excel. (Cited on page 105.)
- [42] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS 13*, 4, 451–490. (Cited on page 76.)
- [43] DEBROY, V. AND WONG, W. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*. Paris, France, 65–74. (Cited on pages 67 and 71.)
- [44] DELAMARE, R., BAUDRY, B., AND LE TRAON, Y. 2009. AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors. In *Proceedings of the IEEE International Conference*



- on Software Testing, Verification, and Validation Workshops*. ICSTW '09. IEEE Computer Society, Washington, DC, USA, 200–204. (Cited on page 50.)
- [45] DELAMARO, M. E. 1993. phdthesis. Ph.D. thesis, University of São Paulo, Sao Paulo, Brazil. (Cited on page 47.)
- [46] DELAMARO, M. E., MALDONADO, J. C., AND MATHUR, A. P. 2001. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering* 27, 3 (May), 228–247. (Cited on page 47.)
- [47] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11, 34–41. (Cited on pages 34, 38, and 39.)
- [48] DEMILLO, R. A., PAN, H., AND SPAFFORD, E. H. 1996. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*. 121–134. (Cited on page 73.)
- [49] DENMAT, T., GOTLIEB, A., AND DUCASS, M. 2007. Improving constraint-based testing with dynamic linear relaxations. In *In 18th IEEE International Symposium on Software Reliability Engineering (ISSRE 2007)*. (Cited on page 91.)
- [50] DEREZINSKA, A. AND SZUSTEK, A. 2008. Tool-Supported Advanced Mutation Approach for Verification of C# Programs. In *Proceedings of the 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*. DEPCOS-RELCOMEX '08. IEEE Computer Society, Washington, DC, USA, 261–268. (Cited on page 50.)
- [51] DING, W. 2000. Master Thesis, George Mason University, Fairfax, VA. (Cited on page 50.)
- [52] DO, H. AND ROTHERMEL, G. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Softw. Eng.* 32, 9 (Sept.), 733–752. (Cited on page 50.)
- [53] DURAN, J. W. AND NTAFOSS, S. C. 1984. An evaluation of random testing. *Software Engineering, IEEE Transactions on SE-10*, 4 (july), 438–444. (Cited on page 30.)
- [54] ECLEMMMA. 2006. Download Site. <http://www.eclemma.org/>. (Cited on pages 57 and 58.)
- [55] ELBAUM, S., ROTHERMEL, G., KARRE, S., , AND II, M. F. 2005. Leveraging user-session data to support web application testing. *Journal of IEEE Transactions on Software Engineering* 31, 3, 187–202. (Cited on page 110.)

- [56] ESSER, M. AND STRUSS, P. 2007. Fault-model-based test generation for embedded software. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*. Hyderabad, India, 342–347. (Cited on page 72.)
- [57] FABBRI, S., MALDONADO, J., SUGETA, T., AND MASIERO, P. 1999. Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*. 210–219. (Cited on page 34.)
- [58] FABBRI, S. C. P. F., MALDONADO, J. C., MASIERO, P. C., AND DELAMARO, M. E. 1999. Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC'99)*. Talca, Chile, 96. (Cited on page 47.)
- [59] FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND STUMPTNER, M. 2000a. Consistency based diagnosis of configuration knowledge bases. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Berlin. (Cited on page 70.)
- [60] FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND STUMPTNER, M. 2000b. An integrated development environment for the design and maintenance of large configuration knowledge bases. In *Proceedings Artificial Intelligence in Design*. Kluwer Academic Publishers, Worcester MA. (Cited on page 70.)
- [61] FERRARI, F. C., NAKAGAWA, E. Y., MALDONADO, J. C., AND RASHID, A. 2011. Proteum/AJ: a mutation system for AspectJ programs. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development Companion*. AOSD '11. ACM, New York, NY, USA, 73–74. (Cited on page 47.)
- [62] FISHER, M., CAO, M., ROTHERMEL, G., COOK, C., AND BURNETT, M. 2002. Automated test case generation for spread-sheets. *Proceedings of the 24rd International Conference on Software Engineering*. (Cited on page 109.)
- [63] FRIEDRICH, G., STUMPTNER, M., AND WOTAWA, F. 1999. Model-based diagnosis of hardware designs. *Artificial Intelligence 111*, 2 (July), 3–39. (Cited on page 71.)
- [64] GENT, I. P., JEFFERSON, C., AND MIGUEL, I. 2006. MINION: A Fast, Scalable, Constraint Solver. *17th European Conference on Artificial Intelligence ECAI-06*, 98–102. (Cited on pages 78, 92, and 96.)
- [65] GHOSH, S. 2000. phdthesis. Ph.D. thesis, Purdue University, West Lafayette, Indiana. (Cited on page 47.)

