UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Applying TSR techniques over large test suites

João Pedro Pereira Becho

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor Rui André Oliveira

2020

Acknowledgments

First of all, I would like to thank my advisor, Professor Rui Oliveira, for guiding me through all these months, for always being available to help me whenever I needed, for giving me tips and advice on how to overcome my obstacles, both curricular and personal. I would also like to thank Professor Frederico Cerveira, from the University of Coimbra, for his cooperation in this project, for providing support and for being available to help me during this journey.

Thanks to all the researchers I contacted when I needed their support during the study of some tools, namely Andrea Stocco from the University of Washington, Professor José Campos, from the University of Lisbon, and Yves Ledru, from the *Laboratoire d'Informatique de Grenoble*.

To my parents, who have always patiently waited for me to achieve my goals, and for always being understanding and supportive, and for giving me all the opportunities I needed, and to my brother and sister, who have always been by my side.

I would also like to thank the people that went through this with me, for being with me since my bachelor degree and for all the good times we passed at FCUL, namely João Batista, Francisco Araújo, Nuno Rodrigues, Tiago Correia, João Pinto and Nuno Burnay.

To my friends from Santarém, for always being with me, even if we don't see each other very often anymore, for always supporting me and for always helping me make the right decisions in life, namely Kiko Antunes, Sofia Claro, Leonardo Ricardo, Patrícia Guilherme, Gonçalo Alves, Edmundo Marvão. A special thanks to Mariana Gaspar Fernandes, for always pushing me, for always taking the time to help me when I was blocked, even when you were busy with your own thesis, for the days in "caleidoscopio" or McDonald's procrastinating while observing pigeons.

To Francisco Picado and 'padrinho' João Pedro Almeida, thank you for all the moments we had, the dinners, board games, long conversations, exchange of ideas, and for being there since the beginning of this long journey.

Finally, to my girlfriend Rita Beja, for being one of the main reasons I was able to finish this thesis, for never giving up on me and always giving me strength to continue. You were my most present companion throughout all this work. It was a long ("muito longa") ride, and I'm really happy I had you by my side since the very beginning. For all the smiles, all the tears, all the moments, and all those clichés stuff you make so real, I will always be grateful to you.

Thank you all.

To Lola, for all the fun but frustrating moments

Resumo

Com o aumento da necessidade de garantir a qualidade do software criado atualmente, tem sido cada vez mais indispensável dedicar parte do tempo de desenvolvimento (por vezes mais de metade) a testar o código desenvolvido.

Apesar da elevada importância desta tarefa, é uma atividade que algumas vezes é ignorada ou negligenciada, devido a ser um trabalho por vezes monótono e cansativo.

Esta é uma área bastante explorada por investigadores que tentam, de diversas maneiras, automatizar algum deste processo e, também, reduzir o tempo e recursos necessários para efetuar esta tarefa. Nomeadamente, considera-se a proposta de técnicas de redução de testes e ferramentas que as implementem, com o objetivo de agilizar esta tarefa, eliminando casos de testes desnecessários sem comprometer a cobertura dada pelo conjunto de testes original, bem como técnicas de prioritização de testes, que reorganizam conjuntos de testes com o objetivo de executar os mais relevantes (de acordo com um determinado critério) primeiramente, aumentando assim a sua capacidade de atingir algum objetivo.

Neste contexto, destacam-se os testes de regressão, que permitem testar alterações ao software, verificando se as funcionalidades antigas continuam a funcionar com as alterações feitas. Salientam-se estes testes na medida em que são considerados o tipo de testes de software mais cansativo, comportando elevados custos e não evidenciando ganho claros a curto-prazo e, consequentemente, beneficiando particularmente das técnicas descritas anteriormente.

Com o surgimento destas técnicas, é inevitável surgirem algumas comparações, tentando escolher as que melhor se adequam a diferentes necessidades. O objetivo deste trabalho consiste em dar resposta a duas questões, particularmente interessantes dado o estado de arte atual: "Qual a melhor ferramenta de redução de testes?"(de acordo com parâmetros pré-definidos) e "Se uma ferramenta de prioritização de testes for modificada, pode substituir de forma eficiente uma ferramenta de redução de testes?".

Para realizar a presente dissertação, foi estudado um grupo de ferramentas de redução de testes, de forma a ter uma melhor noção do estado de arte atual. Apesar de inicialmente terem sido encontradas onze ferramentas que poderiam vir a ser usadas com este propósito, os testes realizados, assim como as propriedades de algumas ferramentas, restringiram a utilização da maioria delas, Assim, foram consideradas três ferramentas: EvoSuite, Testler e Randoop. De forma a tornar o objetivo deste trabalho mais enriquecido,

foi também estudada uma ferramenta de prioritização de testes, Kanonizo.

