FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Regression Testing with GZoltar: Techniques for Test Suite Minimization, Selection, and Prioritization

José Campos

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Abreu (PhD)

14^{th} February, 2012

Regression Testing with GZoltar: Techniques for Test Suite Minimization, Selection, and Prioritization

José Campos

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: João Faria (PhD)

External Examiner: Marco Vieira (PhD)

Supervisor: Rui Abreu (PhD)

14^{th} February, 2012

Abstract

Software testing occurs simultaneously during the software development to detect errors as early as possible and to guarantee that changes made in software did not affect the system negatively.

However, during the development phase, the test suite is updated and tends to increase in size. Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to reduce the effort of regression testing. Several approaches have been studied to reduce the effort of regression testing: test suite minimization, selection, and prioritization. Test suite minimization techniques aims at identifying and eliminating redundant test cases from the suite. Test suite selection techniques identifies a subset of test cases from suite, required to re-test the changes in the software. Test suite prioritization techniques schedule test cases for execution in an order to increase the early fault detection.

Although, the approaches present the literature have some limitations (resources, times, etc.). In this manuscript we propose a new approach to the minimization, selection and prioritization problem that try to overcomes these limitation.

Our approach allows for (1) easy encoding of the relation between a test case and a component (source code statement) in a coverage matrix, (2) mapping this matrix in a set of constraints, and (3) computing optimal solutions (with the same coverage and fault detection of original set of tests) to minimization and selection problems by leveraging a fast and scalable constraint solver. We implemented our approach in a Eclipse plug-in called GZOLTAR, which is perfectly integrated in the IDE and provides functions to debugging a software program using visualization in OpenGL. Features for regression tests were implemented with the name Regression-Zoltar, RZOLTAR for short.

To test our toolset, we conducted experiments to measure the efficiency and the size of regression suite. In particular, we evaluated our tool using real-world and large programs, as well as a user study in order to measure its usability. In our empirical evaluation, we show how RZOLTAR can be used to instantiate large-programs and efficiently find an solution for such problems. Our experiments show that our approach can significantly reduce the size of original test suite, and a corresponding reduction in the cost of regression testing. We also make a user study to find if we are in right way in terms of usability and functionalities.

Acknowledgements

This thesis would have not been possible without the help of some several people.

First of all, I would like to express my gratitude to my thesis supervisor Prof. Dr. Rui Maranhão. In September of 2009 I had the opportunity to meet not only a professor, but 'The Professor' who open my eyes, to the beautiful of Operating Systems. Between March and June of 2010 we work together, me as a teaching assistant, at Computer Laboratory and was a very good experience. :) With regards to this thesis, thank you for accepting my idea and to be my supervisor. It was a great journey and without your support would not be possible. Now that I look to the past, I think this whole journey was predestined, the proposal that I presented to you in 28^{th} of January, fits like a glove in the project that comes from you PhD and was continued by André Riboira. (and that time I did not know anything about or PhD or André works) There are people who call it destiny, but whatever has been, it was a pleasure working with you. In the course of time we have created a great friendship, and in addition to supervise and guide me, we are now more than just co-workers, we are friends above of all.

Thank to the people of my laboratory, Prof. João Pascoal Faria, Francisco Andrade, Hugo Sereno, Nuno Flores, Nuno Cardoso (thanks for the tip *StringBuffer*), and of course to André Riboira for every support and clarifications about GZoltar. Because this MSc would not be possible without my academic and professional background, I cannot forget to mention also my gratitude to Prof. José Almeida, Prof. Lopes, Prof. Eduarda, Prof. Luís Reis, Prof. Ana Rocha, Prof. João Cardoso, Prof. Ana Paiva, and all my teachers in Escola Secundária Dr. António Granjo, Prof. Vanderley, Prof. Adelaide, Prof. Ana and Prof. Luisa, from Escola Secundária Fernão de Magalhães, Prof. Jorge Teixeira and from Escola Secundária Dr. Júlio Martins, Prof. Jorge Pires, in Escolar Primária de Valpaços a special word to Prof. Jorge.

Many thanks to my group of academic projects: João Sousa, Luis Calado, Rodolfo Silva, João Batista, Manuel Magina, Luis Pedrosa, and Tomé Duarte. To all the volunteers who participated on usability tests, thank you for your time and comments. All your feedback was of great value to improve and evaluate my work. Concerning the non-academic side of my life in Porto, I have to thank a great number of people for their friendship and the many enjoyable moments we spent together. Special thanks to Isabel Ribeiro for help me with English writing.

I would also take this opportunity to thank all my friends. First of all, to "Os Mosqueteiros" a group of 4 musketeers (Sérgio André, Tiago Gomes, and Miguel Silva) that met for the first time several years ago in Valpaços. To my football team of mondays, we must continue playing football to see if I lose a few more "quilinhos".

A special word for all my friends, we know each other a long time ago, thank you for your support in every moments of my life, the good and the bad ones. I am very grateful to all of them. I will not enumerate them because they are so many and I could forget to mention some of them.

In this jouney I am indebted to my family and my girlfriend. (now in Portuguese) Quero enviar um especial agradecimento para o meus pais, José e Paula. Obrigado por todo o apoio que me deram em toda a minha vida, sem vocês não teria valido a pena. Vocês foram os meus melhores amigos, pais, companheiros, confidentes... E acima de tudo, obrigado por me terem ensinado a ser uma pessoa humilde, generosa e amiga do próximo. Vocês foram incansáveis para que eu pudesse chegar aqui, e por isso, estarei-vos eternamente agradecido. Os meus Pais são os meus heróis! Para a minha querida irmã, um obrigado também muito especial. Quero também agradecer a toda a minha familia que sempre se preocupou comigo e com a minha participação no curso que agora termino, em especial aos meus primos Filipe, Ricardo, Pedro, Tiago e Rodriguinho, às minhas primas Ângela, Ana e Claudia, aos meus tios Carlos e Sara, Rosa, Cândido e à minha madrinha Fernanda. Não me podia esquecer também da minha Avó Regina e do meu Avô Benjamim (que tantas vezes sonho contigo), obrigado por tomarem conta de mim quando os meus Pais tinham que ir para o trabalho, por terem brincado comigo e pelos momentos bem divertidos que passamos juntos, como por exemplo, vestir a bata todos as manhas para ir para o infantário.

Por último, mas não menos importante, quero agradecer à minha namorada, e acima de tudo à minha melhor amiga, Sandra ("Flor"). Aposto que agora já sabes umas coisitas sobre programação e de certeza que sabes responder à pergunta: quantos bits tem um byte? A tua ajuda e compreensão foi fundamental para percorrer todo este caminho, sem ti não teria sido tão agradável. "Flor, obrigado por tudo!"

Porto, February 14, 2012

Zé Carlos

“Code without tests is broken as designed”

Jacob Kaplan-Moss

Contents

CONTENTS

List of Figures

List of Tables

LIST OF TABLES

List of Algorithms

Abbreviations

APFD Average Percent of Fault-Detection

API Application Programming Interface

ARC Ames Research Center

ASEJ Automated Software Engineering Journal

BN Bayesian Networks

CDG Control Dependence Graph

CFG Control Flow Graph

CPT Conditional Probability Distribution Table

CPU Central Processing Unit

CSP Constraint Satisfaction Problem

CVS Concurrent Versions System

DA Diagnosis Algorithm

FCA Formal Concept Analysis

FEUP Faculty of Engineering - University of Porto

IDE Integrated Development Environment

IJCICG International Journal of Creative Interfaces and Computer Graphics

IJUP Investigação Jovem na Universidade do Porto

IL Integer Programming

ILP Integer Linear Programming

ISSTA International Symposium on Software Testing and Analysis

LOC Lines of Code

LRU Last Recently Used

MS Mutation Score

MSC Minimum Set Cover

ABBREVIATIONS

OpenGL Open Graphics Library

PARC Palo Alto Research Center

RTS Regression Testing Selection

SIR Software Infrastructure Repository

SUT Software Under Test

TSL Test Specification Language

VM Virtual Machine

Chapter 1

Introduction

During the software development cycle, the phase of tests is in general the most important method to determine whether the software diverge from the requirements or not. There, testing is a princial part of the development and maintenance phases of any software system, from the very small to the huge. Therefore, probably, the intension of testing increases exponentially with the product size and its desired level of reliability required [LH88].

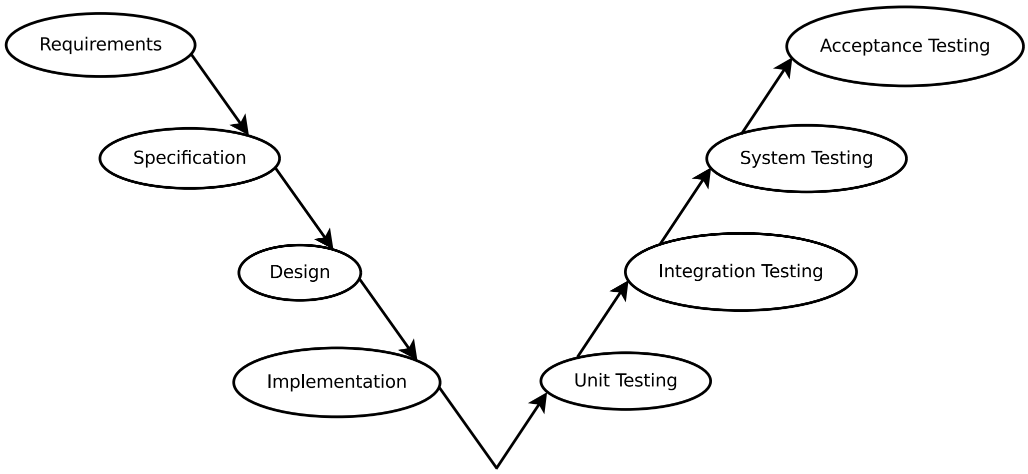


Figure 1.1: V-model of the Systems Engineering Process [Som07].

Normally, software testing is executed in different phases and there is a short relationship with the several phase of the cycle of software development. As we can see in Figure 1.1, at the time of module development, *unit testing* is accomplished. When main software components are combined to produce several of the subsystems, *integration testing* occurs. When all system exists as a complete entity, happens the *system testing* phase. Finally, *acceptance testing* is the phase of testing used to verify if the system cover all requirement which specified in the requirements analysis phase. In theory, all test phases must complement each other and share the same goal, which is, try to hide faults that appears in specification, and/or implementation of the software [VF98].

1.1 Regression Testing

Software that has been changed (to fix some know fault or to delete/add new requirement) must be validated (retested) with focus in the following objectives: (1) assure that the new requirements have been implemented correctly; (2) ensure that new requirements does not affect previous functionalities (which continue working as expected); (3) test those parts of the software that have not been checked before. The process of retesting the software to ensure that modified program still work correctly with all of the test cases used to test original program (objective 2) is know as *regression testing* [VF98].

In recent years, the software testing research community have given enough importance to the subject of regression testing. Some approaches have been presented to maximize the test suite of each SUT: minimization, selection and prioritization. Test suite minimization is the technique to denote what test cases are redundant or obsolete and then remove them from the test suite. Test suite selection choose a subset of test case that will be used to test the parts that have been changed in software. Finally, test suite prioritization identify the best order of test case that maximize some property, such as fault detection [YH10].

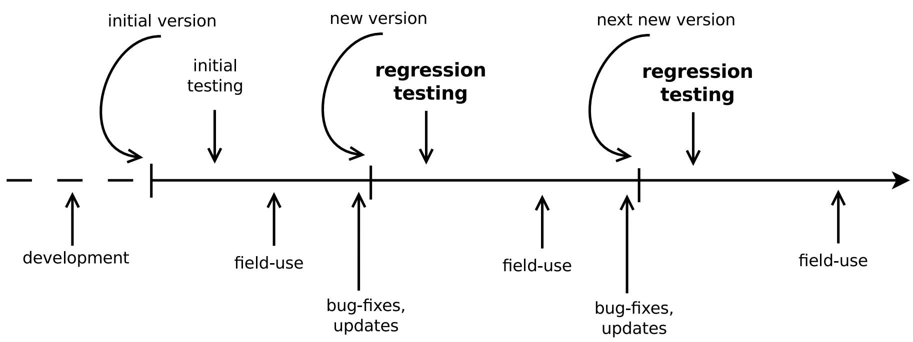


Figure 1.2: Life of a Software System [AHKL93].

In Figure 1.2, Agrawal *et al.* [AHKL93] describes a common example of a timeline during the life of a software system. As we can see, the execution of regression testing is a considerable fraction of the system's lifetime. Unfortunately, regression testing cannot always be completed because of the frequent changes and updates of a system. When a program has been modified, it is indispensable to guarantee that the changes work correctly but also to ensure that the unmodified modules of the program have not been influenced by the modifications. It it extremely necessary because even small modifications, in one part of a program, may have a negative impact in other independent parts of the program. However the changed program can produce correct outputs with test cases particular constructed for that version. But it can produce incorrect outputs to other test cases which original program returns correct outputs. So, during the regression testing, the changed program is executed with all regression tests to check if it maintains the same functionalities present in the original program (except new changes) [AHKL93].

1.2 Motivation

Software development is an iterative process: there are new requirements, some need to be reworked, removed, implement new features, etc. And the best existing bug prediction techniques in the literature predict that modifications on original software introduce bugs in 78% of times [KWZ08]. A software bug is the term used in informatics to describe a flaw, mistake, or fault in computer program that produces an incorrect or unexpected result. The results of bugs may be extremely serious.

Failures in software occur every day, and what is worse is that people are losing jobs and in some cases their liberty for software failures which can be preventable. Some examples of software failures in 2011[1]:

Computer system bugs cause Asian banking facilities' downtime "Computer system problems at one of Japan's largest banks resulted in a nationwide ATM network of more than 5.600 machines going offline for 24 hours, internet banking services being shut down for three days."

Cash machine bug benefits customers by giving them extra money "An Australian bank began giving out large sums of money from 40 cash machines across one city. Officials at the company said they were operating in stand-by mode, so could not identify the account balances of customers."

Bugs in social networking app for tablet just hours after delayed release "Just hours after its release, this social networking sites' long-awaited tablet app was already receiving reports about minor bugs from clicking through to pages via panel icons to problems posting comments."

Other example happened in 2002: "A study commissioned by the US Department of Commerce' National Institute of Standards and Technology concluded that software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated $59 billion annually, or about 0.6% of the gross domestic product"[2]. Software bugs is a consequence of bad quality politics or the nature of human errors in the programming task. So, it is very important testing the software. Testing is usually performed for the following purposes:

- **To improve quality**. Actually, computers and software are used in critical applications and the result of a bug can cause very serious damages.

[1] http://www.businesscomputingworld.co.uk/top-10-software-failures-of-2011/, 2012.
[2] http://www.nist.gov/itl/csd/bugs_110910.cfm, 2012.

- **For Verification & Validation (V&V)**. Testing can be use such as metrics and actually is heavily used as a tool in the V&V process. People how make tests, can interpret and report the testing results, if the program work as defined or it does not work [Het88].

- **For reliability estimation**. Software reliability is an important measure for aspects related with software: it structure, of testing it has been subjected to. Based on an operational profile [Lyu96], testing can provide as a statistical sampling method to gain failure data for reliability estimation.

Therefore, when changes are make to the software, it is really important to re-execute the test to guarantee that features which already exist (before modifications) and new features introduced are working as expected. Normally, to re-test the software, regression test exercise all the test cases that were used to test the software before modifications were made. In practice, essentially on large software systems, re-execute all test case can not be practical, mainly because the time and cost associated. In these situations testers at organization should decide which test cases to use in their regression testing [VF98].

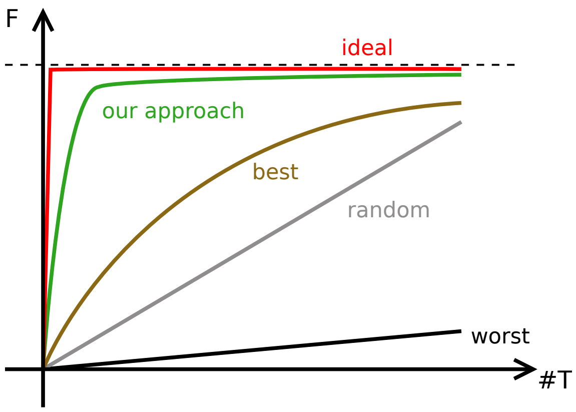


Figure 1.3: Detect Faults Early.

As we can see in Figure 1.3, the worst regression approach mentioned in the literature, requires a many test cases to detect a minimum faults in the software. In the random approach, the number of test cases is proportional to the faults founded. The best approaches mentioned in the literature can already have a good ratio between number of detected faults and test cases. However, we think there is an opportunity for us to improve the existing regression techniques. The ideal approach with very few test cases immediately find all the teorical faults in the software. Thus, is very hard and its necessary an ideal heuristic to do that, but this heuristic does not exist. So our approach focuses on trying to find more faults with fewer test cases as possible. Once the number of test cases increases, the curve of our approach will converge to the total number of failures (dashed line, hypothetical failures that may be in software).

1.3 Objectives

The adoption of regression testing techniques (minimization, selection and prioritization) remain limited, because testers does not have tools available which implement those techniques. For unit testing there are a number of frameworks based on the xUnit architecture [Mes07], and a weakness of regression testing is providing tools to support the studies that have taken in this area.

This is the first goal of this project, add to GZOLTAR the feature of unit testing. GZOLTAR [RA10] is an Eclipse plug-in [McC06] for automatic debugging, which integrates Zoltar [JAG09] core processing, along with other tools, to display powerful graphical visualizations representing the SUT, and indicating which component have higher failure probability. The main goal of these choice are to allow a developer to adopt this tool without much effort, by using the Integrated Development Environment (IDE) such as Eclipse, to test case management. Then, the objective is to implement several approaches for techniques of regression testing: minimization, selection, prioritization; which can reduce the effort time of re-test the software.

The main goal of this project is to **develop an ecosystem where the developed can re-test and debug their software, with new latest studies in the field of regression testing and debugging software**.

1.4 Concepts

Definition 1. Test Suite is a collection of test cases which purpose is to be utilized to test a software system.

Definition 2. Test case is a collection of settings or variables which every testers use to determine if a software system works as expected.

Definition 3. Minimum Set Cover (MSC) is the problem of finding the minimum sub collection (cover) of subsets whose union gives elements cover all the elements of main set.

1.5 Contribution

Overall, this thesis makes three contributions described as follow:

- We proposed a constraint solver-based novel technique for test suite minimization, selection and prioritization;
- The proposed technique has been implemented in the GZOLTAR toolset, this way providing an ecosystem for Debugging & Testing software programs;
- We have empirically analyzed the added value of our method using large-real world programs.

The work proposed in this thesis has been disseminated as follows. A paper, covering Chapter 4, has been published in Investigação Jovem na Universidade do Porto (IJUP), 2012. Section 5.2 has been included in the paper submitted for publication in the International Journal of Creative Interfaces and Computer Graphics (IJCICG), 2012. Currently, we are preparing a submission to the top-tier International Symposium on Software Testing and Analysis (ISSTA), 2012, which will mainly discuss the proposed technique described in Chapter 3 as well as the results discussed in Chapter 4. Finally the toolset is going to be submitted for publication for a special edition on tools in the Automated Software Engineering Journal (ASEJ). Most publications are co-authored with Rui Abreu and André Riboira.

1.6 Document Structure

In this section will be presented this document structure. Apart from Introduction, this report has five more chapters.

Chapter 2 contains the state-of-the-art in regression testing field.

Chapter 3 presents our approach to test suite minimization, selection and prioritization on MSC problem.

Chapter 4 describes in detail the architecture of GZOLTAR Project (where RZOLTAR will be implemented).

Chapter 5 report the empirical evaluation on applying our approach to some open-source projects, and results about user studies.

Chapter 6 has the conclusions about this project, and also some ideas for future work.

Chapter 2

State of the Art

During the life-cycle of software development, software testing try to detect errors as early as possible to assure that modifications to existing software do not break the software. When software evolves test suites are frequently reused and updated. This resulting in a large test suite which can has some test cases considered redundant (requirements covered by these tests are also covered by other test cases). For re-executing large test suites is necessary resources and time, so it is important to develop techniques to minimize test suite by eliminating redudante test case, techniques to select test cases which are important to identify the parts of the SUT that have been modified, and techniques to prioritize test cases that find faults more earlier as possible, or maximize the rate of fault detection [TG05].

This chapter as organized as follow: Section 2.1 will be described the related work of technique for test suite minimization, in Section 2.2 will be described the studies of technique for test suite selection. In Section 2.3 will be described the studies of technique for test suite prioritization. Finally, the last Section 2.4 describe the perform tests that we conducted to choose between two testing frameworks.