- [66] GLIGORIC, M., JAGANNATH, V., AND MARINOV, D. 2010. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code. *International Conference on Software Testing, Verification, and Validation, 2008 0*, 55–64. (Cited on page 48.)
- [67] GOTLIEB, A. 2009. Euclide: A constraint-based testing framework for critical c programs. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. 151–160. (Cited on page 91.)
- [68] GOTLIEB, A., BOTELLA, B., AND RUEHER, M. 1998. Automatic Test Data Generation using Constraint Solving Techniques. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software testing and analysis*. Clearwater Beach, Florida, United States, 53–62. (Cited on pages 72 and 91.)
- [69] GRÜN, B. J. M., SCHULER, D., AND ZELLER, A. 2009. The Impact of Equivalent Mutants. In *IEEE International Conference on Software Testing, Verification, and Validation Workshops*. Denver, USA, 192–199. (Cited on pages 48 and 91.)
- [70] GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. 2005. Locating faulty code using failure-inducing chops. In *Automated Software Engineering (ASE)*. 263–272. (Cited on page 72.)
- [71] HIERONS, R. M., HARMAN, M., AND DANICIC, S. 1999. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability* 9, 4 (December), 233–262. (Cited on pages 83 and 90.)
- [72] HOWARD, S. 2002. Computational vulnerability analysis for information survivability. In *Eighteenth national conference on Artificial intelligence*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 919–926. (Cited on page 34.)
- [73] HOWDEN, W. E. 1982. Weak Mutation Testing and Completeness of Test Sets. *IEEE Trans. Softw. Eng.* 8, 4 (July), 371–379. (Cited on page 45.)
- [74] HUSSAIN, S. 2008. phdthesis. Ph.D. thesis, King’s College London, UK. (Cited on page 43.)
- [75] IBM. 2001. The Eclipse Project. <http://www.eclipse.org/>. (Cited on pages 57 and 89.)
- [76] II, M. F., JIN, D., ROTHERMEL, G., AND BURNETT, M. 2002. Test reuse in the spreadsheet paradigm. *Proceedings of the 13th International Symposium on Software Reliability Engineering - ISSRE 2003*. (Cited on page 109.)
- [77] IMB. 2002. Rational Functional Tester. <http://www-01.ibm.com/software/awdtools/tester/functional/>. (Cited on page 111.)

- [78] I. MOORE. May 2001. Jester - a JUnit Test Tester. In *Proceedings of the 2nd. International Conference on Extreme Programming and Flexible Processes in Software Engineering*. 84–87. (Cited on page 47.)
- [79] IRVINE, S., PAVLINIC, T., TRIGG, L., CLEARY, J., INGLIS, S., AND UTTING, M. 2007. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. 169–175. (Cited on pages 34, 48, 58, and 96.)
- [80] ITREGISTER. 2007. Plextest. <http://www.itregister.com.au/products/plextest.htm>. (Cited on page 47.)
- [81] JAVALANCHE. 2011. Download Site. <https://github.com/david-schuler/javalanche/>. (Cited on page 57.)
- [82] JDT. JDT core plugin. <http://www.eclipse.org/jdt/>. (Cited on page 60.)
- [83] JI, C., CHEN, Z., XU, B., AND ZHAO, Z. 2009. A Novel Method of Mutation Clustering Based on Domain Analysis. In *SEKE (2009-06-29)*. Knowledge Systems Institute Graduate School, 422–425. (Cited on page 43.)
- [84] JIA, Y. AND HARMAN, M. Mutation testing repository. [http://crestweb.cs.ucl.ac.uk/resources/mutation\\_testing\\_repository/index.php](http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/index.php). (Cited on page 39.)
- [85] JIA, Y. AND HARMAN, M. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 249–258. (Cited on page 44.)
- [86] JIA, Y. AND HARMAN, M. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering* 99, 1. (Cited on page 12.)
- [87] JONES, J. A. AND HARROLD, M. J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings ASE'05*. ACM Press, 273–282. (Cited on pages 71 and 72.)
- [88] KANER, C., BACH, J., AND PETTICHORD, B. 2002. *Lessons Learned in Software Testing*. John Wiley & Sons. (Cited on page 33.)
- [89] KICILLOF, N., GRIESKAMP, W., TILLMANN, N., AND BRABERMAN, V. 2007. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd international*

- workshop on Advances in model-based testing*. A-MOST '07. ACM, New York, NY, USA, 1–11. (Cited on page 33.)
- [90] KING, K. N. AND OFFUTT, A. J. 1991. A fortran language system for mutation-based software testing. *Softw. Pract. Exper.* 21, 7 (June), 685–718. (Cited on pages 41 and 47.)
- [91] KO, A. J., ABRAHAM, R., BECKWITH, L., BLACKWELL, A., BURNETT, M., ERWIG, M., SCAFFIDI, C., LAWRENCE, J., LIEBERMAN, H., MYERS, B., ROSSON, M. B., ROTHERMEL, G., SHAW, M., AND WIEDENBECK, S. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys* 43, 3, 21:1–21:44. (Cited on pages 106 and 109.)
- [92] KOREL, B. AND LASKI, J. 1988. Dynamic Program Slicing. *Information Processing Letters* 29, 155–163. (Cited on page 73.)
- [93] KOREL, B. AND RILLING, J. 1997. Applications of Dynamic Slicing in Program Debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG-97)*. Linköping, Sweden, 43–58. (Cited on page 73.)
- [94] KRENN, W. AND AICHERNIG, B. 2009. Test Case Generation by Contract Mutation in Spec#. In *Proceedings of Fifth Workshop on Model Based Testing (MBT'09)*. York, UK, 71–86. (Cited on pages 34 and 50.)
- [95] KULESZA, T., BURNETT, M., STUMPF, S., WONG, W., DAS, S., GROCE, A., SHINSEL, A., BICE, F., AND MCINTOSH, K. 2011. Where are my intelligent assistant's mistakes? a systematic testing approach. *Proceedings of End-User Development - Third International Symposium, IS-EUD 2011*. (Cited on page 110.)
- [96] KULESZA, T., STUMPF, S., WONG, W., BURNETT, M., PERONA, S., KO, A., AND OBERST, I. 2011. Why-oriented end-user debugging of naive bayes text classification. *Journal of ACM Transactions on Interactive Intelligent Systems*. (Cited on page 110.)
- [97] KUSUMOTO, S., NISHIMATSU, A., NISHIE, K., AND INOUE, K. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7, 49–76. (Cited on page 72.)
- [98] LANGDON, W., HARMAN, M., AND JIA, Y. 2009. Multi Objective Higher Order Mutation Testing with Genetic Programming. In *Testing: Academic and Industrial Conference - Practice and Research Techniques, TAIC PART '09*. 21–29. (Cited on page 44.)

- [99] LAWRENCE, J., CLARKE, S., BURNETT, M., AND ROTHERMEL, G. 2005. How well do professional developers test with code coverage visualizations? an empirical study. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*. (Cited on page 106.)
- [100] LEVER, S. 2005. Eclipse Platform Integration of Jester &#8211; the JUnit Test Tester. In *Proceedings of the 6th international conference on Extreme Programming and Agile Processes in Software Engineering*. XP'05. Springer-Verlag, Berlin, Heidelberg, 325–326. (Cited on page 48.)
- [101] LIEBERMAN, H. AND WAGNER, E. 2003. End-user debugging for electronic commerce. *Proceedings of the 8th international conference on Intelligent user interfaces*. (Cited on page 111.)
- [102] LIU, H., KUO, F., AND CHEN, T. Y. 2010. Teaching an end-user testing methodology. *Proceeding of the 23rd IEEE Conference on Software Engineering Education and Training*. (Cited on page 110.)
- [103] LIVER, B. 1994. Modeling software systems for diagnosis. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis*. New Paltz, NY, 179–184. (Cited on page 70.)
- [104] MA, Y., OFFUTT, A. J., AND KWON, Y. MuJava: a Mutation System for Java. In *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, year =. (Cited on page 58.)
- [105] MA, Y., OFFUTT, J., AND KWON, Y. 2005. Mujava : An automated class mutation system. *Software Testing, Verification and Reliability* 15, 97–133. (Cited on pages 45, 48, 52, 57, and 82.)
- [106] MA, Y.-S., HARROLD, M. J., AND KWON, Y.-R. 2006. Evaluation of Mutation Testing for Object-Oriented Programs. In *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*. Shanghai, China, 869–872. (Cited on page 41.)
- [107] MA, Y.-S. AND OFFUTT, J. 2005. Description of Method-level Mutation Operators for Java. (Cited on page 52.)
- [108] MADEYSKI, L. 2011. Judy - Java Mutation Tester. (Cited on page 49.)
- [109] MADEYSKI, L. AND RADYK, N. Feb.2011. Judy - A Mutation Testing Tool For Java. *Software, IET* 4, 1, 32–42. (Cited on pages 49 and 64.)
- [110] MALDONADO, J. C., DELAMARO, M. E., FABBRI, S. C. P. F., ADENILSO DA SILVA SIM A., SUGETA, T., VINCENZI, A. M. R., AND MASIERO, P. C. 2001. Proteum: A Family of Tools to Support Specification and Program Testing Based On Mutation. In *Proceedings of the*

- 1st Workshop on Mutation Analysis (MUTATION'00)*. San Jose, California, 113–117. (Cited on page 47.)
- [111] MALDONADO, J. C., DELAMARO, M. E., FABBRI, S. C. P. F., DA SILVA SIMO, A., SUGETA, T., VINCENZI, A. M. R., AND MASIERO, P. C. Proteum Family Tools. <http://www.labes.icmc.usp.br/proteum/>. (Cited on page 47.)
- [112] MATEO, P. AND USAOLA, M. Sept. 2012. Bacterio: Java mutation testing tool: A framework to Evaluate Quality of Tests Cases. In *28th IEEE International Conference on Software Maintenance (ICSM), 2012*. 646–649. (Cited on page 49.)
- [113] MATHUR, A. P. AND KRAUSER, E. W. 1988. Mutant Unification for Improved Vectorization. techreport SERC-TR-14-P, Purdue University, West Lafayette, Indiana. (Cited on page 45.)
- [114] MAYER, W. 2007. Static and hybrid analysis in model-based debugging. *PhD Thesis, School of Computer and Information Science University of South Australia*. (Cited on pages 69, 71, 84, and 88.)
- [115] MAYER, W., ABREU, R., STUMPTNER, M., AND VAN GEMUND, A. J. 2009. Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis (DX)*. (Cited on page 69.)
- [116] MAYER, W. AND STUMPTNER, M. 2003. Model-based debugging using multiple abstract models. *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging AADEBUG-03*, 55–70. (Cited on page 71.)
- [117] MAYER, W., STUMPTNER, M., WIELAND, D., AND WOTAWA, F. 2002a. Can AI Help to Improve Debugging Substantially? Debugging Experiences With Value-Based Models. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. IOS Press, Lyon, France, 417–421. (Cited on page 71.)
- [118] MAYER, W., STUMPTNER, M., WIELAND, D., AND WOTAWA, F. 2002b. Towards an Integrated Debugging Environment. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. IOS Press, Lyon, France, 422–426. (Cited on page 71.)
- [119] MCCONNELL, S. Daily build and smoke test. <http://www.stevemccconnell.com/ieeesoftware/bp04.htm>. (Cited on page 30.)
- [120] MCILRAITH, S. 1993. Generating tests using abduction. In *Proc. DX'93 Workshop*. (Cited on page 72.)