2.1 Techniques for Test Suite Minimization

Technique to find and remove the redundant test case from the test suite is called as test suite minimization. Test cases become redundant because (1) their input/output relation is no longer meaningful due to changes in program, (2) these tests were developed for a specific program that has been modified or (3) their structure is no longer in conformity with the software coverage. Minimization or "test suite reduction" means a constant elimination, even so, the two concepts are related since all techniques act as a momentary subset of the test suite, in spite of only minimization techniques can always remove test cases [YH10].

More formally, following Harrold and Guppta [HGS93], test suite minimization is defined as follows:

Definition 4. (Test Suite Minimization Problem). Given: A test suite T, a set of test requirements $r_1, \ldots, r_n$, that must be satisfied to provide the desired testing coverage of the program, and subsets of T, $T_1, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i's.
Problem: Find a representative set, T', of test cases from T that satisfies all r_i's.

The testing criterion is satisfied when every test requirement in $r_1, \ldots, r_n$ is satisfied. A test requirement, r_i, is satisfied by any test case, t_j, that belongs to the T_i, a subset of T. Therefore, the representative set of test cases is the hitting set of the T_i's. Furthermore, in order to maximize the effect of minimization, T' should be the minimal hitting set of the T_i's [YH10]. Trying to find the minimal hitting set[1] of a test suite that covers the same set of requirements by the original test suite is a NP-complete problem and this can be show by a polynomial time reduction from the MSC [GJ90] problem.

Trying to find the minimal test suite that covers the same set of requirements as the original one is a NP-complete problem, but can be solved in a polynomial time using the minimum set cover problem [GJ90]. NP-completeness of the test suite minimization problem encourages the usage of heuristics. Precedent work on test case minimization has advanced the state-of-the-art of heuristic approaches to the minimal hitting set problem, [Chv79, OPV95, HGS93, TG05, JG05, BMK04, HO09].

Chavatal [Chv79] proposes the usage of a greedy heuristic that chooses test case which covers almost all requirements about to be covered, until all requirements have been accomplished. This algorithm has been commonly seen as a solution to the minimum set-cover problem and an upper limit of how far apart it can be from the optimal size solution in the worst case that has been tested [Cor01]. However, a potential weakness of the greedy approach is that the early selection made by the algorithm can eventually be rendered redundant by the test cases subsequently selected and when there is a tie between multiple test cases, one test case is randomly selected [YH10]. In Table 2.1 Chavatal [TG05] consider that test suite and testing requirements to explain their approcah. The greedy approach will select t_1 first as it satisfied the maximum number of testing requirements, and then continues to select t_2, t_3 and t_4. However, after the selection of t_2, t_3 and t_4, t_1 is considered redundant.

This heuristic only takes advantage of the implications among test cases in order to find out which test case became excessive while reducing a test suite [TG05].

Offutt, Pan, and Voas [OPV95] dealt equally with the test suite minimization issue as the dual of the minimal hitting set problem, i.e. the set cover problem. Their heuristic can equally be looked at as a variation of the greedy approach to the set cover problem. Nonetheless, they implemented various teste case orderings, rather than the fixed ordering in the ambitious approach.

[1] A minimal hitting set is an hitting set such that none of its subsets is an hitting set.

Test case	Testing Requirements					
	r_1	r_2	r_3	r_4	r_5	r_6
t_1	x	x	x			
t_2	x			x		
t_3		x			x	
t_4			x			x
t_5					x	

Table 2.1: Example test suite taken from Tallam and Gupta [TG05]. The early selection made by the greedy approach, t_1, is rendered redundant by subsequent selections, t_2, t_3, t_4.

Their empirical study pulled up techniques to a mutation-scored-based test case minimization, reducing the size of test suites by over 33% on average [YH10].

Another greedy heuristic, based on a determined number of test case covering specific demand, has been developed by Harrold, Gupta, and Soffa [HGS93] to choose a minimal subset of test case which covers the same set of demands as the un-minimized suite [TG05]. The proposal by Harrold *et al.* [HGS93] generates solutions which are always equally good or better than the ones computed by Chavatal [Chv79], although they have the worst case execution time of $O(|T| \times max(|T_i|))$ [YH10]. In this case, $|T|$ represents the size of the original test suite, and $max(|Ti|)$ represents the cardinality of the largest group of test cases among $T_1, \ldots, T_n$ [YH10]. Harrold *et al.*'s [HGS93] technique is more expensive than the ping-pong procedure presented by Offutt *et al.* [OPV95], and takes more information (specifically information as to which test cases satisfy each requirement). It appears that HGS technique is just as effective as OPV, and probably more efficient [OPV95].

The early selection made by the greedy algorithm is likely to be a weakness of the greedy approach, which can eventually provide redundant by the test cases subsequently selected. Tallam and Gupta [TG05] tried to overtake the weakness of the greedy approach by developing a concept lattice , a hierarchical clustering based on the relation between test cases and testing requirements [YH10]. Tallam *et al.* [TG05] heuristic, dubbed Delayed-Greedy, upgrades upon the prior heuristic by exploiting the implications and between the test cases and the implications between the coverage demands, were able to influence only independently from each other in the previous work [TG05]. Delayed-Greedy performs in three main phases: (1) apply object reductions (i.e., remove test cases which test requirements is subsumed by other test cases); (2) apply attribute reductions (i.e., remove test requirements that are not in the minimal requirement set); (3) build reduced test suite from the remaining test cases using a greedy method [HO09]. Empirical it was observed that, the test suite minimized by the 'delayed-greedy' approach were either the same size or smaller or even smaller than those which had been minimized by the classical greedy approach or by the HGS heuristic [YH10]. All these approaches ([Chv79], [OPV95], [HGS93], [TG05]) focus alone on single standard minimization problems. With this attitude, important dimensions of the issue are neglected and may bring forth test suites which are suboptimal when considering such dimensions. For example, they can generate test suites which contain a reduced number of test cases, while having longer running time than other possible test suites. Or they may generate test suites that are minimal in terms of running time but have a considerably reduced fault detection ability [RHOH98], [WHLB97]. Furthermore, because even single-criterion problem are NP-complete, as demonstrated above, most of these techniques are based on approximated algorithms that make the problem manageable at the cost of computing answers that are suboptimal even for the simplified version of the minimization problem they attempt. Two studies by Rothermel *et al.* [RHOH98] and Wong *et al.* [WHLB97] investigated the boundaries of single-criterion minimization techniques. More precisely, these studies fulfilled experiments to estimate how minimized test suites are effective in terms of fault-detection ability. Their results exhibited that test suites minimized using single-criterion minimization technique might detect a lot less faults than the complete test suites.

The technique proposed by Jeffrey and Guppta [JG05] deals with the limitations of traditional single-criterion minimization techniques by taking in account several sets of testing demands (e.g., coverage of different entities) and introducing selective redundancy in the minimized test suites. Jeffrey *et al.* [JG05] illustrate their approach with an example (see Table 2.2) to generate a satisfactory minimized branch coverage test suite by keeping some redundant test cases in the reduced suite. And collect the coverage information for a secondary criteria, such as the all def-use pairs criteria, for all the test cases in the test suite T.

Test Case	B_1^T	B_1^F	B_2^T	B_2^F	B_3^T	B_3^F	B_4^T	B_4^F
$t1$	X		X			X		
$t2$		X		X	X			X
$t3$		X	X			X		
$t4$		X	X		X		X	
$t5$		X		X	X		X	

Table 2.2: Brach coverage information for test cases in T [JG05].

In the example presented by Jeffrey *et al.* [JG05], after inserting t_1 and t_2 in the minimized suite by HGS algorithm [HGS93], t_3 is described as redundant with respect to branch coverage since all the branches covered by t_3 are already covered by t_1 and t_2. Jeffrey *et al.* [JG05] verify if t_3 is also redundant with respect to the secondary criteria and in this case they found that t_3 covers the def-use pair $x(4,6)$ that is not covered by either t_1 or t_2. So, they do not identify t_3 as redundant and they add it to the minimized test suite. In their approach the minimized test suite at this point is constituted by t_1, t_2, and t_3. Next, either one of t_4 or t_5 can be chosen to cover the branch B_4^T. Jeffrey *et al.* [JG05] added t_4 to the constituted test suite and mark the requirement B_4^T as coverage. At this point Jeffrey *et al.* [JG05] identified the test case t_5 as redundant with respect to branch coverage as well as with respect to the coverage of the secondary criteria. Therefore, the minimized test suite generated by their approach for this example is $\{t_1, t_2, t_3, t_4\}$.

Jeffrey *et al.* [JG05] proceed an empirical evaluation using branch coverage as the first set of testing requirements and all-uses coverage information obtained by data-flow analysis. They compared their results to two versions of the HGS heuristic, based on branch coverage and def-use coverage. Therefore, the results showed by Jeffrey *et al.* [JG05] demonstrate that, although their technique returns larger test suites, the fault-detection adequacy was better preserved compared to single-criteria versions of the HGS heuristic [YH10]. However, in spite of their approaches improve existing techniques, it continues to be an heuristic, approximation [HO09].

An improved attempt at overtaking the limitation of existing approaches is the technique that Black, Melachrinoudis, and Kaeli, [BMK04] proposed which consists on a two criteria variant of single-criterion traditional test suite minimization attempts and computes optimal solutions using an integer linear programming solver [HO09]. A limitation of this approach is that fault detection is reputable concerning a single fault (rather than several faults), and accordingly there may be limited confidence that the limited suite be useful in detecting several other faults. Also, the approach is limited in typed of minimization problems it can handle [HO09].

Hsu and Orso [HO09] such as Black *et al.* [BMK04] also took in consideration that the use of an Integer Linear Programming (ILP) solver with multi-criteria test minimization. Hsu *et al.* [HO09] demonstrate that the majority of minimization techniques proposed have two limitations: they perform minimization based upon a single criterion and produce approximated suboptimal solution. Therefore, they extended the work of Black *et al.* [BMK04] by comparing several heuristics for a multi-criteria ILP formulation: the weighted-sum approach, the prioritized optimization and a hybrid approach. In prioritized optimization, the user assigns a priority to each given criteria. After optimizing for the first criteria, the outcome is added as a constraint, at the same time as optimizing for the second criteria, and so on. However, one possible weakness shared by these approaches is that they require additional input from the user of the technique in the forms of weighting coefficients or priority assignment, which might be biased, unavailable or costly to provide [YH10].

2.2 Techniques for Test Suite Selection

Test case selection, or the Regression Testing Selection (RTS) problem, is basically similar to the test suite minimization problem (defined in Section 2.1). The two technique have the same problems about choosing a subset of test cases from the test suite. The main difference between these two approaches in literature is the focus against the changes in the SUT. While test case selection techniques try to minimize the cardinality of a test suite, the majority of selection techniques are modification-aware. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified modules of the new version of program. Test cases are selected because they are relevant to the changed parts of the SUT, which regulary involves a while-box static analysis of the program code [YH10].

More formally, following Rothermel and Harrold [RH96], selection problem is defined as follows:

Definition 5. (Test Case Selection Problem). Given: The program, P, the modified version of P, P' and a test suite, T.
Problem: Find a subset of T, T', with which to test P'.

Based on Rothermel's *et al.* [RH96] formulation of the problem, it can be said that test case selection techniques for regression testing focus on identifying the modification-traversing test cases in the given test suite. The details of the selection procedure differ according to how a specific technique defines, seeks and identifies modifications in the SUT. Various techniques have been proposed using different criteria including Integer Programming (IL) [FRC81], program slicing [HGS93], dynamic slicing [AHKL93], graph-walking [RH97], textual difference in source code [VF98], and modification detection [CRV94].

Fisher, Raji, and Chruscicki [FRC81] developed an approch that use IL to try resolve the selection problem. One weakness in Fischer's approach is its inability to deal with control-flow changes in P'. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$, depends on the control-flow structure of the SUT. If the control-flow structure changes, the test case dependency matrix can be updated only by executing all the test cases, which negates the point of applying the selection technique [YH10].

The criteria for testing data flow, is based on the tracking of a variable values through a program. These are recognized by the variable's names and definition locations, but when a variable name denotes multiple values, they can be tough to track. This problem shows up with the use of arrays and pointers [Tah92]. Harrold, Gupta, and Soffa [HGS93] applied program slicing technique to recognize definition-use pairs which are influenced by a code modification. Using slice technique allows identification of definition-use pairs in need to be tested without representing a complete data-flow analysis, often very expensive. One impotence that all data-flow analysis based test care selection technique share, is the fact that they are not able to detect changes which are not related to data-flow change [YH10]. For instance, if P' has new procedure calls without a parameter, or changed output statements which have no variable uses, data flow techniques might not choose test cases executing these.

Agrawal, Horgan, Krauser, and London [AHKL93] presented a family of test case selection techniques supported by different program slicing technique. An execution slice of a program concerning a test case is something which is usually referred to as an execution trace for it is the set of statements executed by the given the test case. A dynamic slice of a program concerning a test case is the group of statements in the execution slice that influences an output statement. An execution slice is able to contain statements that do not affect the program output, so a dynamic slice is a subset of an execution slice. To make a more precise selection, Agrawal *et al.* suggested additional slicing criteria: a relevant slice and an approximate relevant slice. An relevant slice of a program concerning test case is the dynamic slice concerning the same test case together with all the predicate statements in the program that, if analyzed in a different way, could have been the cause for the program to produce a different output. An approximated relevant slice is a more preservative approach to include predicates which could have caused a different output; for it is the dynamic slice including all the predicate statements in the execution slide [YH10]. Agrawal *et al.* built, in the first place, their technique on cases with restricted modifications to those which do not change the Control Flow Graph (CFG) of the program that is being tested. For as long as the CFG of the program continues to be the same, their technique is secure and can be looked at as an improvement over Fisher's IL approach [FRC81] in its whole. Slicing erases the necessity to formulate the linear programming issue, decreasing the required effort from the tester [YH10].

The technique developed by Rothermel and Harrold [RH97] bases itself on the idea of CFG to perform and compare the older and newer versions of a program. Their algorithm constructs graphs representing control dependence for a program and its modified version, and uses these graphs to identify tests that are potentially revealing and tests that cannot possibly expose errors. Their algorithm detects regions of code that differ in the two versions of the program, and selects for retest all tests that traverse these regions. Harrold *et al.* concluded this is a safe technique, able to select a pretty precise subset of the test cases. It was implemented in a prototype research tool named DejaVu.

Volkolos and Frankl [VF98] suggested a selection technique based upon the textual difference between the source code of two versions of SUT. They identified partes which had been modified of SUT by applying the `diff` Unix tool to the source code of a different version. The source code was previously processed into canonical forms to remove the impact of cosmetic differences. Although their technique operates on a different representation of SUT, its behavior is very similar to the CFG-based graph walk approach [YH10]. After running various experiments, they concluded, concerning the time needed for the analysis, it appears based on their data, that textual differencing is a lot faster than the technique previously developed by Harrold *et al.* [RH97].

Chen, Rosenblum, and Vo [CRV94] introduced a testing framework called TestTube, which utilizes a *modification based* technique to select test cases. TestTube partitions the SUT into program entities, and monitors the execution of test cases to establish connections between test cases and the program entities that they execute. TestTube also partitions *P*' into program entities, and identifies program entities that are modified from *P*. All the test cases that execute the modified program entities in *P* should be re-executed [YH10].

Chen *et al.* explained the idea behind TestTube in Figure 2.1. In the Figure 2.1, the boxes represent subprograms and circles represent variables. The arrows represent static and dynamic dependency relationship (e.g., variable references and function calls). The shaded entities were modified to create a new version of the SUT. Using s simple technique such as retest-all, all three test units must be rerun in order to test the changes. However, Chen *et al.* concluded that relationship between the test units and the entities they cover, it is possible to eliminate test unit 1 and 2 from the regression testing of the new version and rerun only test unit 3.

TestTube is less precise [YH10] than Rothermel *et al.* [RH97] tool, DejaVu, but more resistant in analyzing large software systems [VF98]. One weakness of TestTube is pointer handling. By

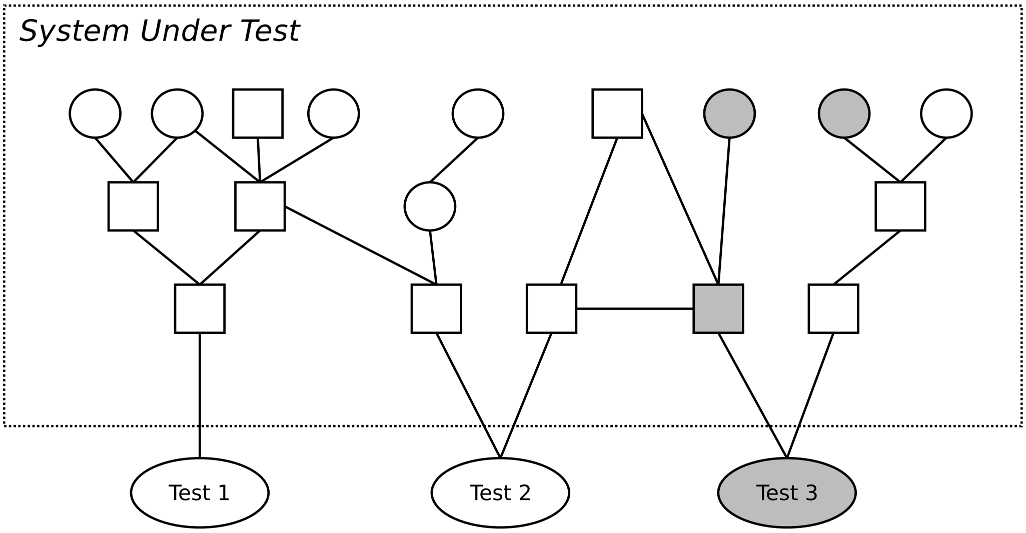


Figure 2.1: Select minimum set using Chen *et al.* approach, TestTube [CRV94].

including variable and data types as program entities, TestTube requires that all value creations and manipulations in a program can be inferred from source code analysis. This is only valid for languages without pointer arithmetic and type coercion.

2.3 Techniques for Test Suite Prioritization

Prioritization technique order the test cases based on defined criteria such that the test cases with a low chance of finding defects are executed at the end, and that test cases with a high probability are executed first [MT08]. This technique provide to testers the possibility to order their test cases so that test cases with large priority (according to some criterion), are executed first and after test cases with less priority are executed in the regression testing process. For example, testers might wish to reorder test cases in an order that achieves code coverage at the fastest rate possible, exercises subsystems in an order that reflects their historically demonstrated propensity to fail [RUCH01]. More formally, following Rothermel and Harrold [RH96], prioritization problem is defined as follows:

Definition 6. (Test Case Prioritization Problem). Given: A test suite, T, the set permutations of T, PT, and a function from PT to real numbers, $f : PT \rightarrow \mathbb{R}$.
Problem: To find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Average Percent of Fault-Detection (APFD) is commonly used to evaluate test case prioritization techniques. The APFD value for T' is calculated as follows [EM02]:

$$APDF = 1 - \frac{TF_1 + \ldots + TF_m}{nm} + \frac{1}{2n}$$

Maximization of APFD would be possible only when every fault that can be detected by the given test suite is already know. This implies that all test cases have already been executed, which does the need to prioritize. APFD is computed after the prioritization only to measure the performance of the prioritization technique [YH10].

Various approaches have been proposed using different techniques and criteria, [RUCH01, DRK04, LP03, KP02, MT08].

Rothermel, Untch, Chu, and Harrold [RUCH01] describe their approach with nine different test case prioritization techniques: untreated (no prioritization), random (randomized ordering), optimal (ordered to optimize rate of fault detection), statement-total (prioritize in order of coverage of statements), statement-additional (prioritize in order of coverage of statements not yet covered), branch-total (prioritizes in order of coverage of branches), branch-additional (prioritize in order of coverage of branches not yet covered), FEP-total (prioritize in order of total probability of exposing faults), FEP-additional (adjusted to consider effects of previous test cases).

To illustrate the APFD measure obtained with Rothermel *et al.* [RUCH01] approach, consider an example program with 10 faults and a test suite of five test cases, A through E, with fault detecting abilities, as shown in Table 2.3.

	Faults									
Test	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Table 2.3: Test suite and faults exposed [RUCH01]

Rothermel *et al.* first consider the test cases in order **A-B-C-D-E** to form a prioritized test suite T_1. After running test case A, two of the 10 faults are detected; thus, 20% of the faults have been detected after 0.2 of test suite T_1 has been used. After running test case B, two more faults are detected and, thus 40% of the faults have been detected after 0.4 of the test suite has been used. The APFD is 50% percent in this example. When the order of test cases in changed to **E-D-C-B-A**, yielding test suite T_2, a “faster detecting” suite than T_1 with an APFD of 64%. Changing to **C-E-B-A-D**, by inspection, they concluded this ordering results in the earliest detection of the most faults and illustrates an optimal ordering with an APFD of 84%. Their empirical results concluded that random prioritization could exploit the weakness of untreated test cases in several cases [YH10].