- [121] MEDINA, R. AND PRATMARTY, P. Uispec4j. <http://www.uispec4j.org/>. (Cited on page 49.)
- [122] MONSON-HAEFEL, R. 2001. *Enterprise JavaBeans*, 3rd ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA. (Cited on page 52.)
- [123] MOORE, I. 2000. Jester Website. <http://jester.sourceforge.net/>. (Cited on page 47.)
- [124] MRESA, E. S. AND BOTTACI, L. 1999. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability* 9, 4 (December), 205–232. (Cited on page 43.)
- [125] MYERS, G. J. 1979. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA. (Cited on pages iii, v, 11, 19, 21, 22, 30, and 32.)
- [126] NAMIN, A. S., 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 (ICSE'08)*. Leipzig, Germany, 351–360. (Cited on page 43.)
- [127] NICA, M., NICA, S., AND WOTAWA, F. 2010a. Does Testing Help to Reduce the Number of Potentially Faulty Statements in Debugging? In *Testing: Academic & Industrial Conference Practice and Research Techniques (TAIC PART)*. Springer LNCS. (Cited on pages 12, 19, 72, 73, 89, 93, and 103.)
- [128] NICA, M., NICA, S., AND WOTAWA, F. 2010b. Using Distinguishing Tests to Reduce the Number of Fault Candidates. *Proceedings of the 21st International Workshop on the Principles of Diagnosis (DX-10)*. (Cited on page 12.)
- [129] NICA, M., NICA, S., AND WOTAWA, F. 2012. On the Use of Mutations and Testing for Debugging. *Software: Practice and Experience*. (Cited on pages 12, 51, and 65.)
- [130] NICA, M., WEBER, J., AND WOTAWA, F. 2008. How to Debug Sequential Code by Means of Constraint Representation. In *International Workshop on Principles of Diagnosis (DX-08)*. Leura, Australia. (Cited on pages 52, 69, and 71.)
- [131] NICA, S. 2011. On the Improvement of the Mutation Score Using Distinguishing Test Cases. *Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011) PhD Symposium*. (Cited on page 12.)
- [132] NICA, S., NICA, M., AND WOTAWA, F. 2010c. Improving the Mutation Score by Means of Distinguishing Test Cases. *Nordic Workshop on Programming Theory (NWPT'10)*. (Cited on page 12.)



- [133] NICA, S., NICA, M., AND WOTAWA, F. 2011. Detecting Equivalent Mutants by Means of Constraint Systems. *Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*. (Cited on page 12.)
- [134] NICA, S. AND PEISCHL, B. 2009. Challenges in Applying Mutation Analysis on EJB-based Business Applications. In *Proceedings of Metrikon 2009*. Kaiserslautern, Germany. (Cited on pages 12, 51, 52, and 96.)
- [135] NICA, S., RAMLER, R., AND WOTAWA, F. 2011. Is Mutation Testing Scalable for RealWorld Software Projects? *Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*. (Cited on pages 12 and 51.)
- [136] NICA, S. AND WOTAWA, F. 2012a. EqMutDetect-A Tool for Equivalent Mutant Detection in Embedded Systems. *Proceedings of the Tenth Workshop on Intelligent Solutions in Embedded Systems (WISES 2012)*. (Cited on pages 12 and 51.)
- [137] NICA, S. AND WOTAWA, F. 2012b. Using Constraints for Equivalent Mutant Detection. *Proceedings of the 2nd Workshop on Formal Methods in the Development of Software (FMDS 2012)*. (Cited on pages 13 and 51.)
- [138] OFFUTT, A. J. 1989. The coupling effect: Fact or fiction. *ACM SIGSOFT Software Engineering Notes* 14, 8 (December), 131–140. (Cited on page 39.)
- [139] OFFUTT, A. J. 1992. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology* 1, 1 (January), 5–20. (Cited on page 39.)
- [140] OFFUTT, A. J., LEE, A., ROTHERMEL, G., UNTCH, R., AND ZAPF, C. 1996. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology* 5, 99–118. (Cited on pages 43, 71, and 83.)
- [141] OFFUTT, A. J. AND LEE, S. D. 1991. How Strong Is Weak Mutation? In *Proceedings of the Symposium on Testing, Analysis, and Verification*. TAV4. ACM, New York, NY, USA, 200–213. (Cited on page 45.)
- [142] OFFUTT, A. J. AND PAN, J. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability* 7, 3 (September), 165–192. (Cited on pages 71, 83, and 90.)
- [143] OFFUTT, A. J., PAN, J., AND VOAS, J. M. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the 12 International Conference on Testing Computer Software*, 111123. (Cited on page 50.)

- [144] OFFUTT, A. J. AND UNTCH, R. H. 2001. Mutation testing for the new century. Kluwer Academic Publishers, Norwell, MA, USA, Chapter Mutation 2000: uniting the orthogonal, 34–44. (Cited on page 42.)
- [145] OFFUTT, J., MA, Y.-S., AND KWON, Y.-R. 2004. An Experimental Mutation System for Java. *SIGSOFT Softw. Eng. Notes* 29, 5 (Sept.), 1–4. (Cited on pages 45 and 48.)
- [146] OFFUTT, J. A. AND LEE, S. D. 1994. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering* 20, 5, 337–344. (Cited on page 71.)
- [147] PAN, J. 1994. Using Constraints to Detect Equivalent Mutants. M.S. thesis, George Mason University, Fairfax, VA. (Cited on page 90.)
- [148] PANKO, R. R. 2008. Spreadsheet errors: What we know. what we think we can do. *Proceedings of The European Spreadsheet Risks Interest Group*. (Cited on page 106.)
- [149] PAPADAKIS, M. AND MALEVRIS, N. 2010. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*. 90–99. (Cited on page 44.)
- [150] PARASOFT. 2006. Parasoft Insure++. <http://www.parasoft.com/jsp/products/home.jsp?product=Insure>. (Cited on page 47.)
- [151] POLI, R., LANGDON, W. B., , AND MCPHEE, N. F. 2008. A Field Guide to Genetic Programming. (Cited on page 44.)
- [152] POLO, M., PIATTINI, M., AND GARCÍA-RODRÍGUEZ, I. 2009. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Softw. Test. Verif. Reliab.* 19, 2 (June), 111–131. (Cited on page 44.)
- [153] POLO USAOLA, M., REALES MATEO, P., AND PÉREZ LAMANCHA, B. 2012. Reduction of Test Suites Using Mutation. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*. FASE'12. Springer-Verlag, Berlin, Heidelberg, 425–438. (Cited on page 50.)
- [154] PREMRAJ, R. AND ZELLER, A. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. PROMISE '07. 9. (Cited on page 65.)

- [155] PURUSHOTHAMAN, R. AND PERRY, D. June 2005. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering* 31, 6, 511–526. (Cited on page 44.)
- [156] REITER, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1, 57–95. (Cited on page 70.)
- [157] RESEARCH, M. 2007. Pex - automated white box testing for .net. <http://research.microsoft.com/Pex>. (Cited on page 33.)
- [158] ROTHERMEL, G. AND HARROLD, M. 1990. Empirical Studies of A Safe Regression Test Selection Technique. *IEEE Transactions on Software Engineering* 24, 6, 401–419. (Cited on page 100.)
- [159] ROTHERMEL, G., LI, L., AND BURNETT, M. 1998. What you see is what you test: a methodology for testing form-based visual programs. *Proceedings of the 1998 International Conference on Software Engineering*. (Cited on pages 32 and 108.)
- [160] ROTHERMEL, G., LI, L., AND BURNETT, M. 2005. Testing strategies for form-based visual programs. *Proceedings of the Eighth International Symposium on Software Reliability Engineering*. (Cited on page 106.)
- [161] ROTHERMEL, K., COOK, C., BURNETT, M., SCHONFELD, J., GREEN, T., AND ROTHERMEL, G. 2000. WYSIWYT: Testing in the spreadsheet paradigm: an empirical evaluation. *Proceedings of the 2000 International Conference on Software Engineering*. (Cited on page 108.)
- [162] RUBYFORGE. 2007. Heckle. <http://seattlerb.rubyforge.org/heckle/>. (Cited on page 50.)
- [163] SAHINOGLU, M. AND SPAFFORD, E. H. 1990. A Bayes Sequential Statistical Procedure for Approving Software Products. *Proceedings of the IFIP Conference on Approving Software Products (ASP 90)* 9, 43–56. (Cited on page 43.)
- [164] SCAFFIDI, C., MYERS, B., AND SHAW, M. 2008. Tool support for data validation by end-user programmers. *Proceedings of ACM/IEEE 30th International Conference on Software Engineering - ICSE '08*. (Cited on page 110.)
- [165] SCAFFIDI, C., SHAW, M., AND MYERS, B. 2005. Estimating the numbers of end users and end user programmers. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'05)*, 207–214. (Cited on page 106.)