Do, Rothermel, and Kinneer [DRK04] applied the most popular unit testing framework, JUnit, to coverage-based prioritization technique. The results show that prioritized execution of JUnit test cases improved the detection of faults. They finding that random prioritization sometimes resulted in an higher value of APFD than the untreated ordering, i.e., in the order of creation, newer unit tests are executed later. With the difficulty to known what tests expose a fault, test case prioritization techniques depend on surrogates, assuming that early maximization of a surrogate criteria will result in maximization of earlier fault detection. In a controlled-regression-testing environment, the result of prioritization can be measured by executing test cases according to the fault detection rate [YH10].

Leon and Podgurski [LP03] developed a prioritization techniques that integrate coverage-based prioritization with distribution-based prioritization. This hybrid approach is based on the observation that basic coverage maximization achieve a reasonably performs when as compared to repeated coverage maximization. Repeated coverage maximization, after reaching 100% coverage, starts again from 0% of coverage repeating the prioritization of test cases. In the other hand, basic coverage maximization stops prioritizing when 100% coverage is accomplished. Leo *et al.* observed that the fault-detection rate of repeated coverage maximization is not as high as that of basic coverage maximization. This motivated them to consider a hybrid approach that first prioritizes test cases based on coverage, then swap to distribution-based prioritization as soon as the basic coverage maximization is accomplished. They considered two different distribution-based technique: one-per-cluster and failure-pursuit. The one-per-cluster approach choose one test case from each cluster, and prioritizes them according to the other of cluster creation during the clustering. The failure-pursuit approach act equivalently, but it adds the k closest neighbors of any test case that reveal a fault. They [LP03] concluded that distribution based techniques can be as efficient or more efficient for revealing defects than coverage-based technique, but two techniques are also interconnected in order that they find different faults.

Kim and Porter *et al.* [KP02] proposed a history-based approach to prioritize test cases that are already selected by RTS. If the number of test cases selected by an RTS technique continue to high, or if the execution costs are too high, then the selected test cases may have an addition prioritization. After all, the relevance to the recent change in SUT is deduced by the use of an RTS technique, Kim *et al.* focus on the execution history of each test case, extracting this information from statistical quality control. Kim *et al.* define Last Recently Used (LRU) prioritization by using test case execution history as H_{tc} with α value that is as close to 0 as possible. The empirical evaluation showed that the LRU prioritization approach can be competitive in a severely constrained testing environment, i.e. when executing all test cases selected by an RTS technique can not be done [YH10].

Mirarab and Tahvildari [MT08] introduced a probabilist approach to prioritization problem which utilizes Bayesian Networks (BN) and a *feedback* mechanism. BN [Pea88] is a directed acyclic graph consisting of three elements: *nodes* representing random variables, arcs representing probabilistic dependencies among those variables, and a Conditional Probability Distribution Table (CPT) for each variable, which includes the conditional probabilities of outcomes of its variable given the valued of all its parents. BN is used to predict the probability of each test case finding a fault using different several sources of information. Mirarab *et al.* conduct an extensive empirical study of evaluate the performance of the BN approach. The controlled experiments compare several different realizations of the proposed approach, and using mutation technique which can provide large number of faults. The objective of their controlled experiments is to compare different implementations of the proposed approach and to understand the effects of its underlying parameters. They concluded that most of the other prioritization techniques from the literature are reported to perform more efficiently when used in conjunction with feedback [EM02].

2.4 Unit Testing Framework

For the development of this project was necessary to find a framework for the execution of automated unit test. A important benefit of such tools is their capability to execute unattended (e.g., to run overnight or over weekend), providing to the testing task a significant gain in productivity. Automated tools can also provide more confidence in the SUT by allowing more tests to be executed in the same time as that taken for an equivalent manual test [Wat01]. And the chosen framework had allows easy integration with GZOLTAR architecture, in particular with JaCoCo [Hof11a] (Java-based API that offers code coverage capabilities) module. So, for comparison we chose two frameworks: JUnit and TestNG.

JUnit[2] is the Java version of popular xUnit framework and it can be describe as the father of automated unit test, where frameworks like NUnit and PHPUnit its based. JUnit like others frameworks exist for other programming languages and this show how can versatile the framework can be.

TestNG (the NG stands for Next Generation)[3] is a framework for automated unit test too, inspired in JUnit and NUnit but with some improvements. It introduced some new features like groups of tests, parallel testing, etc.
In next sections we present more details about differences between JUnit and TestNG.

2.4.1 Features

In Java, the most popular frameworks for unit test are JUnit and TestNG and both are very similar in features. Both make easy the test task making it practical and simple and also have a vast community that supports its continued development. We can see the differences between frameworks just in Core design. JUnit was always a framework directed to unit test, was developed to make easy the test of simple objects, and it does this very efficient. Otherwise, TestNG was developed to raise the level of testing (unit, functional, end-to-end, integration, etc) and therefore has some features not available in JUnit.

In Table 2.4 we can see a brief resume about the most important features present in both frameworks.

[2] Kent Beck. JUnit Homepage `http://kentbeck.github.com/junit/`, 2012.
[3] Cédric Beust. TestNG Homepage `http://testng.org/doc/index.html`, 2012.

Feature	Framework	
	JUnit 4.9	TestNG v5.14.10
Annotation	YES	YES
Supported by Eclipse	YES	YES
Parallel test	YES	YES
Dependence between tests	NO	YES
Groups of tests	YES	YES
Objects	EasyMock	DataProvider
Parametrized tests	YES	YES
Threads	NO	YES
First Release	1997	2004

Table 2.4: Features comparison between JUnit and TestNG frameworks

After the version 1.5 of Java, the platform has support to *Annotations*. *Annotations* is a very special form of syntactic meta-data which can be added to our Java code, and allow to annotate packages, class, methods, parameters, variables. TestNG support this features since the beginning.

The most significant difference between JUnit e TestNG is dependencies between tests. TestNG support this feature naively and JUnit not. This implies that beyond simple unit testing, with TestNG we can also perform integration tests.

If we have some code that will create threads on execution, we need TestNG to test them. TestNG support threads test but JUnit not, because it run every tests on one single thread. However, JUnit is in the development environment for years, while TestNG appeared only in 2004.

2.4.2 Performance

As we can see, both frameworks are very identical, but until now we have not considered the parameter performance. To test the performance of both frameworks, we developed a trivial project called Calculator Project (see Section 2.4.2.1).

In every experience of project was measured the execution time of test suite for every framework, JUnit and TestNG. During the experiences that test suite was increased in size.

2.4.2.1 Calculator Project

Calculator Project is a simple project developed in Java that simulates a normal calculator with functions like: add, subtract, module, maximum, etc. To test the functionalities of Calculator first we write ten tests, one test for every feature. Afterwards we replicate the test in order to increase the size of test suite. In Table 2.5 we can see the average results for each framework and the size of test suite. For every details about experiences and its results, consult Appendix B.

Test Suite Size	Framework		
	JUnit	TestNG	TestNG (with 5 threads)
10 Tests	0.024s	0.411s	0.331s
50 Tests	0.063s	2.009s	0.518s
100 Tests	0.095s	3.999s	1.188s
500 Tests	0.296s	20.116s	7.351s
1000 Tests	0.516s	40.167s	19.310s

Table 2.5: Average execution time of JUnit and TestNG in Calculator Project

As we can see, for suites test with size more than 500, the performance of TestNG framework becomes unacceptable. While JUnit framework maintain quite low execution times, even with 1000 tests.

2.4.3 Summary

For automated unit testing we need to choose a framework to do that, so we evaluated JUnit and TestNG. We discuss the features differences between both frameworks and compare them about performance. Although the TestNG has some something innovative in the area, enabling the creation of high level testing (integration testing, functional, etc.) but your performance is not satisfactory. One goal of this project is minimizing the time of test phase, without affecting the confidence in testing and in project quality, so the framework chosen should run as fast as possible. We concluded the best option for this project is JUnit.

Chapter 3

Test Suite Minimization, Selection, and Prioritization

A change in a program causes a test case to become *obsolete* by removing the reason for the test case's inclusion in the test suite. A test case is *redundant* if other test cases in the test suite provide the same coverage of the program. Thus, because obsolete and redundant test cases, the test suite has a needless test cases which make the test suite to enlarge as changes are made. Reducing the size of the test suite decreases the effort of maintaining the test suite and the number of test cases that should be rerun subsequently modification in the software. Therefore, it is important to determine a minimum set of test cases that provides the same coverage of the changed parts of the program. This selective technique, choose the tests from the old test suite that are necessary to test new and old features of the program. Another technique for supporting regression testing is prioritization technique. The test cases presented in the test suite can be ordered based on certain criteria such that the cases with a higher probability of finding faults are executed earlier. These techniques help testers to adjust their time and budget by running as many test cases as they can afford in a given period of time. In this chapter, we present a new approach to test suite minimization, selection and prioritization on MSC problem. The process of selecting a MSC of test cases, such that all feasible required components of a program are covered, can be modeled by a *Coverage Matrix* (explained in the next section). Such matrix stores the relation between execution of test cases and involvement of components, which in turn can be mapped to several *constraints*.

To the best of knowledge this approach has not been presented before and has proven to have a significant reduction of suite tests, maintaining the same percentage of reliability (see Chapter 5).

The reminder of this chapter is organized as follows. We start by introducing the coverage matrix, followed by some concepts about MSC. Subsequently, transform the coverage matrix into constraints is described, followed by how to solve the Constraint Satisfaction Problem (CSP) ant then the time/space complexity are presented. Finally, a summary of this chapter is given.

3.1 Coverage Matrix

White-box software testing methods examines the physical structure of a program, highlighting certain structural components of the code which are examined during the testing process. Each test case from the test suite, when run through the program, will cover a subset of the total set of structural components. By tracking which components each test case activate (covers), a $N \times M$ binary matrix A, can be created (see Figure 3.1).

$$\begin{array}{cc} & M \text{ components} \\ N \text{ test cases} & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix} \end{array}$$

Figure 3.1: Coverage Matrix, N means tests executions, M means a component of SUT, and a_{ij} means coverage.

In Figure 3.1, the entries cross-reference test requirements and tests. Columns correspond to M, different components (source code statements in the context of this manuscript) of a software program, and lines represent an execution of a test case. An element a_{ij} is equal to 1 if and only if component m_j is covered when the test t_i is executed:

$$\omega(t_i,\ m_j) = (a_{ij} == 1\ ?\ true : false) \tag{3.1}$$

That is, while a test may be designed to cover a specific component, other components may be covered incidentally. So, the result of interception between test and component is returned by function $\omega(t_i,\ m_j)$, Equation 3.1.

The minimum set coverage method as described in the next section performs such an analysis and produces a reduced test suite which still provides 100% test coverage of components.

3.2 Minimum Set Coverage

The MSC problem is one of the class of NP-hard problems which has applicability to the scope of software testing [GJ90].

Let S be a collection of N non-empty sets $S = \{s_1,\ \ldots,\ s_N\}$. Each set $s_i \in S$ is a finite set of components (source code statements in our case), where each of the M components is represented by $m_j,\ j \in \{1,\ \ldots,\ M\}$, so, formally, $s_i = \{m_j \mid \omega(t_i,\ m_j) = 1\}$. A minimum set coverage of S is a set T' such that:

$$\forall s_i \in S, s_i \cap T' \neq \emptyset \ \wedge\ \nexists T'' \subset T' : s_i \cap T'' = \emptyset$$

i.e., each member of S has at least one component of T' as a member, and no suitable subset of T', T'', is a coverage set. During the construction of the minimum set, each test should be analyzes to determine the component which it covers and all t_i in T', must cover at least one component:

$$\forall t_i'' \in T'',\ \omega(t'',\ m) = true$$

There may be several minimum set coverage for S, which constitutes a collection of minimum set coverage $T = \{t_1,\ \ldots,\ t_k,\ \ldots,\ t_{|T|}\}$. In order to maximize the effect of minimization, T'' should be the minimal hitting set of the T_i's.

As an example, consider the coverage represented by the matrix in Figure 3.2.

$$\begin{array}{c c}
 & \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\
\begin{array}{c} t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} & \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{array}$$

Figure 3.2: Matrix example, with four tests (lines) and three components (columns). E.g., the component m_2 is activated when set $\{t_1,\ t_3\}$ is executed.

In coverage matrix (see Figure 3.2), the global set is $S = \{\ \{t_1,t_2\},\ \{t_1,t_3\},\ \{t_4\}\ \}$, where, m_1 is covered by set $\{t_1,t_2\}$, m_2 covered by set $\{t_1,t_3\}$, and m_3 covered by set $\{t_4\}$.

A brute-force approach to compute the collection T' of minimum set coverage for S, would be to iterate through all possible component combinations to (1) check if it is a coverage set, and (2) whether it is a coverage set, if it is minimal, i.e., not subsumed by any other set of lower cardinality. As all possible combinations are checked, the complexity of such an approach is $O(2^M)$. For the example above, the following sets would be checked: $\{t_1\}$, $\{t_2\}$, $\{t_3\}$, $\{t_4\}$, $\{t_1,\ t_2\}$, $\{t_1,\ t_3\}$, $\{t_1,\ t_4\}$, $\{t_2,\ t_3\}$, $\{t_2,\ t_4\}$, $\{t_3,\ t_4\}$, $\{t_1,\ t_2,\ t_3\}$, $\{t_1,\ t_2,\ t_4\}$, $\{t_1,\ t_3,\ t_4\}$, $\{t_2,\ t_3,\ t_4\}$, $\{t_1,\ t_2,\ t_3,\ t_4\}$ to find out that just set $\{t_1,\ t_4\}$ or $\{t_2,\ t_3,\ t_4\}$ are necessary to coverage all components.

3.3 Map Coverage Matrix into Constraints

Identifying MSC of a collection of sets is an important problem in many domains, such as Air Crew Scheduling, in which a set of flights must be covered by a collection of available crew members [Wil99], where the MSC are the solutions for the scheduling problem.

As explained in the previous Section, brute-force algorithms have a cost that is exponential in the number of components. We take however an off-the-shelf CSP toolkit, namely MINION, to solve the MSC. We use a constraint solver to compute the MSC solutions since current toolkits already solve large CSPs problems with up to million of variables (components) and million of constraints. In this Section we present our approach: transform the coverage matrix into constraints, aimed to increase search efficiency. The Section 3.4 describes how to solve the constraints using the constraint solver MINION.

Algorithm 1 Map Coverage Matrix into Constraints

Input: Matrix A, number of components M, number of test cases T
Output: Conjunctions of disjunctions C
$C \leftarrow \emptyset$ ▷ empty conjunction
for all $i \in \{1 \ldots M\}$ **do**
  $C' \leftarrow \emptyset$ ▷ empty disjunction
  for all $j \in \{1 \ldots T\}$ **do**
    if $a_{ij} = 1$ **then**
      $C' \leftarrow C' \vee j$
    end if
  end for
  $C \leftarrow C \wedge (C')$
end for

The key idea, behind our approach is breaking the problem up into a set of distinct constraints (see Algorithm 1), each of which have to be satisfied for the problem to be solved.

For the example (Figure 3.2), to ensure that component m_1 is covered, the test t_1 or the test t_2 needs to be executed:

$$c_1 = \{t_1 \vee t_2\}$$

To coverage the component m_2, the test t_1 or test t_3 are essential, so the mapping is

$$c_2 = \{t_1 \vee t_3\}$$

To coverage the component m_3, it is only required to execute the test t_4

$$c_3 = \{t_4\}$$

So, the final solution is the concatenation of all equations

$$C = \{\ \{t_1 \vee t_2\} \wedge \{t_1 \vee t_3\} \wedge \{t_4\}\ \}$$

3.4 Solve Constraint Satisfaction Problem

Using constraint satisfaction as a global framework, concede the possibility to encode several real world problems, including formal verification, vehicle routing and scheduling. A CSP [FFH+11] can be expressed in a set of variables that should be attributed values from their domains in such a way that a set of constraints must be satisfied. Generally and such as MSC, searching a solution of a CSP is an NP-complete problem. Therefore, several researchers have been studying this field to find restricted classes of CSPs that admit polynomial time algorithm.

Since coverage matrix is a boolean matrix, it can be converted into boolean CSP. Boolean CSPs include as special cases some NP-complete problems, such as SAT. SAT is the prototypical NP-complete problem. However, we can approximate solutions that satisfy the maximum number of clauses in polynomial time.

In this Section we present our solution to the CSP, using the state-of-art solver MINION [GJM06] and Trie structure.

3.4.1 MINION

MINION[1] is a general-purpose constraint solver, with an expressive input language based on the common constraint modeling device of matrix models. Experimental results show that MINION typically runs between 10 and 100 times faster than state-of-the-art constraint toolkits such as ILOG Solver[2], ECLiPSe[3], and GeCode[4] on huge and hard problems. For little problems or instances where solutions are discovered with a simple search, gains are not so well [GJM06].

From the point of view of the user, MINION is a black box which provides few options and is an open source project that continue under development. A individually characteristic of this CSP solver is the fact that variables are represented at hardware level to optimize the solving algorithm (backtracking) [NPW08]. It supports four individually-optimised integer domain types: 0/1 domains, usually used for logical expressions; *bounds* domains, which keep only the lower and upper value of the domain; *sparse* domains, where domain values need not be adjacent, e.g. $\{2, 7, 11\}$; and *discrete* domains, originally the lower and upper bound is defined but support the elimination of valued from between the bounds [GMR07].

Considering the restriction of Section 3.3 it can be easily converted into constraints in MINION syntax:

```
MINION 3

**VARIABLES**
```

[1] MINION Homepage http://minion.sourceforge.net/, 2012.
[2] ILOG Solver Homepage http://www.lkn.ei.tum.de/arbeiten/faq/man/ILOG/CONCERT/concert20/userman/soluser_preface.html, 2012.
[3] ECLiPSe Homepage http://eclipseclp.org/, 2012.
[4] GeCode Homepage http://www.gecode.org/, 2012.

```
BOOL t1
BOOL t2
BOOL t3
BOOL t4

**SEARCH**

VARORDER [t1, t2, t3, t4]
PRINT ALL

**CONSTRAINTS**

watched-or({eq(t1,1), eq(t2,1)})
watched-or({eq(t1,1), eq(t3,1)})
watched-or({eq(t4,1)})

**EOF**
```

Briefly, in this model there are three boolean variables, each variable is identified by 't_i', for i in 1 . . . (N - 1) which represent test cases. In constraints section appears the logical constraint `watched-or` for formalizing all disjunctions. Assuming that the priorities of components and the complexity of test are uniform, MINION produces five possibles sets:

```
1. {t1, t4}
2. {t1, t2, t4}
3. {t1, t3, t4}
4. {t2, t3, t4}
5. {t1, t2, t3, t4}
```

Analyzing the result returned by MINION, the set {t1, t2, t3, t4} is clearly selected as a valid solution, because it is the selection of all test cases in the original suite.
The first set, provided by MINION, is the minimization of original set, from four to two test cases: test number 1 and 4. The result coverage matrix can be found in Figure 3.3.

$$\begin{array}{c c} & \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{array}{c} t_1 \\ t_4 \end{array} & \left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

Figure 3.3: Matrix coverage for first set, {t1, t4}, provided by MINION.

This reduced matrix (Figure 3.3), containing only two tests, obviously has the same complete test coverage as the original matrix. However, if test t1 were omitted from the suite, then three components would not be covered by the remaining test (m_1 and m_2). Similarly, the omission of test t4 would leave component m_3 uncovered.

Likewise, for the other three sets the coverage matrix is described in Figure 3.4.

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

(a) Matrix coverage for first set, {t1, t2, t4}.

$$\begin{array}{c} \\ t_1 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

(b) Matrix coverage for second set, {t1, t3, t4}.

$$\begin{array}{c} \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} m_1 & m_2 & m_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

(c) Matrix coverage for third set, {t2, t3, t4}.

Figure 3.4: Matrix coverage for sets: {t1, t2, t4}, {t1, t3, t4}, and {t2, t3, t4}.

To filter the results provided by MINION (eliminate the redundant set) was used a special data structure, dubbed Trie, explained in next subsection.

3.4.2 TRIE

The term TRIE [Fre60] appeared from retrieval. According with the etymology, the creator, Edward Fredkin, pronounces it /'triː/ "tree". However, other authors pronounce traɪ/ "try".

In computer science, a TRIE is a tree data structure (typically ordered) which can be used to keep an associative array where the keys are commonly strings. The key idea is that strings with a common prefix share nodes and edges in the tree. Each node can have at most k children, where k is one more than the size of the alphabet, and each edge is classified either with a character from the alphabet or a appropriate terminating character. The root node has a child for each distinct first character, α, in the set of strings, and the edge connecting the two is classified with α. Each internal node n has a child for each string that has a prefix corresponding to the characters on the edges in the path from the root to n, read in order. Searching for a string in a trie (the main operation useful for supporting a table constraint), is $O(d)$ where d is the length of the string [GJMN07].

The main advantages of TRIE:

- Ease with handling sequences of several lengths;
- Add and delete can be facility achieved;
- Speed of storage and access;

The main drawback is the relative storage space requirement, but this inefficiency is not problematic when the store is very large [Fre60].

To illustrate this data structure consider, again, the previous example in Figure 3.2 and the output provide by MINION:

```
1. {t1, t4}
2. {t1, t2, t4}
3. {t1, t3, t4}
4. {t2, t3, t4}
5. {t1, t2, t3, t4}
```

First TRIE structure starts with an empty node and then the first set, {t1, t2} is inserted (see Figure 3.5).

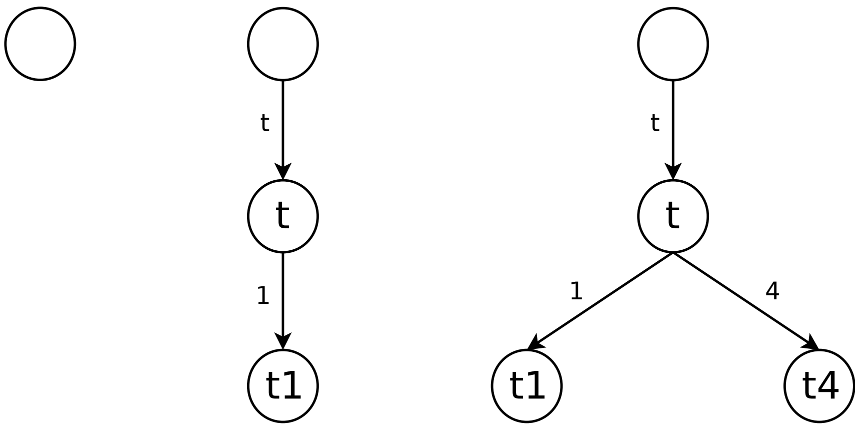


Figure 3.5: TRIE Example: empty TRIE (left), added the string "t1" (center), and the final TRIE, after added the string "t4" (right)

The set {t1, t2} can be inserted with success, because elements "t1" and "t4" do not exist in TRIE structure. But if trying to insert the others sets {t1, t2, t4}, {t1, t3, t4}, or {t1, t2, t3, t4} it cannot do that, because all these sets have at least one element that already exist in TRIE structure. So, with TRIE structure it can be remove non minimal sets, that have other sets inside. Using TRIE, the solution for MSC of example described in Figure 3.2 are {t1, t2} and {t2, t3, t4}.

3.5 Complexity Analysis

To measure the complexity of our approach we show the time and space complexity.

Time Complexity

To find a MSC, all test cases need to be executed in order to obtain the coverage matrix. Without loss of generality, test cases are assumed to take the same amount of time to execute. Thus, the time complexity for this phase is $O(|T|)$ where $|T|$ is the cardinality of the test suite (i.e., the number of test cases). Second, the matrix has to be mapped into constraints, representing a time complexity of $O(N)$. Third, MINION is executed to compute the solutions for the problem at hand $O(MINION)$. It uses a depth-first search method, in general it will take worst-case exponential time $O(2^N)$, although many problems are in practice solved in polynomial time $O(n^2)$. Fourth, the results are filtered so that only minimal solutions are kept $O(|D|)$ where $|D|$ is the number of solutions spit by MINION.

All in all, the time complexity of our approach is $O(|T| + N + MINION + |D|)$. It is reasonable to assume that $|T|$ and $|D|$ are of the same order of magnitude as N. Thus, time complexity is $O(3N + MINION)$. As the number of tests is smaller than the time complexity, the overall time complexity of our approach is $O(MINION)$.

Theoretically, $O(MINION)$ is a worst-case exponential time $O(2^N)$, but in practice that complexity does not affect the global performance (see Section 5.1.3).

Space Complexity

With respect to space complexity, for each invocation of our approach, the complexity is $O(M \times N)$ to keep the coverage matrix and, if D is the number of solutions, $O(|D|)$ to retain all solutions. Therefore, the total space complexity is $O((M \times N) + |D|)$.

3.6 Summary

Present Chapter has introduced a new approach to test suite minimization, selection and prioritization using MSC problem. Is based in a coverage matrix that is used to mapping the relation of components with test cases as a set of constraints. This approach can minimize and select tests, creating minimum sets which coverage the same percentage of components, that original set. After minimization, several metrics of prioritization can be applied, e.g., order minimum sets through cardinality.

Chapter 4

Toolset

The first goal of this project was to develop an ecosystem where developers can re-test and debug their software, with new latest studies in the field of regression testing and debugging software. So, we create and integrate Regression-Zoltar (RZOLTAR) in GZOLTAR Project. And, such as GZOLTAR, RZOLTAR should be used in Eclipse IDE to management the test cases.

RZOLTAR should provide several approaches for techniques of regression testing: minimization, selection, prioritization; which can reduce the effort time of re-test the software. Users should be able to choose from a list of possible sets, the preferential minimal set that coverage all project under analysis with the same coverage of original set (with all test cases).

4.1 Process Components

RZOLTAR components can be divided into five main areas (see Figure 4.1):

- Initial Eclipse Integration;
- JUnit and JaCoCo;
- MINION;
- TRIE;
- Visualization and Final Eclipse Integration.

Initial, Eclipse Integration, allows the detection of all open projects, classes and test classes. JUnit and JaCoCo, executes test cases and produce code coverage information, to create the coverage matrix. MINION, read coverage matrix and try to resolve the CSP. TRIE, can filter the results provided by MINION, to eliminate redundant results. At the end, Visualization and Final Eclipse Integration, display the several minimal sets, and integrates into default Eclipse code editors. For a detailed diagram about process components, see Figure 4.2.

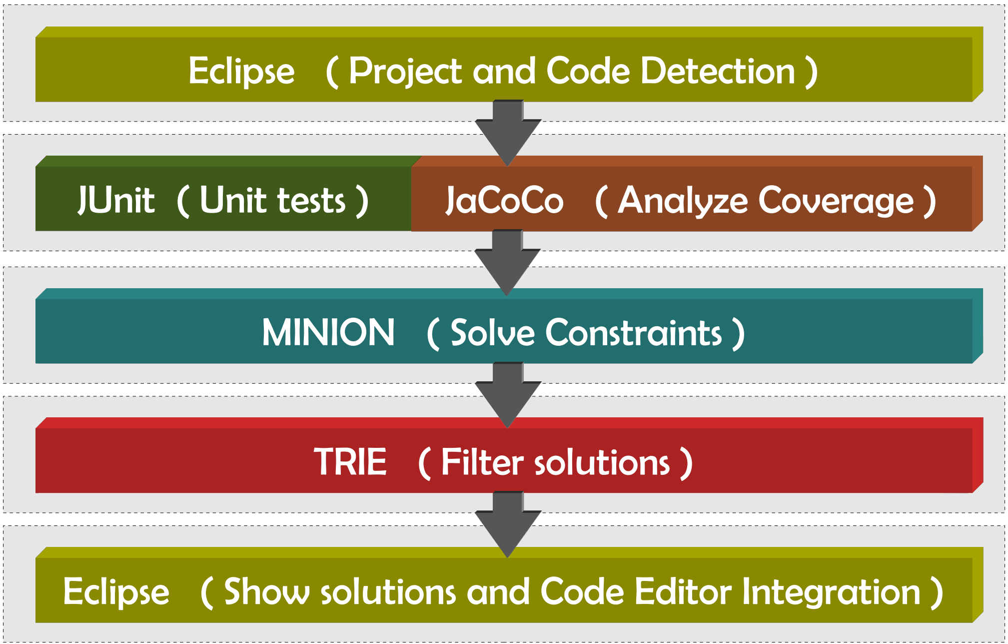


Figure 4.1: **RZOLTAR Brief Process Flow.** RZOLTAR integrates well into Eclipse. It detects its projects, run unit test and save the coverage of them, solve constraints, creates a Eclipse view with regression sets and integrates with code editor.

4.1.1 Initial Eclipse Integration

Eclipse is a well known open source project of The Eclipse Foundation [Fou11]. Eclipse most known application is its IDE. This tool is used by many programmers around the world, and is supported by many major software companies [Gee05]. It allows software development in many programming languages, even though its native programming language is Java.

On Eclipse, it is possible to automatically detect all open projects in Eclipse IDE, so GZOLTAR plug-in can then work with them. After having a list of the open projects, GZOLTAR search them for all its packages, java files, classes and methods; thus classes that have JUnit syntax has marked such as 'Test' class to be execute later. To avoid differences between the source files and the compiled classes, at this stage it is built the entire project, to guarantee that GZOLTAR has the latest version of the code. This is essential to the correct work of code coverage, as explained further in more detail. After having all information about all open projects, and knowing which one are test classes, GZOLTAR plug-in is ready to start the next stage.

4.1.2 JUnit & JaCoCo

For each project there is a list of all test classes. For each test class all the code is instrumented, to allow the code coverage process. That process aims to detect if a given line of code was or not executed, during an execution. All open projects are instrumented, because open projects can call methods from other projects. After that, test classes are executed. Test classes are Java classes that implement JUnit syntax.

Returned value will be stored at test class respectively, to be used later by the RZOLTAR View, to show if test passed, failed or happened an error. Then code coverage results are analyzed. Test classes are ignored during code coverage analysis, because they are not important to calculate the regression set. At this stage is possible to store the Coverage Matrix line about this test (see Section 3.1). Because the project was automatically built at the time of project and class detection, it is guaranteed that the code coverage made against compiled classes corresponds exactly to the source code files. This is essential to avoid errors in code analysis. If this procedure was not considered, code coverage information could point to the wrong line of code.

JaCoCo [Hof11a], a recent project from the EclEmma [Hof11b] project team was used. JaCoCo can instrument Java code, execute it and analyze each line of code to check if it was executed or not. EclEmma is based on Emma [Rou11], a discontinued project. Because of that, EclEmma project team started a brand new project to create Java Code Coverage. This project is JaCoCo, which started in mid of 2009. JaCoCo was been presented on Eclipse Summit Europe 2010 [Hof11c] and now has totally integrated in EclEmma. Java instrumentation is much more complex than a native application one, because Java code coverage tools have to deal with Java Virtual Machine (VM). JaCoCo is an Application Programming Interface (API) that allow the integration of Java code coverage functionalities on GZOLTAR.

Performing a code coverage analysis with JaCoCo is based on three steps:

- Instrumentation of the code to be analyzed;
- Execution of that code;
- Code coverage data analysis.

RZOLTAR does exactly those three steps. All code from open projects are instrumented, test classes are executed and code coverage data is analyzed. At the end, the Coverage Matrix is built.

4.1.3 MINION

After collecting the coverage of SUT, the coverage matrix can be converted into constrains (see Section 3.3) and passed to MINION constraint solver. The MINION project is distributed for the most used operating systems (Microsoft Windows, Mac OS X and also Linux Systems), so the call to MINION is executed in other process with the following parameters:

- No print solutions, `-noprintsols`;
- Sometimes the user choose (in interface) to calculate all solutions, `-findallsols`, sometimes he choose limited solution, `-sollimit`;
- Redirect the solution output to a specific file: `-solsout`.

4.1.4 TRIE

The results provided by MINION are filtered by TRIE structure to delete the non minimal sets.

4.1.5 Final Eclipse Integration

While analyzing the several minimal sets, on the RZOLTAR visualization, user can:

1. see the result of test (pass, fail or error) by the color of icon. If test fail or return an error, user can check the message of test result at “Trace” window;

2. double-click on any test case, and jump directly to that test class (an Eclipse code editor is opened, and the text cursor placed on the selected test);

If user correct any failed test, he just need to refresh RZOLTAR (by pressing `CTRL + F5`, for more details see Section 4.4.) to have an updated version of the system, after the last fault fix. This way, user can perform regression testing tasks (run only minimal sets to test SUT and fix existing faults) at the same place, with the advantage of being a well known tool (Eclipse IDE).

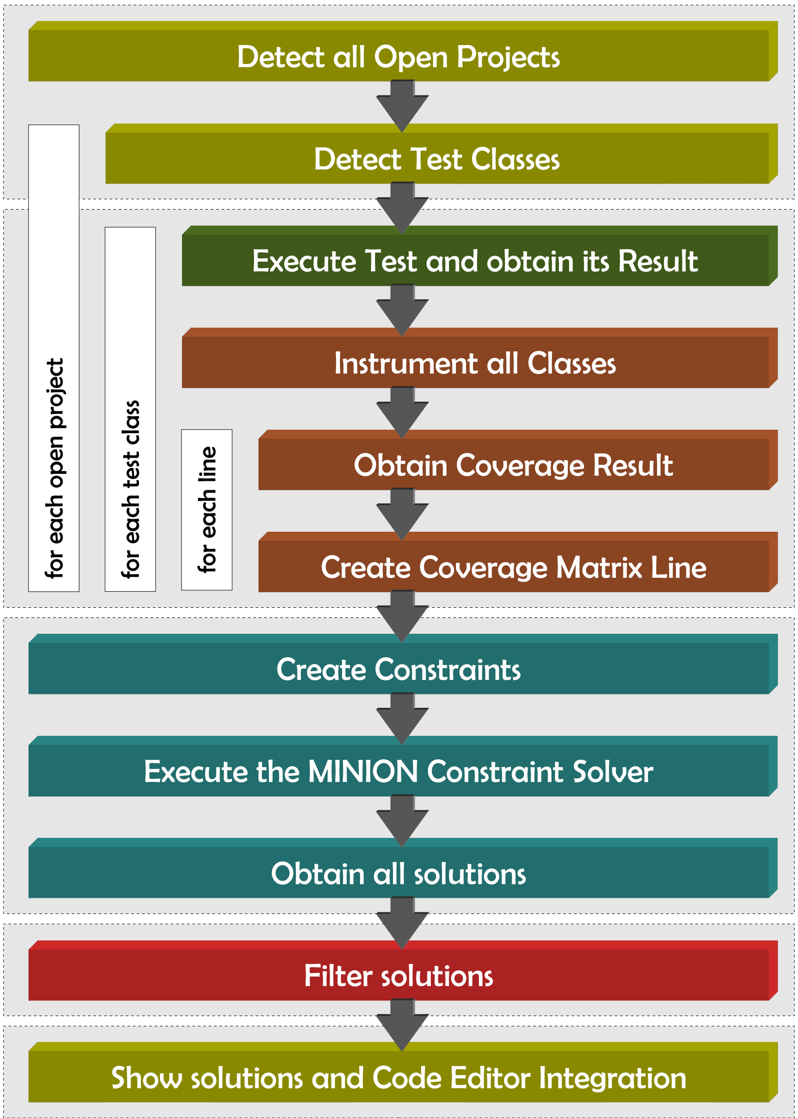


Figure 4.2: **RZoltar Detailed Process Flow.** All major tasks from RZOLTAR start until show solutions and code editor integration.

4.2 Logical Layers

RZOLTAR uses many different technologies to accomplish its tasks. It is written in Java and is an Eclipse View integrated in GZOLTAR plug-in. To obtain information about open projects, their classes, it uses Eclipse's Workspace component. RZOLTAR uses JaCoCo, a Java-based API that offers code coverage capabilities and JUnit to execute unit tests. To produce Eclipse User Interface (UI) related tasks, it uses Eclipse's Workbench component. This component has SWT, that allows the creation of RZOLTAR view. In RZOLTAR, but written in C is the MINION constraint solver. TRIE structure has a interface written in Java and implementation written also in C. For a schematic view of these technological layers, see Figure 4.3.

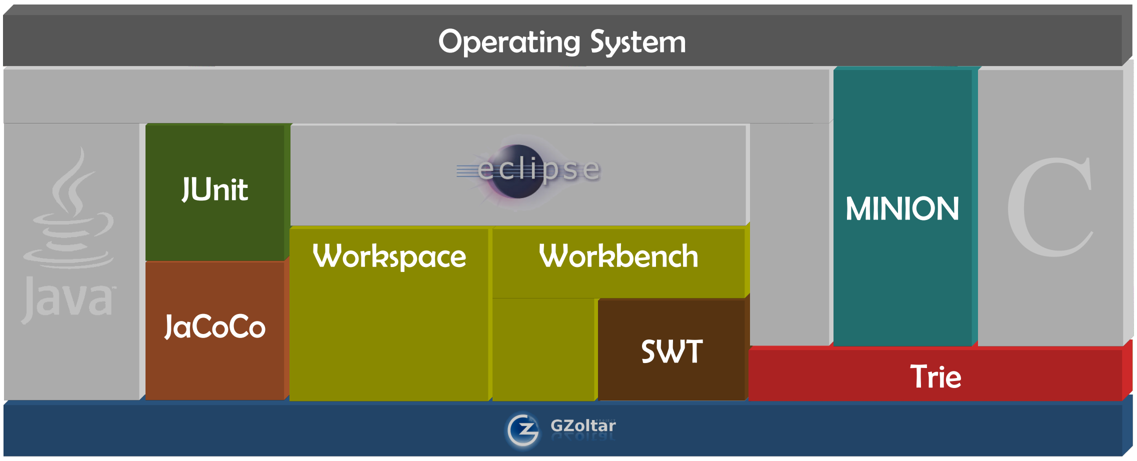


Figure 4.3: **RZOLTAR Layers.** Integration between RZOLTAR and other technologies.

4.3 Modular Architecture

GZOLTAR was developed in a modular way, allowing a faster and organized development in the future. It has seven packages, that work much like service providers between them. In this Section is presented the way GZOLTAR code is encapsulated (for a conceptual representation of GZOLTAR architecture, see Figure 4.4).

4.3.1 Algorithms Package

The main goal of GZOLTAR was to develop a tool that could provide a graphical view of the project under analysis, clearly indicating the fault probability of each module. To do this, the authors of GZOLTAR plug-in implement the algorithm Ochiai. Ochiai, is a statistics-based algorithm for software fault diagnosis. It takes as input component involvement in the execution of a given test case and whether or not the test case has passed. The diagnostic report yielded by Ochiai is a ranked list of components likely to be responsible for observed failures [AZvG07]. So, when "Zoltar" class needs to calculate the software faults, calls "Ochiai" class. New algorithms, for fault localization, developed/implemented in the future should be placed at this package.

4.3.2 CSP Package

The constraint solver used in RZOLTAR plug-in is MINION, so the “Minion” is responsible to interact with that constraint solver: create the input (transform coverage matrix in constraints), call MINION executable and catch the output (result). Similar to “Algorithms Package”, new implementation or calls to others constraint solver should be placed at this package.

4.3.3 Plugin Package

When “GZOLTAR View” or “RZOLTAR View” start, the class “Activator” present in this Package is called. This class has the minimum required code to start-up Eclipse Plug-in. Like “Activator” class, the classes GZOLTAR and RZOLTAR only executes on start-up and only has the minimum required code to start GZOLTAR View or RZOLTAR View.

Two different visualizations were created for “GZOLTAR View”: “Sunburst” and “Treemap”. This visualizations have just a minimal amount of code, because they are focused on data visualization only. They have access to the tree with all needed data, provided by the main “GZOLTAR” class, and to all the needed auxiliary processing is provided by the two classes in the “Utils” package. New visualizations that could be developed in the future should also be placed at this package.

The class “RZOLTAR” prepare the all environment to “RZOLTAR View”, such as copy MINION executable to the temporary folder.

4.3.4 Utils Package

Plug-in also has a special package, called “Utils”, that has many useful services that can be used by the all classes. It has two classes: a “FileUtils”, that provides services related with functions of read or write and delete files, and a “Markers”, that provides services more oriented to generate markers (warnings) into Eclipse code editor. These classes work much like service providers, that offers services to the two main classes, “GZOLTAR” or “RZOLTAR” class, and to all other classes.

4.3.5 Views Package

The two views of GZOLTAR plug-in are: “GZOLTAR View” and “RZOLTAR View”. These views interact directly with user, “GZOLTAR View” use Open Graphics Library (OpenGL) technology embedded in an Eclipse View to create all the 3D scenes of the different visualizations, “RZOLTAR View” create a simple Eclipse View with regression sets and a few buttons which can be used to initiate the plug-in, parametrize some options, etc (for more details see Section 4.4.).

4.3.6 Workspace Package

To the save the coverage of all open projects in workspace, GZOLTAR plug-in saves important information of workspace, information such as: name of all open projects, paths to the sources directories and output directory, the name of all packages, sources files, classes, etc. With this information "Zoltar" class can know how many unit tests that has to execute, register the coverage in respectively line. "RZOLTAR View" also can use this information to integrate view with Eclipse code editor.