- [166] SCHULER, D. AND ZELLER, A. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*. Amsterdam, Netherlands, 297–298. (Cited on pages 48, 58, and 96.)
- [167] SCHULER, D. AND ZELLER, A. 2010. (Un-)Covering Equivalent Mutants. In *ICST '10: Third International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 45–54. (Cited on pages 48, 59, 83, and 90.)
- [168] SHAPIRO, E. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts. (Cited on page 70.)
- [169] SHIRLEY, M. H. AND DAVIS, R. 1983. Generating distinguishing tests based on hierarchical models and symptom information. In *IEEE International Conference on Computer Design*. (Cited on page 72.)
- [170] SMITH, B. H. AND WILLIAMS, L. 2007. An Empirical Evaluation of the MuJava Mutation Operators. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*. Windsor, UK, 193–202. (Cited on page 48.)
- [171] SPILLNER, A., LINZ, T., AND SCHAEFER, H. 2007. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook Inc. (Cited on pages 19, 25, 28, and 30.)
- [172] SYNOPSIS. 2006. Certitude Functional Qualification System. <http://www.springsoft.com/products/functionalqualification/certitude>. (Cited on page 47.)
- [173] TELERIK. 2002. Telerik Test Studio. <http://www.telerik.com/automated-testing-tools/>. (Cited on page 111.)
- [174] THOUGHTWORKS. 2004. Selenium. <http://docs.seleniumhq.org/>. (Cited on page 111.)
- [175] TRICENTIS. 2010. Tosca TestSuite. <http://www.tricentis.com/en/tosca>. (Cited on page 111.)
- [176] TUYA, J., SUÁREZ-CABAL, M. J., AND LA RIVA, C. D. 2007. Mutating Database Queries. *Inf. Softw. Technol.* 49, 4 (Apr.), 398–417. (Cited on page 49.)
- [177] TWO, R. AND OF WAIKATO, U. 2007. Jumble. <http://jumble.sourceforge.net/index.html>. (Cited on pages 48 and 57.)
- [178] UNTCH, R. H. 1992. Mutation-Based Software Testing Using Program Schemata. In *Proceedings of the 30th Annual Southeast Regional Conference*. ACM-SE 30. ACM, New York, NY, USA, 285–291. (Cited on page 44.)

- [179] UNTCH, R. H. 1995. Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method. Ph.D. thesis, Clemson, SC, USA. AAI9703410. (Cited on page 44.)
- [180] UNTCH, R. H., OFFUTT, A. J., AND HARROLD, M. J. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '93. ACM, New York, NY, USA, 139–148. (Cited on page 44.)
- [181] USAOLA, M. P. 2007. *Testooj User'S Manual*. Alarcos Research Group, Department of Information Systems and Technologies, University of Castilla-La Mancha. DRAFT Version: October 25, 2007. (Cited on page 49.)
- [182] VOAS, J. Building software recovery assertions from a fault injection-based propagation analysis. In *IN PROC. OF COMPSAC'97, PAGES 505510, WASHINGTON D.C, year = 1997, pages = 505510, publisher = IEEE Computer Society*. (Cited on page 34.)
- [183] VOAS, J. 1992. Pie: a dynamic failure-based technique. *Software Engineering, IEEE Transactions on* 18, 8 (aug), 717–727. (Cited on page 34.)
- [184] VOAS, J., CHARRON, F., MILLER, K., AND FRIEDMAN, M. 1997. Predicting how badly "good" software can behave. *IEEE Software* 14, 14–4. (Cited on page 34.)
- [185] VOAS, J. AND MCGRAW, G. 1999. Software fault injection: inoculating programs against errors. *Software Testing, Verification and Reliability* 9, 1, 75–76. (Cited on page 34.)
- [186] WAGNER, E. AND LIEBERMAN, H. 2004. Supporting user hypotheses in problem diagnosis on the web and elsewhere. *Proceedings of the International Conference on Intelligent User Interfaces*. (Cited on page 111.)
- [187] WAGNER, E. AND LIEBERMAN, H. <http://web.media.mit.edu/~lieber/Lieberary/End-User-Debugging/End-User-Debugging-Intro.html>. Woodstein. 2003. (Cited on page 111.)
- [188] WAGNER, E. J. 2003. Woodstein: A web interface agent for debugging e-commerce. M.S. thesis, MIT Media Laboratory. (Cited on page 111.)
- [189] WEIMER, W., NGUYEN, T. V., GOUES, C. L., AND FORREST, S. 2009. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 512–521. (Cited on pages 67, 69, and 71.)
- [190] WEISER, M. 1982. Programmers use slices when debugging. *Communications of the ACM* 25, 7 (July), 446–452. (Cited on page 72.)