4.3.7 Zoltar Package

During the utilization of GZOLTAR View or RZOLTAR View, when the user presses `CTRL + F5` key (for more details see Section 4.4.), the class "Zoltar" is invoked from this package. This class has the code to process all automatic debugging results, to be used by the different visualizations. This main "Zoltar" class, that makes all the processing, it also has a second one, "ZoltarTree" that has the model to store the processed tree structured data.

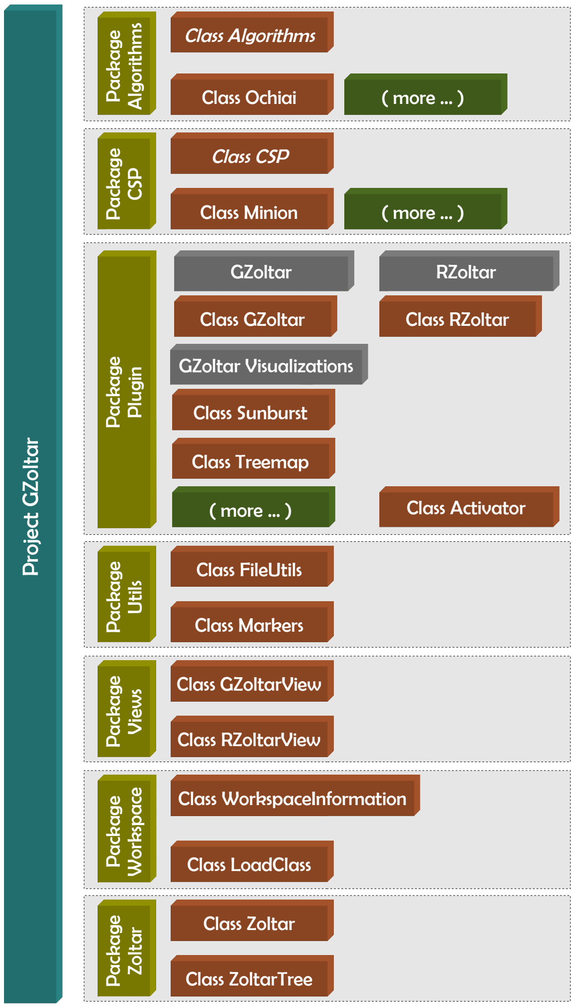


Figure 4.4: **GZOLTAR Modules.** GZOLTAR Project is compound by seven packages. "Algorithms" has algorithms to calculate the failure probability of each component. "CSP" has interfaces to classes which trying to resolve the constraint satisfaction problem. "Plugin" has main Eclipse bundle, "GZOLTAR", "RZOLTAR" class, and the main class "Activator". "Utils" has several auxiliary classes. "Views" has "RZOLTAR" and "GZOLTAR" tab views. "Workspace" has information about *workspace* from Eclipse, and "Zoltar" has automatic debugging classes.

4.4 Eclipse View

RZOLTAR is compatible with Microsoft Windows, Mac OS X and also Linux Systems (see Fig. 4.5). It is also compatible with 32 and 64 bit Central Processing Unit (CPU) architectures.

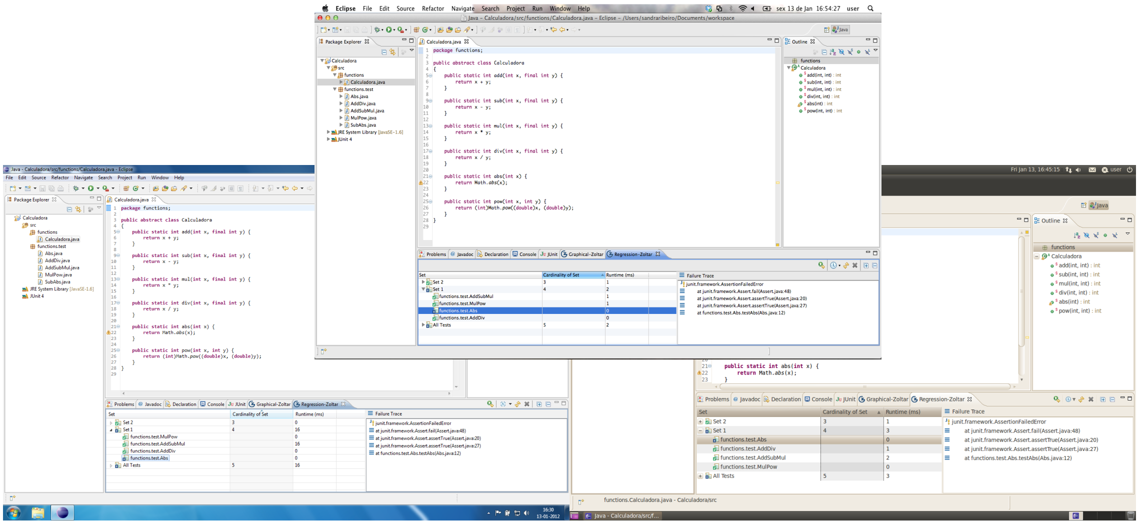


Figure 4.5: **RZOLTAR is Multi-Platform.** RZOLTAR can be installed and used without any limitation in many different systems.

By default, RZOLTAR presents an Eclipse View integrated into the IDE, such as many other views, like "Console" or "Problems" (see Fig. 4.6). It is possible however to expand RZOLTAR view by double-clicking on its view tab (with "Regression-Zoltar" label). This way RZOLTAR viewable area gets bigger, but all the other views gets hidden.

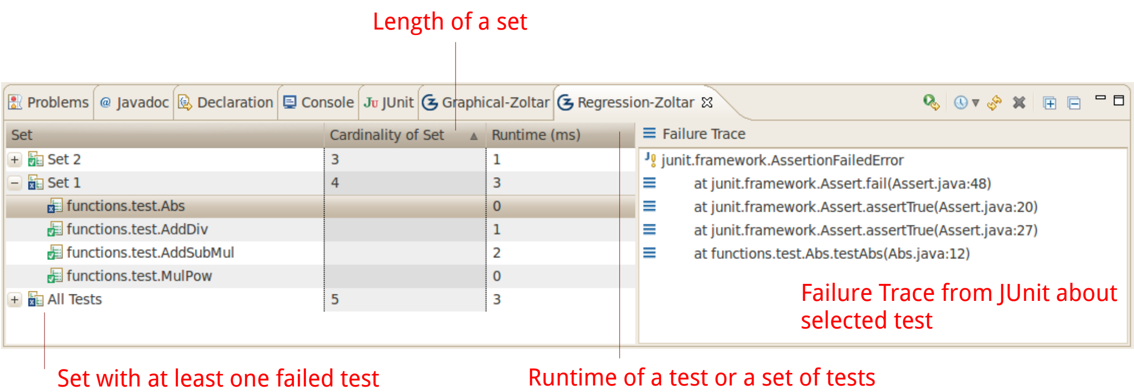


Figure 4.6: **RZOLTAR View in Eclipse IDE.** This is the default RZOLTAR View placement.

RZOLTAR View is responsible for all actions about regression testing. This view is divided in two layers, on the left user can access to the list of minimum set coverage (including the set with all test cases); on the right user can see "Failure Trace" of faulty tests. At two layers, user can always double-click on a test case and jump to the file with test case, or at failure trace, double-click goes to line (presents in that layer).

At RZOLTAR view there are two implicit prioritizations: Cardinality of Set and Runtime. First prioritization can order the minimum sets through the number of tests in every sets. Second prioritization can sort the minimum sets by the total of time which set needed to re-run.

Every time you change your project code, RZOLTAR have to update its information. This is not made automatically because of performance reasons. To do so, click on "Refresh" button (`CTRL + F5`) or if you already do this, you can only select a set of tests and click on Re-Run, to run again the selected set. You will then see a view with the updated data. If you are working with big projects, it is normal if you have a delay between the time you press "Refresh" button and have the view ready to navigate. User can always clear, expand and collapse results, with respectively buttons (see Figure 4.7).

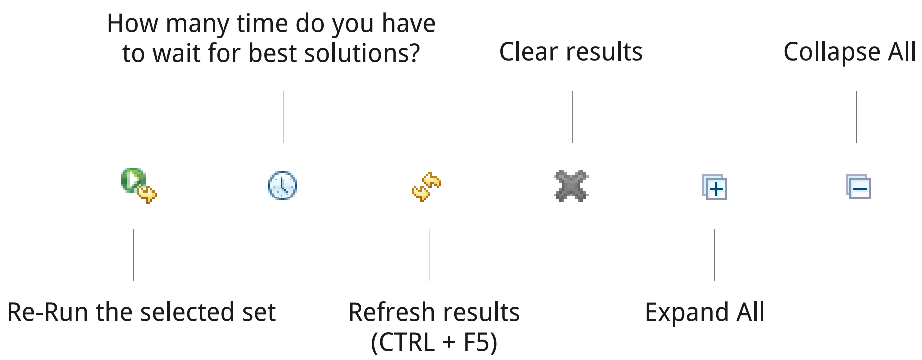


Figure 4.7: RZOLTAR Toolbar.

From the left to the right, in Figure 4.7 appears: 'Re-Run', 'Time/Solutions', 'Refresh', 'Clear Results', 'Expand' and 'Collapse'. Re-run allows the re-execution of a selected set. Time/Solutions provides some options to user, related with time to calculate the minimum set coverage, for example, "I want to wait 30sec (max limit) for a minimum solution", or "I want the first 100 minimum solutions". Refresh, runs all unit tests presented in Java project, and calculate the minimum set coverage. Clear results, erase the RZOLTAR view. Expand and Collapse, show/hide the sets in RZOLTAR view.

4.5 Summary

This chapter presented the system's implementation of the proposed architecture for RZOLTAR. It can be concluded that integration Regression Testing with GZOLTAR, dubbed RZOLTAR, is working as expected. The interaction with user is very intuitive (more information in next chapter with user studies), because it was used familiar icons and colors from Eclipse themes.

Chapter 5

Evaluation

This chapter presents an evaluation of the results achieved by two evaluations: empirical and with user studies. In order to empirically evaluate the performance and results the approach presented in Chapter 3 three open-source programs were used. The goals of this evaluation were to measure the percent reduction in test suite achieved by RZOLTAR, and to measure the analysis time to achieve the test set reduction. To evaluate the usability of GZOLTAR, an open-source program was used. The study give to users some minutes to find a fault previously injected in source-code.

5.1 Empirical Evaluation

To asses the performance of our approach, we performed an empirical evaluation using several software subjects. In our evaluation, we investigate the following research question:

RQ1: Can RZOLTAR find an optimal solution for a test suite minimization problem in a reasonable time?

Sections 5.1.1 and 5.1.2 presents the software subjects that we used in the study and our experimental setup. Section 5.1.3 illustrates and discusses our experimental results.

Subject	Description	Version	Classes	Unit Tests	LOCs	Coverage
NanoXML	XML parser	2.2.3	32	9	5483	52%
org.jacoco.report	Report generation	0.5.5	93	225	5973	97.2%
JTopas	Text data parser	0.8	64	57	9037	84.3%

Table 5.1: Subject program used in the empirical study

5.1.1 Experimental Subjects

Three subjects written in Java are used in the empirical study. NanoXML is a is a small XML parser for Java [Sch12]. org.jacoco.report is a module of JaCoCo project that has utilities for report generation [Hof11a]. JTopas is a Java library used for parsing text data [Bla12].

Table 5.1 provides summary information about the subjects programs. The first object have Test Specification Language (TSL) suite [OB88], all tests are defined by a input/output file which contains the input for every test and respectively output expected. The last two have JUnit test suites. For each program, the table shows a description, the official version that we considered, the number of classes, the number of JUnit test classes, the number of JUnit test methods, the number of lines of code (non-comment lines), and the percentage of coverage. To gather that coverage data, we used an Eclipse plug-in, Metrics[1], to perform coverage analysis.

5.1.2 Experimental Setup

All subjects, except NanoXML have unit tests written in JUnit. For NanoXML, we have create several test cases in JUnit with the inputs from test files (defined in NanoXML project) and insert an assert to check if the output of the test case is equal to the output from output file, as expected. With this we still maintaining the same purpose and inputs/outputs of tests.

For NanoXML, we have create a several test cases, which is the mapping of TSL tests into JUnit tests. We create a test with the inputs from test files (defined in NanoXML project) and insert an assert to check if the output of the test case is equal to the output from output file, as expected. So, we can ensure the same purpose of test in TSL and JUnit. To org.jacoco.report and JTopas, we convert all JUnit test cases, in real unit tests. These two subjects have a lot of test case which have at least two or three unit tests. So, to check the real purpose of every unit test, we transfer every unit test presenting in test case to a single test case (e.g. if a test case has 3 unit test (u1, u2 and u3), we create test case c1 with u1, c2 with u2, and c3 with u3, with all needed definitions). This transformation is valid and legitimate, because does not change source code, or increase/decrease percentage of coverage or number of tests. It has necessary to do that because RZOLTAR, has a technological limitation. It does not handle the individual tests in a suite per set, but considers each suite as one test.

For three subjects ten experiences are made. For each one we measured how long RZOLTAR took to compute a solution and the cardinality of solutions (minimum sets) founded. We also calculate the average and the standard deviation taken by our approach (refer to table Table 5.5 for further information).

For our experiments, we ran all subjects on Linux, on a 2.27 Ghz Intel Core i3-350M with 4 GB of RAM running Debian Gnu/Linux Wheezy.

[1]Metrics Homepage `http://metrics.sourceforge.net/`, 2012.

5.1.3 Results and Discussion

Experimentation shows that for large programs a significant reduction in the test suite may be achieved. The reduction for our examples ranged from 11% to 75%, in 0.1 seconds and 2 seconds, respectively.

RQ1 Can RZOLTAR find an optimal solution for a test suite minimization problem in a reasonable time?

To answer RQ1, we first analyzed the data collected in our experiments (see Appendix C for more details).

By looking at the data in Table 5.2 we found that RZOLTAR can compute, at least, a solution for all of the cases considered. The reduction for subjects ranged from 11.11% in case of NanoXML, to a significant result of 75.56% in org.jacoco.report subject.

Subject	**#Original Suite**	**#Minimized Suite**	**% of reduction**
NanoXML	9	8	11.11%
org.jacoco.report	225	55	75.56%
JTopas	57	26	54.39%

Table 5.2: Percentage of size reduction. The #Minimized Suite represent the smallest set found.

In NanoXML and JTopas subjects, RZOLTAR was able to return more than one minimum set coverage. For NanoXML it returns 2 sets, each set with 8 tests. As we can see in Table 5.3, for JTopas, RZOLTAR returns several minimum sets (all with the same number of tests, 26), dependently the parameter defined in 'Time/Solutions' button (see Section 4.4).

Parameter	**Number of Minimum Sets**
1 second	6
30 seconds	12
60 seconds	12
first 100 solutions	2
first 1000 solutions	4

Table 5.3: Number of minimum sets for JTopas subject

Next, we measure how long it took for RZOLTAR to compute one solution, Table 5.4 shows the results of this analysis.

Subject	⟨MINION⟩	⟨Trie⟩	⟨MINION + Trie⟩	σ
NanoXML	0.106	0.001	0.107	0.013
org.jacoco.report	0.429	0.002	0.430	0.014
JTopas	1.198	1.016	2.214	0.019

Table 5.4: Average and standard deviation of time (in seconds) execution of MINION + Trie

The results in Table 5.4 show that RZOLTAR was able to find an optimal solution for all subjects in 1 second, and in most cases the solution was computed in less than 0.5 seconds. And like we said in Chapter 3, MINION is a really fast constraint solver, e.g. in JTopas subject, with 9037 constraints (number of Lines of Code (LOC)s) and 57 variables, MINION can return six minimum sets in 1.198 seconds. The Trie usage time is also rather satisfactory. In NanoXML and org.jacaco.report the time spent in the Trie is marginal. In JTopas the time spent there is about 1 second.

Overall, when reducing the size of the original test suite, one reduces the time needed to achieve the same coverage. Table 5.5 presents the time needed to execute the original set, as well as the reduced set (minimum set) proposed by RZOLTAR.

Subject	⟨Original Set⟩	σ	⟨Minimum Set⟩	σ	% of reduction
NanoXML	1.976	0.064	1.723	0.077	12.81%
org.jacoco.report	39.004	0.484	10.671	0.197	72.64%
JTopas	275.673	0.586	157.717	0.480	42.80%

Table 5.5: Time (in seconds) reduction from original set to minimum set

Similar to the results reported for the cardinality, the time reduction for minimum sets have the same range of values, e.g., for org.jacoco.report the percentage of time reduction is about 70%, like percentage of size reduction.

5.2 User Study

In order to validate the usefulness of the current version of the plug-in, nine users were selected to test the efficiency of the interactive visualizations. It was recorded the time that each user took to finish the testing and debugging task. At the end of this process, each user filled a form (see Apendix D) with the feedback of their experience and some suggestions for future work. This usability test was important not only to test the efficiency of the presented plug-in but also to aid the development team to fulfill the user's needs in future versions of this tool.

5.2.1 Users Description

Nine developers composed the users group. The users were picked randomly from the Department of Informatics Engineering of the Faculty of Engineering at the University of Porto, Department of Informatics of the Faculty of Science at the University of Lisbon and Universidad Complutense of Madrid, Spain. The number of users was based on J. Nielsen's work related with usability and user tests [NL93]. Nielsen advocates that for a small software project, nine is in the optimal range of users to participate in the usability test [NL93]. This small number of users should be enough to identify the main usability issues. This experiment was conducted to identify the main users' difficulties while using the GZOLTAR plug-in (RZOLTAR View and GZOLTAR View). This information was helpful to create guidelines to improve future developments of this tool, and to have a first assessment of the impact of this plug-in among the users.

The user group was composed by MSc and PhD students and professors in Informatics Engineering, aged 22 to 27 years old, and from both genders. The users were familiarized with three main operating systems: Linux (89%), Microsoft Windows (78%) and Apple Mac OS X (33%). The most common programming languages used by the group members were Java (100%), C (100%), C++ (89%), and PHP (89%), C# (67%), Python (67%) and Assembly (56%). The majority of the developers used regularly an IDE, being the most popular IDEs Eclipse (89%), NetBeans (22%) and Microsoft Visual Studio (67%). There was however one user who did not use regularly any IDE (uses VIM to develop). The most used debugging techniques were breakpoints (89%) and "print" statements in the code (78%). JUnit is the main testing tool used by the group (78%). On the other hand, two elements of the group stated that they do not use any particular testing technique.

5.2.2 Experiment Conditions

Computers with Linux and the Eclipse IDE with the GZOLTAR plug-in were used for this user study. Each user was asked to debug a faulty version of the NanoXML v2.2.3 application [Sch12]. This is an open source, medium size application with 5483 lines of code and a suite of JUnit tests. A fault was injected in the class `XMLUtil`, which can be found in the `net.n3.nanoxml` package. Line 109 (method `skipTag`) was modified from "`case '>':`' to '`case ']':`', making some test cases fail. Users had no previous contact with the application source code and the JUnit tests. A brief explanation (about 5 minutes) was given to each user to explain the assignment and the workflow of GZOLTAR. The users were given 20 minutes to test and localize the fault. After the debugging task, each user filled a survey with questions on their experience.

5.2.3 Results and Feedback

As mentioned before, the time limit for this task was very short - only 20 minutes. The goal was not to record how long the users would take to find and fix the faults, but to obtain feedback about the plug-in usability and usefulness by a set of independent users. It is important to note that

71.4% of the users were able to find the fault in less than 20 minutes (and 42.9% even fixed the fault to ascertain that the suspicion was justified). It is important to highlight that the users did not know which application was going to be used in the experiment, and they did not have any previous knowledge about the source code. From the users that were not able to find and correct the fault, 50% were able to point the most likely fault localization. However, because they were not able to fix the fault, they could not confirm that the fault localization was right. It is important to note that some users were rather uncomfortable with the Eclipse IDE because they never used it or they did not use it on a regular basis.

The survey had a section where the users answered questions related with their profile and development experience, and a section where the users could give their feedback. Replies to the questions about the plug-in interface, performance and embedded concepts were made using a scale from 1 (unacceptable) to 5 (excellent).

A considerable amount of users (44%) found the plug-in difficult to understand at the first. However, the majority (56%) stated that they understood how the tool works in a short period of time. The debugging tasks, using GZOLTAR plug-in, were considered fast and logic by 78% of the users.

The users group also analyzed the performance of the plug-in. An expressive slice of the group (77%) considered the responsiveness of the plug-in as very good. The same number of users (77%) found that the plug-in usefulness increased with their experience and knowledge about the tool. Most importantly, all the users considered that they were able to obtain good results with little knowledge about the tool operation.

The users group also gave feedback about the concepts embedded in GZOLTAR. All the users considered automatic debugging as an important concept, where 67% classified it as "essential". Debugging techniques integrated into IDE were also considered important, having the majority of users (56%) considering them as "essential". A large number of users (78%) also considered visual debugging as an important concept.

Also, users report the global experience using RZOLTAR and GZOLTAR View. To RZOLTAR View, users (66%) classified it as easy to use. For GZOLTAR View, they consider (44%) their experience median.

The final part of the survey had an open question where the users could leave their comments and suggestions. Some suggestions were related with the colors. Some users found that the full-color spectrum affected negatively the visualization analysis. They suggested the limitation of the number of colors (having for example color red for "high probability", yellow for "low probability" and green to "no probability"). The users' comments were very positive. Two users stated that without the GZOLTAR plug-in, they would probably never find the software faults, because they did not know the software they were testing.

This experiment with developers validated our hypothesis. An interactive visualization of automatic debugging reports can help developers to find the faults localizations in a short period of time. Moreover, an IDE plug-in facilitates not only the faults localization but also the fixing of the localized faults. Even not knowing the faulty software, most of the participants were able to find and fix the faults in less than 20 minutes.

5.3 Summary

In summary, RZOLTAR provides a practical and efficient approach to determine the minimum set coverage for any program with JUnit tests.

The performance of our approach was evaluated with several experiences, using three open-source programs: NanoXML, org.jacoco.report and JTopas. Empirical evaluation show that approach can handle large program without sacrifice performance, e.g. for JTopas with 9037 constraints and 57 variables, minimum set coverage can be determined in 2 seconds. In general the results were much satisfactory and in some case the percentage of reduction the original set, was about 75% and save 43% of time to re-execute the minimum set.

Relative to the interface and interaction of user with GZOLTAR, the effectiveness of tool was assessed with an usability test, performed with a small group of informatics engineering MSc and PhD students. The results of this study were very positive, and shown that users found GZOLTAR to be a powerful testing and debugging tool. The study was also useful to get users feedback to aid in future developments of this tool.

Chapter 6

Conclusions and Future Work

In this manuscript, we started by looking at the current issues of the software testing field, specifically about regression testing: (1) time costs associated to re-test a software when some new features are developed or some old features are removed, and (2) tools to support testers to make the regression testing a simple and quickly task. Behind this status we provide a study about several techniques for regression testing: minimization, selection and prioritization. To prove the veracity of our approach a tool was developed to make the regression testing task as more easiest as possible.

6.1 Work Summary

Our first contribution was in the field of test suite minimization. The problem in identifying what tests will be chosen to be re-run was mapped to a CSP and solved using the state-of-the-art solver MINION [GJM06]. After identifying the minimal subsets of tests to run, we can select any set, because we have confidence that all requirements have been satisfied. In terms of prioritization, the subsets may be prioritized (ordered) through two metrics: cardinality of the set and runtime.

Finally, we created an ecosystem in GZOLTAR, where developers can use features such as Debugging & Testing in the same tool.

6.2 Future Work

The initial goals were all achieved, even much more new goals that came during RZOLTAR development were achieved. There are always new ideas, on an almost daily basis, that could improve RZOLTAR project. Even during the case studies, when RZOLTAR plug-in was considered ready, there were new good ideas to improve RZOLTAR behavior.

6.2.1 Convert Test Suite into Test Case

RZOLTAR try to find the minimum set coverage in some Java project, but if exist a Test Suite which invoke all Test Cases presented in project, RZOLTAR select that test suite to be the minimum set coverage. This happen because test suite execute all test case, which covers all program, so test suite cover all program, too, with the same percentage. Converting automatically test suite into several test cases can be interesting to implement in RZOLTAR, because could be more efficient to determine the minimum set coverage.

6.2.2 Improve performance of RZOLTAR

There is an unofficial version of RZOLTAR which use Ant[1] to execute and register the coverage for every unit tests. But that version it is less efficient (in time) then official version. So, in future that should be explored to improve the unofficial version in some way, to a perfect performance.

6.2.3 New Techniques for Prioritization

Prioritization by cardinality or runtime execution of set are two powerful techniques, although new prioritization should be implemented in future, e.g. ranking minimum sets by number of faults that can detect, bayesian probabilities, etc.

6.3 Side-Work

During the study/development of this project, I had the opportunity to participate in an summer internship and a competition, which gave me a totally unique experience.

6.3.1 Summer Trainee at Critical Software[2]

For two months I make a summer internship at Critical Software. This experience gave me the opportunity to improve my technical skills, as well personal/interpersonal.

In the technical field, I had contact with technologies which hitherto had not used, e.g., Concurrent Versions System (CVS), Jira - a project tracking tool, Bugzilla, Wise, and a development environment completely unknown for me. The main goal of the project in which I was involved, was to validate the requirements of control system present in SWARM project. SWARM is a trio of satellites for a unique view inside the Earth The Swarm mission will provide the most detailed data yet on the geomagnetic field of the Earth and its temporal evolution, giving new insights into improving our knowledge of the Earth's interior and climate.

At level personal/interpersonal, the experience allowed to improve my way to interact with complete strangers as well as the integration on teams that was already formed and with several routines.

[1] Apache Homepage http://ant.apache.org/, 2012.

[2] Critical Software Homepage http://www.criticalsoftware.com/, 2012.

6.3.2 DXC'11

NASA Ames Research Center (ARC), Palo Alto Research Center (PARC), and Delft University of Technology decided to combine efforts to create a generalist framework that would establish a typically platform to evaluate and compare diagnosis algorithms [KNP+09]. The goals for developing this framework are to increase research in theories, principles, and computational techniques for monitoring and diagnosis of complex systems.

The First International Diagnostic Competition happens in 2009 and was the first implementation of the above refered framework [KNP+09]. The Third International Diagnostic Competition (DXC'11)[3] includes the same diagnostic problems (present in other past editions), and use cases for the industrial and synthetic tracks as DXC'10. In 2011 the competition implement a new feature, a software track. The objective of this track is to provide typically evaluation techniques that diagnose failures in software systems. In particular, the programs used in this track were taken from both the Siemens and Software Infrastructure Repository (SIR) benchmark programs.

Therefore, we decided to submit the Ochiai algorithm in this year's competition (DXC'11). Ochiai, is a statistics-based algorithm for software fault diagnosis. It takes as input component involvement in the execution of a given test case and if the test case passed or not. The diagnostic report produced by Ochiai is a ranked list of components likely to be responsible for observed failures [AZvG07].

The averaged metrics for each of the three Diagnosis Algorithm (DA) (Ochiai, Raptor-H and Raptor-EPS2) that participate on this year's competition are represented in Table 6.1 [SDH11].

	Metrics		
DA	M_{probe}	M_{cd}	M_{mem}(kb)
Ochiai	109200	1138	**3114**
Raptor-H	26413	**862**	8804
Raptor-EPS2	**11056**	889	10320

Table 6.1: Diagnostic problem IV metrics [SDH11]. M_{probe} represent the probing cost metric, M_{cd} represent the residual diagnostic effort, and M_{mem} represent the memory load.

Both Raptor-H and Raptor-EPS2 have much inferior probing costs than Ochiai. Denote that this is an dishonest comparison since Raptor first includes a test selection heuristic, whereas Ochiai just executes every possible test. Additionally, Raptor-H and RaptorEPS2 have lower residual diagnostic effort as well. Per program, Raptor-H performs better in the cases where the precomputed information has a low error. On average, however, there is little difference between both algorithms. Finally, when it comes to memory consumption, Ochiai is the winner since it is a much lighter-weight approach than Raptor-* Bayesian diagnosis.

[3]DXC Competicion `https://sites.google.com/site/dxcompetition2011/`, 2012.

Appendix A

Meetings Summaries

Description of the subjects raised in the meetings between student and supervisor.

A.1 During Dissertation Planning

A.1.1 28^{th} January, 2011

- Raised Subjects
  - Set the dissertation proposal to submit in the MIEIC.
- Future Work
  - Write the dissertation proposal.

A.1.2 4^{th} February, 2011

- Raised Subjects
  - Set the proposal to submit in the *HP Labs Innovation Research Program*.
- Future Work
  - Write two/three paragraphs about the subject of dissertation.

A.1.3 9^{th} February, 2011

- Raised Subjects
  - Finishing the proposal to HP.
- Future Work
  - Reread the proposal and report feedback.

A.1.4 16^{th} February, 2011

- Raised Subjects
  - Keyworks.
  - Discuss a diagram that elucidates the way to go in this phase. Tests, test types, regression testing, how testing is approached by developers (what kind of tools are used, methodologies, etc.).
  - Influence authors on subject Test and special in Regression Testing.
- Future Work
  - Experiment, test and evaluate which frameworks, JUnit or TestNG is the most appropriate to this project.

A.1.5 23^{rd} February, 2011

- Raised Subjects
  - Difference between JUnit and TestNG frameworks. Show a practical example.
  - How GZOLTAR use JUnit or TestNG? We will not implement JUnit or TestNG on GZOLTAR, because this would limit upgrade of GZOLTAR. Instead, GZOLTAR ask for services from JUnit or TestNG.
  - How Microsoft Visual Studio 2010 implements unit tests.
- Future Work
  - Performance test of both frameworks.
  - Search for paper with related work on Microsoft Research (MSR).

A.1.6 03^{rd} March, 2011

- Raised Subjects
  - Performance of JUnit vs TestNG.
  - Architecture of GZOLTAR: how as implemented, data structures, where new module will be integrate to communicate with JUnit?
  - Similiar tools about regression testing? MINTS.
  - After the version 4.6 of JUnit, it has a new module called MaxCore (it remember the last execution of tests; prefer new test to old; quick tests to slow tests; tests that have failed for some time rather than the tests that failed a long time).
- Future Work
  - Analysis GZOLTAR architecture.

A.1.7 04^{th} April, 2011

- Raised Subjects
  - Timeline.
  - Minimization Heuristics: Greedy, HSG, Delay-Greedy, Selective Redundancy, Bi-Criteria, ILP with one example.
  - Our contribution, Constraint satisfaction problem (CSP).
- Future Work
  - Pseudo-text for Minimization.
  - Begin the study of Selection.

A.1.8 10^{th} May, 2011

- Raised Subjects
  - Talk about first presentation.
- Future Work
  - Begin the study of Prioritization.

A.1.9 14^{th} June, 2011

- Raised Subjects
  - Deadlines.
- Future Work
  - Finish the definition of Regression Testing.
  - Finish the Technique Minimization.

A.1.10 05^{th} July, 2011

- Raised Subjects
  - Delivered the following artifacts: Introduction, Definition of Regression Testing, Technique for Minimization.
- Future Work
  - Finish the Fault-detection and Technique for Selection.
  - Introduction, Motivation, Objectives.
  - Second Presentation.

A.1.11 11^{th} July, 2011

- Raised Subjects
  - Submitted the 'Testing and Automatic Debugging with GZOLTAR' to Google Testing Automation Conference 2011.
  - Talk about second presentation.
- Future Work
  - Finish the Fault-detection and Technique for Selection.
  - Introduction, Motivation, Objectives.

A.1.12 13^{th} July, 2011

- Raised Subjects
  - Delivered the following artifacts: Fault-detection, Technique for Selection.
- Future Work
  - Finish the Technique Prioritization.
  - Abstract.

A.1.13 19^{th} July, 2011

- Raised Subjects
  - Delivered the following artifacts: Technique for Prioritization, Abstract, Introduction, Motivation, Objectives.
- Future Work
  - Begin the second phase of the Dissertation.

A.1.14 20^{th} July, 2011

- Raised Subjects
  - Delivered the final version of 'Regression Testing with GZOLTAR: Techniques for Test Suite Minimization, Selection, and Prioritization'.
- Future Work
  - Vacation, and after that, begin the second phase of the Dissertation.

A.2 During Dissertation

A.2.1 23^{rd} September, 2011

- Raised Subjects
  - Meeting with André to share the last version of GZOLTAR.
- Future Work
  - Create the repository to the group (Rui, André, José, Alex, João, Nuno).
  - Integrate in existing source code.

A.2.2 7^{th} October, 2011

- Raised Subjects
  - Create the repository.
- Future Work
  - Integrate in existing source code. (continue)

A.2.3 14^{th} October, 2011

- Raised Subjects
  - Implementation of TRIE structure in Java.
  - Update JaCoCo plug-in to the last stable version.
- Future Work
  - Try to integrate MINION with GZOLTAR.

A.2.4 21^{st} October, 2011

- Raised Subjects
  - Update the struture of GZOLTAR plug-in.
  - Integration of MINION with GZOLTAR.
- Future Work
  - Create the View for Regression-Zoltar.

A.2.5 28th October, 2011

- Raised Subjects
  - First commit of Regression-Zoltar View.
- Future Work
  - Compile GZOLTAR & RZOLTAR to all plataforms.

A.2.6 4th November, 2011

- Raised Subjects
  - Update Regression-Zoltar View.
  - Talk about the limitation of RZOLTAR (projects that has unit test and need external files to run).
- Future Work
  - Try to integrate Ant with RZOLTAR.

A.2.7 11th November, 2011

- Raised Subjects
  - Update JaCoCo plug-in to the last stable version.
  - Change 'String' to 'StringBuilder' (thanks Nuno).
- Future Work
  - Try to integrate Ant with RZOLTAR. (continue)

A.2.8 18th November, 2011

- Raised Subjects
  - Update RZOLTAR View. Add button to choose, how much wait for results?
- Future Work
  - Try to integrate Ant with RZOLTAR. (continue)

A.2.9 25th November, 2011

- Raised Subjects
  - Update RZOLTAR View. Disable icons at beginning.
  - Update GZOLTAR & RZOLTAR integration. Create Global WorkspaceInformation to the two views.
- Future Work
  - Try to integrate Ant with RZOLTAR. (continue)

A.2.10 2nd December, 2011

- Raised Subjects
  - Talk about the integration of Ant with RZOLTAR.
  - New version of TRIE to MacOs.

A.2.11 9th December, 2011

- Raised Subjects
  - Update RZOLTAR View. Add the option 'All Tests'.
  - Fix some bugs.
- Future Work
  - Start the manual and prepare one project example to Human Study.

A.2.12 15th and 16th December, 2011

- Raised Subjects
  - Human Study.
- Future Work
  - Try to implement JUnit plug-in methods and JaCoCo calls on RZOLTAR.

A.2.13 30th December, 2011

- Raised Subjects
  - It is not possible to implement the JUnit plug-in and JaCoCo plug-in on RZOLTAR.
- Future Work
  - Re-start writing the thesis report.

A.2.14 4th January, 2012

- Raised Subjects
  - Meeting with supervisor to define the last things to do.
  - First version of IJUP2012 paper.
- Future Work
  - Correct and finish the IJUP2012 paper.

A.2.15 5th January, 2012

- Raised Subjects
  - Submitted the 'RZOLTAR: a plug-in Eclipse for Regression Testing' to IJUP2012.
- Future Work
  - Writing the thesis report. (continue)

A.2.16 16th January, 2012

- Raised Subjects
  - Feedback of chapter 3.

A.2.17 18th January, 2012

- Raised Subjects
  - Feedback of chapter 6.

A.2.18 22nd January, 2012

- Raised Subjects
  - Feedback of chapter 3.

A.2.19 26th January, 2012

- Raised Subjects
  - Feedback of chapter 5.

A.2.20 27th January, 2012

- Raised Subjects
  - Feedback of abstract.

Appendix B

Perfomance - JUnit vs TestNG

B.1 Calculator Project

In Table B.1 are represented all values that we measured during the performance test between JUnit and TestNG frameworks. For each framework we use several sizes of test suite, 10, 50, 100, 500, 1000 represented by |T| and for each test suite we regist 100 measures, represented by |E|. We run all tests suites on Linux, on a 2.27 Ghz Intel Core i3-350M with 4 GB of RAM running Debian Gnu/Linux Wheezy.

	JUnit					TestNG										
E \ |T|	**10**	**50**	**100**	**500**	**1000**	**10**	**10 (5 threads)**	**50**	**50 (5 threads)**	**100**	**100 (5 threads)**	**500**	**500 (5 threads)**	**1000**	**1000 (5 threads)**
1	0.025s	0.064s	0.096s	0.300s	0.497s	0.424s	0.314s	1.996s	0.639s	4.231s	1.633s	20.134s	7.096s	40.117s	22.966s
2	0.025s	0.064s	0.094s	0.360s	0.549s	0.418s	0.377s	1.996s	0.609s	4.018s	1.288s	20.121s	7.564s	40.070s	20.992s
3	0.025s	0.057s	0.089s	0.313s	0.565s	0.406s	0.317s	2.000s	0.552s	4.002s	1.478s	20.128s	7.138s	40.111s	17.886s
4	0.024s	0.067s	0.100s	0.299s	0.685s	0.418s	0.349s	2.002s	0.675s	4.015s	1.388s	20.178s	6.960s	40.081s	18.213s
5	0.026s	0.062s	0.099s	0.306s	0.490s	0.564s	0.344s	2.005s	0.636s	3.989s	0.966s	20.055s	7.010s	40.462s	19.598s
6	0.023s	0.066s	0.103s	0.289s	0.470s	0.419s	0.331s	2.026s	0.504s	4.001s	1.084s	20.124s	7.577s	40.676s	19.336s
7	0.028s	0.063s	0.100s	0.305s	0.527s	0.410s	0.276s	2.013s	0.546s	3.974s	1.234s	20.167s	7.610s	40.221s	16.856s
8	0.027s	0.068s	0.123s	0.306s	0.556s	0.417s	0.303s	2.016s	0.530s	3.992s	1.135s	20.017s	7.355s	40.469s	17.988s
9	0.023s	0.062s	0.094s	0.328s	0.537s	0.408s	0.265s	2.000s	0.486s	3.988s	1.236s	20.026s	7.385s	40.313s	17.104s
10	0.024s	0.055s	0.095s	0.268s	0.545s	0.403s	0.292s	2.005s	0.497s	3.996s	1.087s	20.075s	7.519s	40.658s	16.738s
11	0.022s	0.065s	0.098s	0.306s	0.577s	0.416s	0.268s	2.004s	0.528s	3.984s	1.229s	20.025s	7.653s	40.184s	19.295s
12	0.025s	0.067s	0.095s	0.309s	0.508s	0.408s	0.360s	2.007s	0.493s	3.994s	1.208s	20.100s	7.864s	40.409s	17.446s
13	0.025s	0.066s	0.094s	0.267s	0.490s	0.425s	0.310s	2.038s	0.487s	4.004s	1.024s	20.092s	7.583s	40.243s	18.935s
14	0.025s	0.067s	0.095s	0.290s	0.565s	0.406s	0.302s	1.992s	0.522s	4.028s	1.251s	20.145s	7.200s	40.476s	20.207s
15	0.025s	0.064s	0.090s	0.298s	0.497s	0.405s	0.327s	2.000s	0.513s	3.990s	1.281s	20.079s	7.385s	40.258s	19.514s
16	0.024s	0.067s	0.096s	0.266s	0.476s	0.412s	0.308s	2.008s	0.463s	3.985s	1.624s	19.998s	7.363s	40.370s	19.545s
17	0.023s	0.062s	0.094s	0.288s	0.517s	0.406s	0.288s	2.015s	0.420s	3.990s	1.485s	20.032s	6.955s	40.260s	20.076s
18	0.023s	0.064s	0.094s	0.296s	0.503s	0.420s	0.303s	1.987s	0.606s	3.990s	1.284s	20.147s	7.462s	40.445s	21.217s
19	0.026s	0.059s	0.095s	0.271s	0.532s	0.408s	0.388s	1.992s	0.463s	3.975s	1.156s	20.030s	7.532s	40.183s	17.370s
20	0.023s	0.062s	0.095s	0.308s	0.557s	0.412s	0.297s	2.024s	0.581s	3.999s	1.166s	20.036s	7.067s	40.069s	18.775s
21	0.025s	0.060s	0.093s	0.309s	0.507s	0.408s	0.349s	2.004s	0.522s	4.007s	1.025s	20.058s	7.183s	40.281s	19.157s
22	0.023s	0.068s	0.095s	0.295s	0.505s	0.412s	0.415s	2.009s	0.572s	3.996s	1.090s	20.045s	6.957s	40.602s	18.936s
23	0.023s	0.057s	0.098s	0.286s	0.514s	0.401s	0.286s	2.008s	0.422s	3.997s	1.235s	20.028s	7.749s	40.327s	17.577s
24	0.022s	0.069s	0.094s	0.271s	0.490s	0.399s	0.364s	2.000s	0.466s	3.977s	1.327s	20.056s	6.916s	40.262s	19.124s
25	0.024s	0.063s	0.093s	0.277s	0.496s	0.423s	0.386s	1.994s	0.480s	3.997s	1.396s	20.052s	7.636s	40.204s	19.222s
26	0.023s	0.062s	0.087s	0.294s	0.527s	0.408s	0.324s	2.007s	0.493s	4.006s	1.103s	20.007s	7.108s	40.068s	19.991s
27	0.025s	0.064s	0.105s	0.304s	0.572s	0.407s	0.313s	1.995s	0.453s	4.003s	1.267s	20.064s	7.155s	40.154s	19.869s
28	0.029s	0.059s	0.092s	0.268s	0.477s	0.409s	0.296s	2.021s	0.463s	4.002s	1.234s	20.061s	7.652s	40.051s	19.900s
29	0.029s	0.067s	0.090s	0.293s	0.473s	0.415s	0.376s	2.004s	0.537s	3.979s	1.227s	20.232s	7.032s	40.041s	16.882s
30	0.027s	0.054s	0.098s	0.294s	0.546s	0.418s	0.334s	1.990s	0.498s	4.011s	1.100s	20.013s	7.544s	40.059s	18.599s
31	0.026s	0.059s	0.106s	0.301s	0.522s	0.411s	0.280s	1.994s	0.438s	3.972s	1.195s	20.089s	6.905s	40.234s	18.374s
32	0.028s	0.063s	0.102s	0.269s	0.505s	0.417s	0.375s	1.991s	0.479s	3.984s	1.067s	20.190s	7.051s	40.078s	19.698s
33	0.025s	0.065s	0.102s	0.340s	0.491s	0.407s	0.304s	2.002s	0.523s	3.994s	1.052s	20.132s	7.372s	40.186s	19.361s
34	0.023s	0.059s	0.093s	0.294s	0.532s	0.406s	0.350s	2.027s	0.489s	4.006s	1.209s	20.047s	6.828s	40.275s	20.693s
35	0.024s	0.059s	0.094s	0.265s	0.549s	0.409s	0.342s	1.987s	0.570s	4.007s	1.143s	20.025s	7.586s	40.071s	20.748s
36	0.024s	0.063s	0.093s	0.314s	0.489s	0.413s	0.362s	1.992s	0.525s	3.992s	1.078s	20.090s	7.128s	40.518s	20.187s
37	0.029s	0.059s	0.091s	0.294s	0.530s	0.410s	0.378s	2.009s	0.481s	3.991s	1.289s	20.016s	7.136s	40.059s	20.097s
38	0.039s	0.066s	0.093s	0.296s	0.459s	0.408s	0.344s	2.008s	0.515s	3.997s	1.177s	20.089s	7.489s	40.248s	19.339s
39	0.023s	0.064s	0.094s	0.271s	0.590s	0.413s	0.286s	1.990s	0.446s	3.985s	1.334s	20.058s	7.697s	40.114s	18.483s