- [191] WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10, 4 (July), 352–357. (Cited on page 72.)
- [192] WEISS, S. N. AND FLEYSHGAKKER, V. N. 1993. Improved Serial Algorithms for Mutation Analysis. *ACM SIGSOFT Software Engineering Notes* 18, 3 (July), 149–158. (Cited on page 45.)
- [193] WHITTAKER, J. A. 2002. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 37.)
- [194] WONG, W. E. 1993. On Mutation and Data Flow. (Cited on page 43.)
- [195] WONG, W. E., MALDONADO, J. C., DELAMARO, M. E., AND MATHUR, A. P. 1994. Constrained mutation in C programs. In *VIII Simposio Brasileiro de Engenharia de Software (SBES 94)*. Curitiba, PR, Brazil, 439–452. (Cited on page 43.)
- [196] WOODWARD, M. AND HALEWOOD, K. 1988. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*. 152–158. (Cited on page 46.)
- [197] WOTAWA, F. 2002a. Debugging Hardware Designs using a Value-Based Model. *Applied Intelligence* 16, 1, 71–92. (Cited on page 71.)
- [198] WOTAWA, F. 2002b. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence* 135, 1–2, 124–143. (Cited on page 73.)
- [199] WOTAWA, F. 2008. Bridging the gap between slicing and model-based diagnosis. In *Proc. of the 20th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE)*. 836–841. (Cited on page 72.)
- [200] WOTAWA, F. AND NICA, M. 2008. On the compilation of programs into their equivalent constraint representation. *Informatika* 32, 359–371. (Cited on pages 71 and 74.)
- [201] WOTAWA, F., NICA, M., AND AICHERNIG, B. K. 2010. Generating Distinguishing Tests using the MINION Constraint Solver. In *CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis*. Paris, France, 325–330. (Cited on pages 52, 67, 82, 91, and 103.)
- [202] WOTAWA, F., NICA, S., AND NICA, M. 2011. Constraint-Based Debugging Combining Mutations and Distinguishing Test Cases. *Proceedings of the Ninth Workshop on Intelligent Solutions in Embedded Systems (WISES 2011)*. (Cited on page 12.)


- [203] WOTAWA, F. AND PEISCHL, B. 2006. Automated source level error localization in hardware designs. *IEEE Design and Test of Computers* 23(1), 8–19. (Cited on pages 69 and 71.)
- [204] WOTAWA, F., WEBER, J., NICA, M., AND CEBALLOS, R. 2010. On the Complexity of Program Debugging Using Constraints for Modeling the Program’s Syntax and Semantics. In *Current Topics in Artificial Intelligence*, P. Meseguer, L. Mandow, and R. Gasca, Eds. Lecture Notes in Computer Science, vol. 5988. Springer Berlin / Heidelberg, 22–31. (Cited on pages 52 and 71.)
- [205] Y.S.MA, OFFUTT, J., AND KWON, Y. R. MuJava : An Automated Class Mutation System. *Software Testing, Verification and Reliability*, volume =. (Cited on pages 58 and 96.)
- [206] ZAPF, C. N. August 1993. Medusamothra - A Distributed Interpreter for the Mothra Mutation Testing System. M.S. thesis, Clemson, SC, USA. (Cited on page 45.)
- [207] ZELLER, A. AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (feb). (Cited on page 72.)
- [208] ZHANG, X., HE, H., GUPTA, N., AND GUPTA, R. 2005. Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*. 33–42. (Cited on pages 72 and 73.)
- [209] ZHOU, C. AND FRANKL, P. 2009. Mutation Testing for Java Database Applications. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation. ICST ’09*. IEEE Computer Society, Washington, DC, USA, 396–405. (Cited on page 49.)

## PERSONAL INFORMATION

Simona Alina Nica



 Steyrergasse 156 a / 9, 8010 Graz (Austria)

 [simona.nica@ist.tugraz.at](mailto:simona.nica@ist.tugraz.at)  
[simona.nica@gmx.at](mailto:simona.nica@gmx.at)

Date of birth 20 December 1981 | Nationality Romanian Religion Romanian Orthodox Christian

## WORK EXPERIENCE

January 2009 – February 2013

## PhD Research Assistant

Graz University of Technology  
<http://www.tugraz.at/>

Research conducted in software testing, automated test case generation, mutation testing  
 Test support for End User Programming (End User Testing), mutation testing on industry size projects (project involved the Eclipse IDE)

Combined testing with debugging in order to reduce the number of diagnosis fault candidates  
 Research in test management, together with Siemens Austria AG, PSE – EJB mutation testing

*Journal Papers*

1. Mihai Nica, Simona Nica and Franz Wotawa, *On the use of mutations and testing for debugging*, in Journal of Software: Practice and Experience, 2012

*Conference Papers*

2. Simona Nica and Franz Wotawa, *Using Constraints for Equivalent Mutant Detection*, in Proceedings of WS-FMDS 2012, August 2012, Paris, France

3. Simona Nica and Franz Wotawa, *EqMutDetect - A Tool for Equivalent Mutant Detection in Embedded Systems*, in Proceedings of the Ninth Workshop on Intelligent Solutions in Embedded Systems (WISES 2012), July 2012, Klagenfurt, Austria

4. Simona Nica, Rudolf Ramler, and Franz Wotawa, *Is Mutation Testing Scalable for Real-World Software Projects?*, in Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011), November 2011, Barcelona, Spain

5. Franz Wotawa, Simona Nica and Mihai Nica, *Constraint-Based Debugging Combining Mutations and Distinguishing Test Cases*, in Proceedings of the Ninth Workshop on Intelligent Solutions in Embedded Systems (WISES 2011), July 2011, Regensburg, Germany

6. Simona Nica, *On the Improvement of the Mutation Score Using Distinguishing Test Cases*. Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011) - PhD Symposium, March 2011, Berlin, Germany

7. Mihai Nica, Simona Nica, and Franz Wotawa, *Does Testing Help to Reduce the Number of Potentially Faulty Statements in Debugging?* In Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART 2010), September 2010, Cumberland Lodge, Windsor, UK

8. Mihai Nica, Simona Nica, and Franz Wotawa, *Using Distinguishing Tests to Reduce the Number of Fault Candidates*, in Proceedings of the 21st International Workshop on the Principles of Diagnosis (DX-10), October 2010, Portland, USA

9. Simona Nica and Bernhard Peischl, *Challenges in Applying Mutation Analysis on EJB-Based Business Applications*, in Proceedings of Metrikon 2009, November 2009, Kaiserslautern, Germany

*Business or sector Research*

August 2006 – January 2009

## Software Engineer

CS ROMANIA , <http://www.c-s.ro/>, Craiova (Rumänien)

CS CANADA ( <http://www.c-s-canada.ca> ) , Montreal (Canada) - April 2007 – June 2007

1. IADSW (Integrated Avionics Display Software) Thales S76-D Project – with Thales, CS France: unitary tests for Platform Display Unit Software, in accordance with DO178



- B Standard, level A; review of the SRS document Software
  - 2. Integration Tests for PW545C Electronic Engine Control Software; Design of SDD and SRD documents for PW545C software
  - 3. M50 Toll Collection System - Back Office Subsystem, with CS-InTranS Group: Interface Testing
  - 4. Pratt & Whitney Canada, PW545C Electronic Engine Control Software: Software Integration Tests
  - 5. STAEGC Air Traffic Control System: robustness testing
  - 6. CONVERA Corporation: software development for an Ontology Editor, Excalibur project
  - 7. FAIVELEY TRANSPORT, CS France: unitary tests
- Business or sector** *Energy / Industry, Transport, Avionics and Defence, Space & Security*

October 2004 – June 2006 **Software Developer**  
*Syncro Soft SA*  
[http://www.oxygenxml.com/about\\_us.html](http://www.oxygenxml.com/about_us.html)

Oxygen Tool Development :  
 Software development  
 Unit testing  
 Automated testing  
 Quality Assurance

**Business or sector** *XML Technologies*

EDUCATION AND TRAINING

January 2009 - Present **Dr.techn. - PhD in Doctoral School Of Informatics (Dissertation Defence planned for April 2013; Thesis Title: "On the Use of Constraints in Program Mutation and Its Applicability to Testing")** ICSDE 6  
 Graz University of Technology,

Software Test Design  
 Mutation Testing  
 Automated Test Case Generation  
 Constraint Satisfaction Problem  
 Testing and Debugging  
 Software Development

October 2001 – June 2006 **Dipl. Ing.** ICSDE 5  
 University of Craiova, Faculty for Automation, Computers and Electronics (Software Engineering Department – English Class)

Object Oriented Programming, Algorithms and Languages, Compiler Design, Artificial Intelligence, Programming in Assembly Languages, Algorithms Complexity Analysis, Data Communications, Visual Programming Environments, Information Systems Management, Computer Networks, Modelling and Simulation for Computer Systems and Networks, IT Projects Management, Operating Systems Design, Databases Design, Real Time Computing Systems, Formal Languages, Expert Systems, Computer Networks Management, Embedded Systems, Graphical Processing Systems, Logical Design of Digital Computers

PERSONAL SKILLS

Mother tongue(s) Romanian

Other language(s)	UNDERSTANDING		SPEAKING		WRITING
	Listening	Reading	Spoken interaction	Spoken production	
English	C1	C1	C1	C1	C1
German	B2	B2	B2	B2	B2
French	B2	B2	B2	B2	B2
Italian	B2	B1	B2	B2	B1

Levels: A1/A2: Basic user - B1/B2: Independent user - C1/C2: Proficient user

**Social skills and competences** Conference and workshops participation and talks helped me develop my intercultural and communication skills  
Competitive team work together with my colleagues  
Seminar lectures in mutation testing

**Organisational / managerial skills** I was involved in the organization of the following events:  
RoboCup 2009 Championship  
TAROT 2009 Summer School  
The 6th Softnet Conference

**Job-related skills** *Software Entwicklung:*  
- Programmiersprache: Pascal, C, Java, C#  
- IDE: Eclipse, NetBeans, Microsoft Visual Studio 2010  
- Revision Control : SVN  
- Formale Sprachen: Prolog  
*Databases:*  
- Programmiersprache: SQL  
- Plattformen: MySQL, MS Access  
*Testing:*  
- JUnit (Java Unit Testing)  
- NUnit (C# Unit Testing)  
- Microsoft Pex - White Box Testing für .NET  
- ADA / C Unit Testing mit Rational Test Real Time  
- Testing Standards: DO-178 Niveau A und B  
Constraint Programming  
Code Assembly  
Technologien: Apache Ant, NAant, XDoclet, J2EE, XML, .NET, JBoss  
EDV Kenntnisse: Microsoft Office (Word, Excel, PowerPoint, Access)

**Hobbies** Photography  
Classic Piano  
Classic Guitar  
Tennis  
Ski

**Driving licence** B (since August 2000)