Continued on next page

	JUnit					TestNG									
E \ |T|	**10**	**50**	**100**	**500**	**1000**	**10**	**10 (5 threads)**	**50**	**50 (5 threads)**	**100**	**100 (5 threads)**	**500**	**500 (5 threads)**	**1000**	**1000 (5 threads)**
40	0.023s	0.062s	0.093s	0.301s	0.489s	0.408s	0.363s	1.997s	0.421s	3.982s	1.044s	20.172s	6.946s	40.296s	17.120s
41	0.022s	0.066s	0.093s	0.324s	0.538s	0.420s	0.394s	2.006s	0.595s	3.997s	1.142s	20.078s	7.139s	40.251s	18.173s
42	0.025s	0.068s	0.092s	0.297s	0.505s	0.410s	0.333s	1.996s	0.531s	3.987s	1.106s	20.123s	6.985s	40.087s	19.251s
43	0.024s	0.063s	0.094s	0.301s	0.515s	0.409s	0.282s	1.997s	0.540s	3.976s	1.167s	20.044s	7.387s	40.288s	19.513s
44	0.024s	0.058s	0.095s	0.302s	0.502s	0.417s	0.348s	2.003s	0.515s	3.993s	1.254s	20.098s	7.579s	40.232s	19.956s
45	0.024s	0.065s	0.094s	0.297s	0.487s	0.413s	0.315s	2.015s	0.482s	3.991s	1.089s	20.170s	7.145s	40.301s	17.647s
46	0.024s	0.064s	0.087s	0.304s	0.484s	0.403s	0.314s	2.028s	0.602s	4.019s	1.351s	20.023s	7.075s	40.107s	20.629s
47	0.024s	0.058s	0.093s	0.269s	0.476s	0.416s	0.398s	2.020s	0.535s	4.007s	1.155s	20.049s	7.516s	40.182s	17.595s
48	0.022s	0.057s	0.093s	0.298s	0.550s	0.406s	0.275s	2.005s	0.469s	3.977s	1.356s	20.007s	7.894s	40.113s	21.077s
49	0.022s	0.063s	0.092s	0.311s	0.468s	0.400s	0.287s	1.993s	0.512s	3.987s	1.318s	20.153s	7.334s	40.084s	17.277s
50	0.023s	0.060s	0.092s	0.266s	0.511s	0.417s	0.309s	1.996s	0.523s	3.985s	1.095s	20.147s	7.440s	40.120s	18.730s
51	0.024s	0.065s	0.093s	0.259s	0.487s	0.415s	0.330s	2.004s	0.483s	4.000s	1.378s	20.150s	7.872s	40.156s	19.702s
52	0.022s	0.056s	0.094s	0.315s	0.468s	0.410s	0.321s	1.991s	0.439s	3.987s	1.199s	20.131s	7.459s	40.288s	18.579s
53	0.023s	0.068s	0.093s	0.326s	0.553s	0.401s	0.349s	1.998s	0.558s	3.976s	1.206s	20.096s	8.211s	40.189s	19.711s
54	0.023s	0.062s	0.092s	0.299s	0.563s	0.408s	0.393s	2.017s	0.447s	4.017s	1.131s	20.193s	7.826s	40.123s	19.314s
55	0.025s	0.062s	0.093s	0.312s	0.524s	0.402s	0.382s	2.015s	0.530s	4.005s	1.102s	20.153s	7.586s	40.125s	21.071s
56	0.025s	0.067s	0.092s	0.270s	0.481s	0.411s	0.271s	2.006s	0.478s	3.992s	1.273s	20.108s	7.455s	40.161s	20.874s
57	0.022s	0.064s	0.098s	0.308s	0.502s	0.410s	0.330s	1.995s	0.530s	3.997s	1.294s	20.089s	7.175s	40.204s	19.093s
58	0.024s	0.058s	0.095s	0.301s	0.470s	0.410s	0.337s	2.010s	0.535s	4.011s	0.970s	20.160s	7.436s	40.076s	19.695s
59	0.026s	0.062s	0.097s	0.298s	0.547s	0.400s	0.334s	2.023s	0.480s	4.001s	1.190s	20.049s	7.420s	40.019s	18.752s
60	0.024s	0.063s	0.095s	0.332s	0.557s	0.411s	0.267s	2.031s	0.610s	3.990s	1.162s	20.149s	6.968s	39.950s	20.513s
61	0.023s	0.062s	0.088s	0.308s	0.502s	0.408s	0.294s	2.015s	0.463s	3.996s	1.093s	20.011s	7.545s	40.040s	20.286s
62	0.024s	0.059s	0.090s	0.300s	0.470s	0.411s	0.282s	1.999s	0.539s	4.013s	1.263s	20.223s	7.460s	40.192s	19.277s
63	0.023s	0.061s	0.092s	0.307s	0.533s	0.396s	0.304s	2.012s	0.481s	3.978s	1.156s	20.187s	7.131s	39.975s	20.264s
64	0.024s	0.064s	0.091s	0.298s	0.505s	0.401s	0.307s	1.989s	0.544s	3.980s	1.248s	20.211s	7.219s	40.039s	19.369s
65	0.026s	0.055s	0.093s	0.299s	0.501s	0.409s	0.368s	2.005s	0.576s	3.983s	1.152s	20.301s	7.657s	40.027s	22.131s
66	0.022s	0.061s	0.095s	0.309s	0.481s	0.406s	0.375s	1.999s	0.507s	3.994s	1.186s	20.031s	7.291s	40.190s	20.324s
67	0.028s	0.057s	0.104s	0.265s	0.503s	0.412s	0.324s	2.002s	0.454s	3.992s	1.009s	20.136s	6.698s	39.973s	18.685s
68	0.023s	0.064s	0.090s	0.269s	0.537s	0.413s	0.320s	2.004s	0.556s	3.993s	1.310s	20.040s	7.507s	40.602s	19.606s
69	0.024s	0.058s	0.091s	0.289s	0.559s	0.403s	0.319s	2.003s	0.493s	3.993s	1.175s	20.058s	7.162s	40.003s	20.374s
70	0.025s	0.064s	0.093s	0.305s	0.494s	0.403s	0.314s	2.001s	0.502s	4.032s	1.161s	21.516s	6.867s	40.017s	19.005s
71	0.023s	0.065s	0.094s	0.300s	0.547s	0.411s	0.282s	1.993s	0.501s	3.977s	0.878s	20.170s	7.361s	40.125s	20.077s
72	0.024s	0.063s	0.092s	0.270s	0.442s	0.407s	0.298s	2.003s	0.535s	3.985s	1.306s	20.132s	7.556s	40.116s	20.107s
73	0.022s	0.064s	0.101s	0.305s	0.488s	0.412s	0.282s	2.005s	0.518s	3.980s	1.359s	20.153s	6.625s	40.202s	16.383s
74	0.026s	0.058s	0.093s	0.299s	0.471s	0.403s	0.324s	2.032s	0.567s	4.040s	1.282s	20.208s	7.542s	40.196s	19.113s
75	0.024s	0.063s	0.089s	0.299s	0.509s	0.400s	0.322s	2.014s	0.468s	3.983s	1.134s	20.087s	7.573s	39.995s	19.369s
76	0.024s	0.060s	0.094s	0.284s	0.504s	0.407s	0.302s	1.999s	0.523s	3.995s	1.528s	20.079s	7.764s	39.958s	16.927s
77	0.026s	0.060s	0.090s	0.305s	0.574s	0.400s	0.286s	2.000s	0.512s	3.988s	1.004s	20.162s	7.400s	40.004s	19.659s
78	0.024s	0.063s	0.092s	0.310s	0.477s	0.419s	0.381s	2.010s	0.576s	3.990s	0.969s	20.141s	7.455s	40.018s	20.640s
79	0.022s	0.057s	0.092s	0.296s	0.688s	0.405s	0.448s	1.995s	0.520s	4.000s	1.183s	20.175s	7.094s	40.210s	19.526s
80	0.023s	0.058s	0.094s	0.310s	0.585s	0.405s	0.420s	1.997s	0.495s	3.986s	1.204s	20.260s	7.376s	40.024s	19.010s
81	0.023s	0.058s	0.092s	0.268s	0.512s	0.397s	0.367s	2.006s	0.488s	4.007s	1.168s	20.081s	7.140s	40.147s	19.797s
82	0.025s	0.063s	0.097s	0.258s	0.480s	0.411s	0.366s	2.004s	0.464s	4.036s	1.237s	20.071s	7.022s	40.174s	19.990s
83	0.024s	0.073s	0.089s	0.286s	0.498s	0.412s	0.337s	2.360s	0.656s	4.006s	1.063s	20.151s	7.064s	40.188s	19.999s
84	0.023s	0.066s	0.111s	0.311s	0.491s	0.404s	0.344s	2.007s	0.593s	3.985s	1.052s	20.041s	7.365s	39.971s	19.718s
85	0.023s	0.059s	0.099s	0.359s	0.479s	0.404s	0.314s	1.992s	0.561s	4.019s	1.135s	20.263s	7.335s	39.963s	20.849s
86	0.024s	0.057s	0.092s	0.296s	0.494s	0.415s	0.326s	1.997s	0.491s	3.997s	1.167s	20.271s	7.424s	40.029s	20.493s
87	0.024s	0.066s	0.093s	0.301s	0.462s	0.403s	0.295s	1.996s	0.567s	3.992s	0.986s	20.084s	7.559s	40.054s	19.859s
88	0.023s	0.061s	0.093s	0.295s	0.477s	0.412s	0.342s	2.008s	0.548s	3.983s	1.142s	20.103s	7.852s	39.998s	19.813s
89	0.023s	0.065s	0.092s	0.286s	0.527s	0.396s	0.318s	2.014s	0.547s	4.003s	1.137s	20.079s	7.281s	40.001s	19.291s
90	0.023s	0.068s	0.092s	0.306s	0.528s	0.409s	0.383s	2.003s	0.485s	3.980s	1.107s	20.039s	7.603s	40.087s	16.964s
91	0.023s	0.063s	0.093s	0.273s	0.508s	0.411s	0.308s	2.010s	0.488s	4.056s	1.232s	19.994s	7.365s	40.034s	20.212s
92	0.027s	0.074s	0.098s	0.340s	0.490s	0.412s	0.344s	2.012s	0.460s	3.988s	1.281s	20.047s	7.385s	39.990s	18.533s
93	0.023s	0.063s	0.093s	0.262s	0.534s	0.403s	0.358s	2.006s	0.555s	4.000s	1.046s	20.036s	7.457s	40.054s	18.528s
94	0.023s	0.063s	0.094s	0.283s	0.508s	0.406s	0.331s	2.086s	0.540s	4.008s	1.259s	20.231s	7.195s	40.098s	20.306s
95	0.023s	0.060s	0.092s	0.294s	0.516s	0.413s	0.332s	2.009s	0.472s	3.995s	1.270s	20.044s	7.034s	40.166s	20.151s
96	0.026s	0.072s	0.089s	0.301s	0.590s	0.411s	0.329s	2.001s	0.544s	3.996s	1.121s	20.029s	7.454s	40.098s	18.160s
97	0.024s	0.063s	0.091s	0.308s	0.514s	0.401s	0.325s	2.016s	0.493s	4.003s	1.116s	20.009s	7.957s	40.106s	19.408s
98	0.022s	0.066s	0.097s	0.294s	0.540s	0.402s	0.275s	2.007s	0.474s	4.003s	0.929s	20.038s	7.558s	40.073s	18.736s
99	0.024s	0.064s	0.092s	0.271s	0.506s	0.410s	0.411s	1.987s	0.574s	3.991s	1.026s	20.114s	7.434s	40.220s	19.298s
100	0.023s	0.057s	0.099s	0.303s	0.461s	0.405s	0.419s	1.998s	0.523s	4.002s	0.987s	20.240s	7.045s	40.036s	20.222s

Table B.1: Executions time of differente size of test suite with JUnit and TestNG frameworks

Appendix C

Empirical Evaluation, detailed results

Tables C.1, C.2, and C.3 presented the detailed results from empirical evaluation.

C.1 NanoXML

	Original Suite	Minimal Suite		RZOLTAR		
Experiences \ |T|	9	Suite 1: 8	Suite 2: 8	MINION	Trie	MINION + Trie
1	2.082s	1.829s	1.865s	0.108s	0.001s	0.109s
2	2.008s	1.780s	1.784s	0.108s	0.001s	0.109s
3	1.927s	1.760s	1.682s	0.106s	0.001s	0.107s
4	1.927s	1.716s	1.701s	0.102s	0.001s	0.104s
5	1.900s	1.726s	1.664s	0.095s	0.001s	0.096s
6	2.047s	1.857s	1.596s	0.112s	0.001s	0.113s
7	1.901s	1.693s	1.674s	0.136s	0.001s	0.137s
8	2.007s	1.785s	1.789s	0.093s	0.001s	0.094s
9	1.945s	1.736s	1.726s	0.102s	0.001s	0.103s
10	2.020s	1.797s	1.752s	0.094s	0.001s	0.095s
$\langle E \rangle$	1.976s	1.768s	1.723s	0.106s	0.001s	0.107s
σ	0.064	0.052	0.077	0.012	0.000	0.013
$\langle E \rangle$ **% Reduction**		10.55%	12.81%			
|T| % Reduction		11.11%				

Table C.1: NanoXML detailed results

C.2 org.jacoco.report

	Original Suite	**Minimal Suite**	**RZOLTAR**		
|T| / Experiences	**225**	**Suite 1: 55**	**MINION**	**Trie**	**MINION + Trie**
1	39.044s	10.582s	0.439s	0.002s	0.441s
2	39.553s	10.898s	0.447s	0.001s	0.448s
3	37.952s	10.728s	0.407s	0.001s	0.408s
4	39.224s	10.455s	0.422s	0.001s	0.423s
5	38.973s	10.792s	0.422s	0.001s	0.423s
6	39.554s	10.882s	0.427s	0.001s	0.428s
7	39.172s	10.565s	0.436s	0.001s	0.437s
8	39.045s	10.540s	0.416s	0.002s	0.418s
9	38.448s	10.358s	0.452s	0.002s	0.454s
10	39.072s	10.905s	0.417s	0.003s	0.420s
$\langle E \rangle$	39.04s	10.671s	0.429s	0.002s	0.430s
σ	0.484	0.197	0.015	0.001	0.014
$\langle E \rangle$ **% Reduction**		72.64%			
|T| % Reduction		75.56%			

Table C.2: org.jacoco.report detailed results

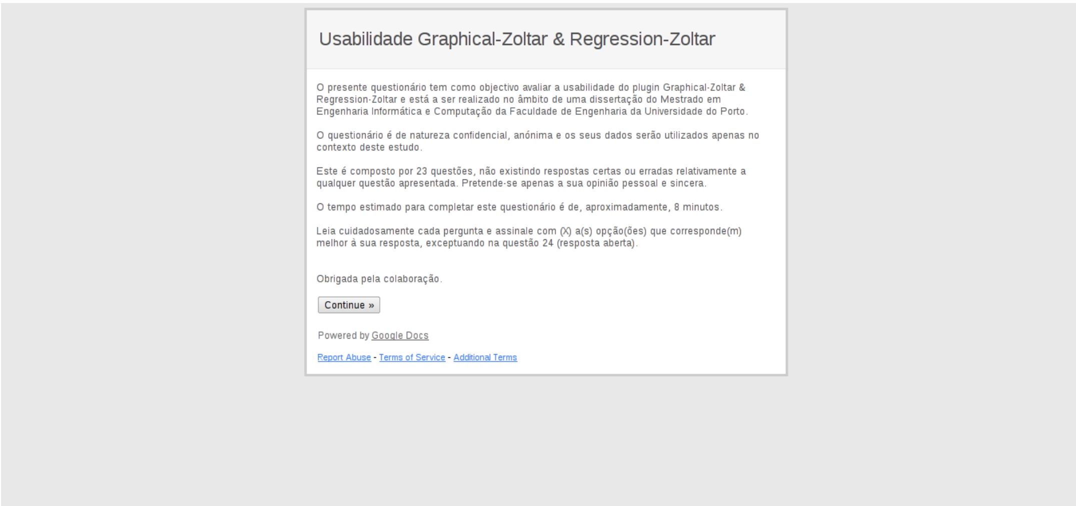
C.3 JTopas

	Original Suite	**Minimal Suite**						**RZOLTAR**		
|T| / Experiences	**57**	**Suite 1: 26**	**Suite 2: 26**	**Suite 3: 26**	**Suite 4: 26**	**Suite 5: 26**	**Suite 6: 26**	**MINION**	**Trie**	**MINION + Trie**
1	275.463s	157.446s	157.427s	157.398s	157.739s	157.710s	157.417s	1.192s	0.991s	2.183s
2	276.035s	158.201s	158.262s	158.209s	158.587s	158.526s	158.270s	1.219s	1.005s	2.224s
3	274.624s	157.437s	157.467s	157.783s	157.459s	157.489s	157.753s	1.185s	1.002s	2.187s
4	276.056s	158.849s	158.570s	158.869s	158.550s	158.547s	158.567s	1.189s	1.039s	2.228s
5	275.376s	157.763s	157.448s	157.740s	157.542s	157.536s	157.723s	1.196s	1.009s	2.205s
6	276.567s	157.444s	157.338s	157.191s	157.578s	157.710s	157.526s	1.204s	1.014s	2.218s
7	275.138s	157.691s	157.551s	157.508s	157.658s	157.613s	157.295s	1.188s	1.017s	2.205s
8	275.552s	158.778s	158.910s	158.547s	158.512s	158.548s	158.189s	1.199s	1.022s	2.221s
9	276.360s	157.695s	157.232s	157.486s	157.637s	157.590s	157.139s	1.213s	1.032s	2.245s
10	275.558s	157.606s	157.509s	157.570s	157.700s	157.622s	157.293s	1.199s	1.025s	2.224s
$\langle E \rangle$	275.673s	157.891s	157.771s	157.830s	157.896s	157.889s	157.717s	1.198s	1.016s	2.214s
σ	0.586	0.535	0.586	0.541	0.458	0.454	0.480	0.011	0.015	0.019
$\langle E \rangle$ **% Reduction**		42.73%	42.80%	42.70%	42.70%	42.70%	42.80%			
|T| % Reduction		54.39%								

Table C.3: JTopas detailed results

Appendix D

Survey



Usabilidade Graphical-Zoltar & Regression-Zoltar

O presente questionário tem como objectivo avaliar a usabilidade do plugin Graphical-Zoltar & Regression-Zoltar e está a ser realizado no âmbito de uma dissertação do Mestrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

O questionário é de natureza confidencial, anónima e os seus dados serão utilizados apenas no contexto deste estudo.

Este é composto por 23 questões, não existindo respostas certas ou erradas relativamente a qualquer questão apresentada. Pretende-se apenas a sua opinião pessoal e sincera.

O tempo estimado para completar este questionário é de, aproximadamente, 8 minutos.

Leia cuidadosamente cada pergunta e assinale com (X) a(s) opção(ões) que corresponde(m) melhor à sua resposta, exceptuando na questão 24 (resposta aberta).

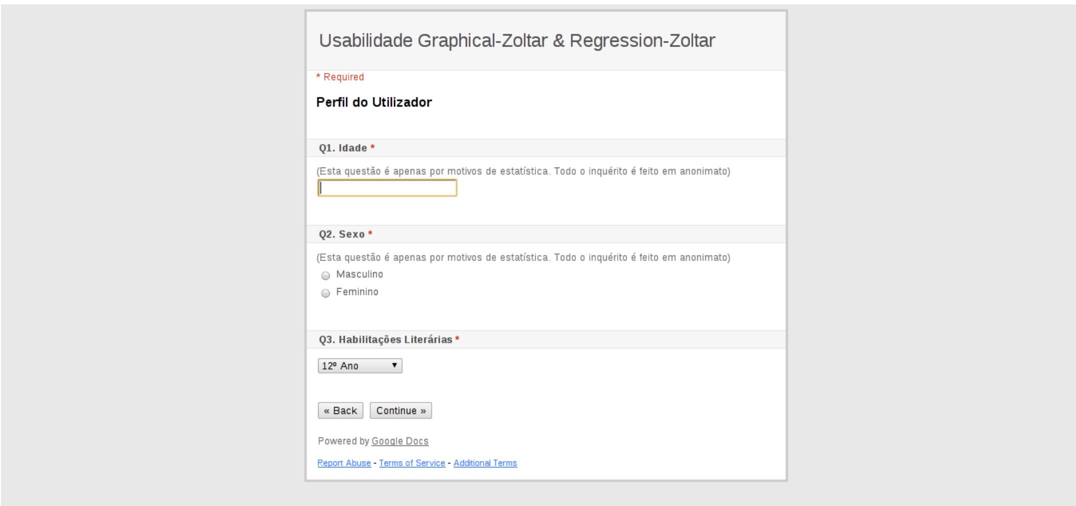
Obrigada pela colaboração.

Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure D.1: Survey - Introduction.



Usabilidade Graphical-Zoltar & Regression-Zoltar

* Required

Perfil do Utilizador

Q1. Idade *

(Esta questão é apenas por motivos de estatística. Todo o inquérito é feito em anonimato)

Q2. Sexo *

(Esta questão é apenas por motivos de estatística. Todo o inquérito é feito em anonimato)

- Masculino
- Feminino

Q3. Habilitações Literárias *

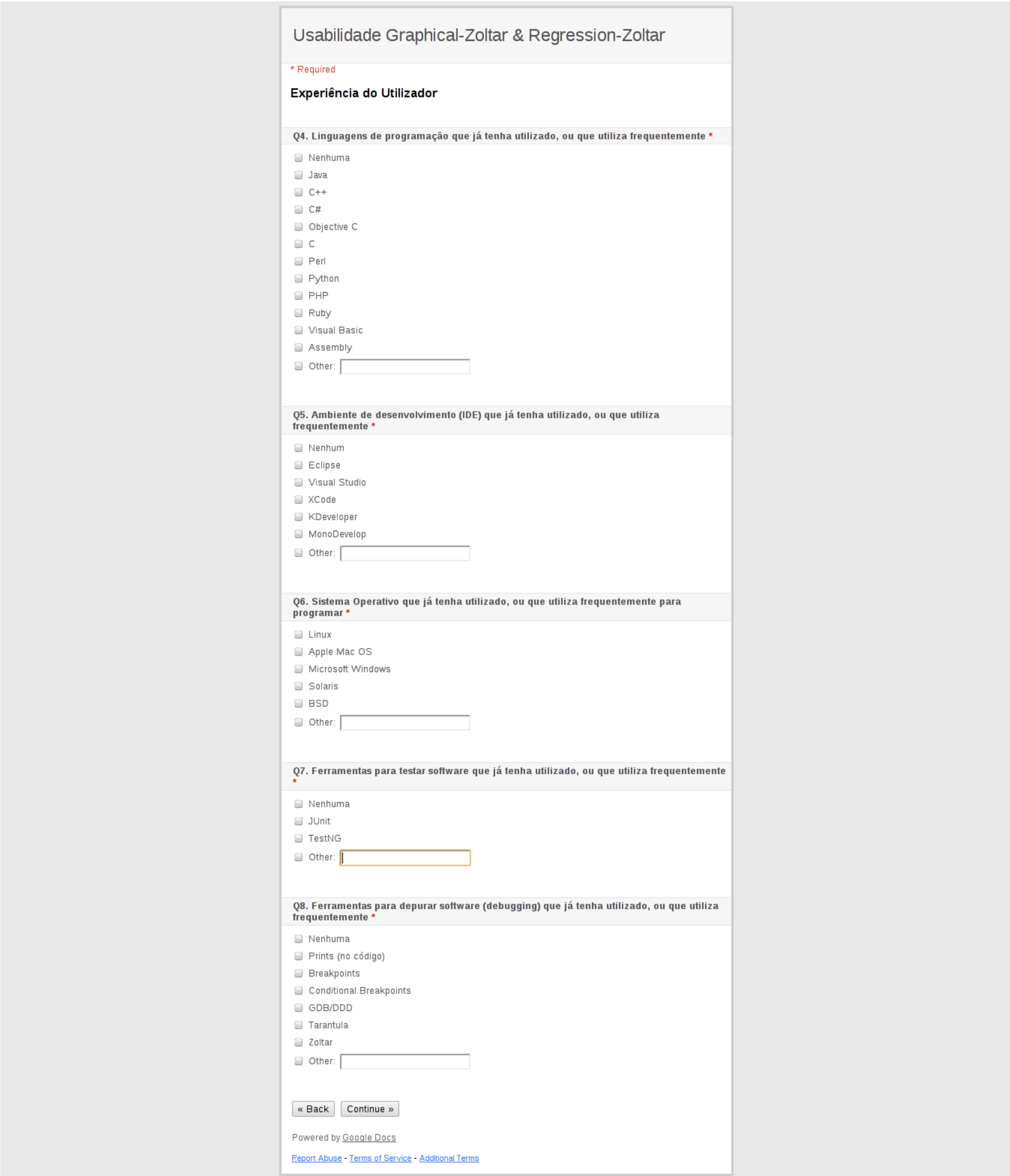
12º Ano

« Back   Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure D.2: Survey - User profile.



Usabilidade Graphical-Zoltar & Regression-Zoltar

* Required

Experiência do Utilizador

Q4. Linguagens de programação que já tenha utilizado, ou que utiliza frequentemente *

- Nenhuma
- Java
- C++
- C#
- Objective C
- C
- Perl
- Python
- PHP
- Ruby
- Visual Basic
- Assembly
- Other:

Q5. Ambiente de desenvolvimento (IDE) que já tenha utilizado, ou que utiliza frequentemente *

- Nenhum
- Eclipse
- Visual Studio
- XCode
- KDeveloper
- MonoDevelop
- Other:

Q6. Sistema Operativo que já tenha utilizado, ou que utiliza frequentemente para programar *

- Linux
- Apple Mac OS
- Microsoft Windows
- Solaris
- BSD
- Other:

Q7. Ferramentas para testar software que já tenha utilizado, ou que utiliza frequentemente *

- Nenhuma
- JUnit
- TestNG
- Other:

Q8. Ferramentas para depurar software (debugging) que já tenha utilizado, ou que utiliza frequentemente *

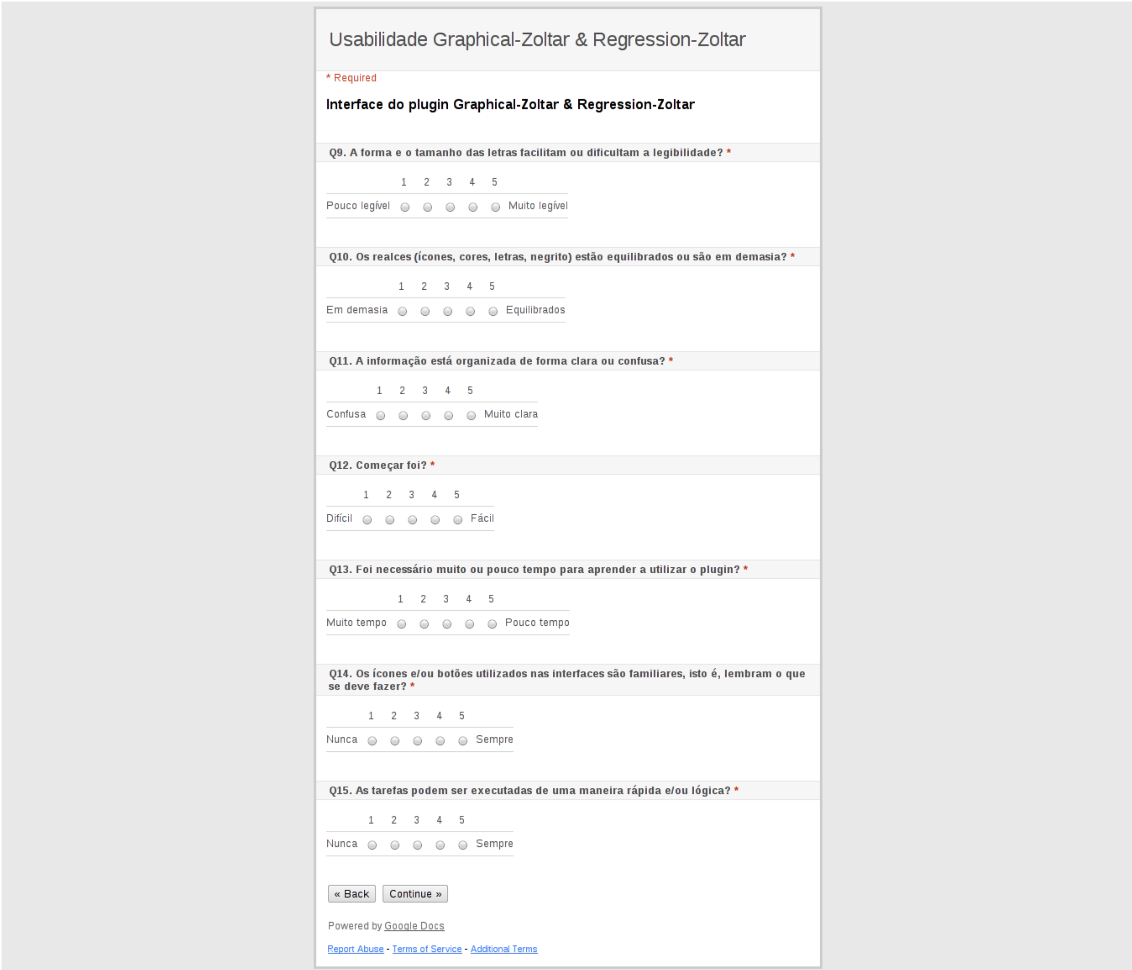
- Nenhuma
- Prints (no código)
- Breakpoints
- Conditional Breakpoints
- GDB/DDD
- Tarantula
- Zoltar
- Other:

« Back | Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure D.3: Survey - Experience.



Usabilidade Graphical-Zoltar & Regression-Zoltar

* Required

Interface do plugin Graphical-Zoltar & Regression-Zoltar

Q9. A forma e o tamanho das letras facilitam ou dificultam a legibilidade? *

	1	2	3	4	5	
Pouco legível	○	○	○	○	○	Muito legível

Q10. Os realces (ícones, cores, letras, negrito) estão equilibrados ou são em demasia? *

	1	2	3	4	5	
Em demasia	○	○	○	○	○	Equilibrados

Q11. A informação está organizada de forma clara ou confusa? *

	1	2	3	4	5	
Confusa	○	○	○	○	○	Muito clara

Q12. Começar foi? *

	1	2	3	4	5	
Difícil	○	○	○	○	○	Fácil

Q13. Foi necessário muito ou pouco tempo para aprender a utilizar o plugin? *

	1	2	3	4	5	
Muito tempo	○	○	○	○	○	Pouco tempo

Q14. Os ícones e/ou botões utilizados nas interfaces são familiares, isto é, lembram o que se deve fazer? *

	1	2	3	4	5	
Nunca	○	○	○	○	○	Sempre

Q15. As tarefas podem ser executadas de uma maneira rápida e/ou lógica? *

	1	2	3	4	5	
Nunca	○	○	○	○	○	Sempre

« Back Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure D.4: Survey - Interface of GZoltar and RZoltar.

Usabilidade Graphical-Zoltar & Regression-Zoltar

* Required

Capacidade do Plugin

Q16. A velocidade de resposta durante a realização das tarefas *

	1	2	3	4	5	
Muito lento	○	○	○	○	○	Excelente

Q17. Considera que o uso do plugin depende do nível de experiência do utilizador? *

	1	2	3	4	5	
Pouco	○	○	○	○	○	Muito

Q18. O utilizador realiza as tarefas com pouco conhecimento? *

	1	2	3	4	5	
Com dificuldade	○	○	○	○	○	Facilmente

« Back Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure D.5: Survey - Capacity of plug-in.

Figure D.6: Survey - Concepts.

Figure D.7: Survey - Global experience.

References

[AHKL93] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser e Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[AZvG07] Rui Abreu, Peter Zoeteweij e Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[Bla12] Heiko Blau. Jtopas - java tokenizer and parser tools. `http://jtopas.sourceforge.net/jtopas/`, 2012.

[BMK04] Jennifer Black, Emanuel Melachrinoudis e David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society.

[Chv79] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[Cor01] T.H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[CRV94] Yih-Farn Chen, David S. Rosenblum e Kiem-Phong Vo. Testtube: a system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[DRK04] Hyunsook Do, Gregg Rothermel e Alex Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 113–124, Washington, DC, USA, 2004. IEEE Computer Society.

[EM02] Sebastian Elbaum e Alexey Malishevsky. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28:159–182, 2002.

[FFH+11] Michael R. Fellows, Tobias Friedrich, Danny Hermelin, Nina Narodytska e Frances A. Rosamond. Constraint satisfaction problems: Convexity makes alldifferent constraints tractable. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 522–527. IJCAI/AAAI, 2011.

[Fou11] Eclipse Foundation. Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/, 2011.

[FRC81] K. Fischer, F. Raji e A. Chruscicki. A methodology for retesting modified software. 1981.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[Gee05] David Geer. Eclipse becomes the dominant java ide. *Computer*, 38(7):16–18, 2005.

[GJ90] Michael R. Garey e David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GJM06] Ian P. Gent, Chris Jefferson e Ian Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*, pages 98–102, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[GJMN07] Ian P. Gent, Chris Jefferson, Ian Miguel e Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 191–197. AAAI Press, 2007.

[GMR07] Ian P. Gent, Ian Miguel e Andrea Rendl. Tailoring solver-independent constraint models: a case study with essence' and minion. In *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation*, SARA'07, pages 184–199, Berlin, Heidelberg, 2007. Springer-Verlag.

[Het88] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[HGS93] M. Jean Harrold, Rajiv Gupta e Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2:270–285, July 1993.

[HO09] Hwa-You Hsu e Alessandro Orso. *MINTS: A general framework and tool for supporting test-suite minimization*, volume 0, pages 419–429. IEEE Computer Society, 2009.

[Hof11a] Marc R. Hoffmann. Eclemma - jacoco java code coverage library. http://www.eclemma.org/jacoco/index.html, 2011.

[Hof11b] Marc R. Hoffmann. Eclemma - java code coverage for eclipse. http://www.eclemma.org/, 2011.

[Hof11c] Marc R. Hoffmann. The future of code coverage for eclipse. http://www.eclipsecon.org/summiteurope2010/sessions/?page=sessions&id=1745, 2011.

[JAG09] Tom Janssen, Rui Abreu e Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, Washington, DC, USA, 2009. IEEE Computer Society.

[JG05] Dennis Jeffrey e Neelam Gupta. Test suite reduction with selective redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 549–558, Washington, DC, USA, 2005. IEEE Computer Society.

[KNP+09] Tolga Kurtoglu, Sriram Narasimhan, Scott Poll, David Garcia, Lukas Kuhn, Johan de Kleer, Arjan van Gemund e Alexander Feldman. Towards a framework for evaluating and comparing diagnosis algorithms. In *Proceedings of the Twentieth International Workshop on Principles of Diagnosis (DX'09), Stockholm Sweden*, pages 373–382. Erik Frisk and Mattias Nyberg and Mattias Krysander and Jan Åslund, June 2009.

[KP02] Jung-Min Kim e Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, New York, NY, USA, 2002. ACM.

[KWZ08] Sunghun Kim, E. James Whitehead, Jr. e Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34:181–196, March 2008.

[LH88] John A. N. Lee e Xudong He. A methodology for test selection. Technical report, Blacksburg, VA, USA, 1988.

[LP03] David Leon e Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 442–, Washington, DC, USA, 2003. IEEE Computer Society.

[Lyu96] Michael R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[McC06] Mike McCullough. Developing eclipse plugins. *Linux J.*, 2006(143):11, 2006.

[Mes07] G. Meszaros. *xUnit test patterns: refactoring test code*. The Addison-Wesley signature series. Addison-Wesley, 2007.

[MT08] Siavash Mirarab e Ladan Tahvildari. An empirical study on bayesian network-based approach for test case prioritization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 278–287, Washington, DC, USA, 2008. IEEE Computer Society.

[NL93] Jakob Nielsen e Thomas K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, pages 206–213, New York, NY, USA, 1993. ACM.

[NPW08] Mihai Nica, Bernhard Peischl e Franz Wotawa. A constraint model for automated deployment of automotive control software. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE 2008), San Francisco, CA, USA, July 1-3, 2008*, pages 899–904. Knowledge Systems Institute Graduate School, 2008.

[OB88] T. J. Ostrand e M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, June 1988.

[OPV95] A. Jefferson Offutt, Jie Pan e Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123, 1995.

[Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[RA10] André Riboira e Rui Abreu. The gzoltar project: a graphical debugger interface. In *Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques*, TAIC PART'10, pages 215–218, Berlin, Heidelberg, 2010. Springer-Verlag.

[RH96] Gregg Rothermel e Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Software Eng.*, 22(8):529–551, 1996.

[RH97] Gregg Rothermel e Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6:173–210, April 1997.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin e Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 34–, Washington, DC, USA, 1998. IEEE Computer Society.

[Rou11] Vlad Roubtsov. Emma: a free java code coverage tool. http://emma.sourceforge.net/index.html, 2011.

[RUCH01] G. Rothermel, R.H. Untch, Chengyun Chu e M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929 –948, oct 2001.

[Sch12] Marc De Scheemaecker. Nanoxml - small xml parser for java. http://devkix.com/nanoxml.php, 2012.

[SDH11] Martin Sachenbacher, Oskar Dressler e Michael Hofbaur. Third international diagnostic competition - dxc'11. In *Proceedings of the Twenty Second International Workshop on Principles of Diagnosis (DX'11), Murnau Germany*, pages 267–278. Scott Poll and Johan de Kleer and Rui Abreu and Matthew Daigle and Alexander Feldman and David Garcia and Alberto Gonzalez-Sanchez and Tolga Kurtoglu and Sriram Narasimhan and Adam Sweet, October 2011.

[Som07] I. Sommerville. *Software engineering*. International computer science series. Addison-Wesley, 2007.

[Tah92] Abu-Bakr Mostafa Taha. *An approach to software fault localization and revalidation based on incremental data flow analysis*. PhD thesis, Gainesville, FL, USA, 1992. UMI Order No. GAX92-02056.

[TG05] Sriraman Tallam e Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Softw. Eng. Notes*, 31:35–42, September 2005.

[VF98] F. I. Vokolos e P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, Washington, DC, USA, 1998. IEEE Computer Society.

[Wat01] J.E. Watkins. *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2001.

[WHLB97] W. Eric Wong, Joseph R. Horgan, Saul London e Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, Washington, DC, USA, 1997. IEEE Computer Society.

[Wil99] H.P. Williams. *Model building in mathematical programming*. A Wiley-Interscience publication. Wiley, 1999.

[YH10] S. Yoo e M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 2010.

REFERENCES