Para respondermos às questões apresentadas, foi desenvolvida uma ferramenta que integra o conjunto de ferramentas de redução de testes seleccionado e que, dado um conjunto de projetos *open-source*, aplica as técnicas destas ferramentas a cada um destes, efetuando assim a redução dos seus testes. Seguidamente, a ferramenta desenvolvida utiliza a ferramenta de prioritização Kanonizo para obter uma lista dos vários testes, ordenados consoante a sua importância; finalmente, são eliminados os testes menos importantes, segundo um valor pré-definido.

De seguida, utilizando uma ferramenta de análise de código, neste caso a OpenClover, são recolhidas métricas referentes a cada conjunto de testes, para os projetos originais e igualmente para cada um dos reduzidos. Estas são depois utilizadas para avaliar a eficácia da redução para cada ferramenta.

A eficácia da redução pode ser influenciada por diversos fatores. No caso da nossa ferramenta, foram considerados o tamanho do ficheiro de testes, o número de testes, o tempo de execução dos mesmos, a percentagem de cobertura total do código e a percentagem de cobertura de "ramos"no código. Por este motivo, decidiu-se adotar uma metodologia de *Multi-Criteria Decision-Making*, mais especificamente a *Analytic Hierarchy Process* que, segundo diversos critérios e valores de importância definidos entre eles, deduz a prioridade que cada critério deve ter ao tentar atingir um objetivo comum, isto é, que peso tem cada critério no cálculo da pontuação de cada ferramenta.

Após a finalização e aperfeiçoamento da ferramenta, foram realizadas experiências que nos permitiram analisar a eficácia de cada ferramenta. Dada a facilidade de configuração da ferramenta, foram efetuadas diversas análises às reduções efetuadas pelas ferramentas, alterando a importância de cada critério considerado, visando verificar de que maneira estes influenciavam a escolha da melhor ferramenta.

As pontuações de cada ferramenta de redução foram calculadas para seis cenários diferentes: um que replicasse as necessidades do "mundo real", dando portanto mais importância ao tempo de execução dos testes e cobertura atingida do que à dimensão dos testes em si; e um focando toda a importância para cada um dos sub-critérios definidos. Cada um destes cenários foi feito duas vezes - uma com os testes gerados pelo EvoSuite, e outra com os testes já existentes - com o objetivo de averiguar se as ferramentas de redução teriam um comportamento semelhante na presença de testes gerados automaticamente ou por humanos.

Ignorando a ferramenta de prioritização, e tendo em conta o facto de que a EvoSuite só poderia ser usada com testes gerados por si mesma, a ferramenta de redução que teve uma melhor pontuação foi a Testler, o que responde à nossa primeira questão do estudo.

Quanto à segunda questão, apesar dos resultados terem mostrado indubitavelmente que a Kanonizo obteve pontuações melhores que as outras ferramentas analisadas, esta é uma questão suscetível à influência de diversos fatores. No contexto das experiências

efetuadas, podemos dizer que a Kanonizo pode substituir qualquer uma das ferramentas de redução e, confiando nas pontuações, fazer um trabalho mais eficaz. No entanto, num contexto mais abrangente, torna-se difícil responder a esta questão sem considerar um número mais elevado de ferramentas de redução de testes. Para mais, dependendo do algoritmo utilizado na ferramenta de prioritização, a redução feita por nós pode não ter qualquer critério (no caso de algoritmos *random*), o que faria essencialmente com que estivéssemos a apagar métodos aleatoriamente, podendo causar grandes perdas em termos de cobertura de código.

Quando falamos de "substituir"uma ferramenta de redução por uma de prioritização com uma abordagem semelhante à nossa, é melhor utilizar algoritmos de prioritização que visem algum critério, como dar prioridade a testes que cubram, por exemplo, linhas de código ainda não cobertas, por exemplo, aumentando assim a probabilidade dos testes menos importantes serem redundantes e, consequentemente, apagados.

Esta ferramenta foi desenvolvida de modo a que fosse facilmente modificável e expansível, oferecendo assim uma maneira fácil para integrar novas ferramentas de redução de testes, permitindo realizar novas comparações à medida que estas ferramentas forem surgindo. Para além disso, destaca-se também a simplicidade da configuração dos critérios a ter em conta quando calculamos a pontuação de cada ferramenta, e o valor que estes têm em comparação com os outros, possibilitando facilmente o cálculo de pontuações tendo em conta diversos cenários.

Ao longo deste trabalho, o maior problema encontrado foi o estudo das ferramentas de redução. Para além de algumas não serem *open-source*, muitas das que eram não iam de encontro aos nossos requisitos para integração na ferramenta, quer por linguagem de programação, quer por abordagem seguida, e muitas vezes a documentação encontrada estava desatualizada ou errada, dificultando todo o processo.

Palavras-chave: Teste de software, Redução de testes, Otimização de testes

Abstract

With the growing need to assure the quality of the software created nowadays, it has become increasingly necessary to devote part of the development time (sometimes as much as half) to testing the developed code.

Despite the high importance of this task, it is an activity that is sometimes ignored and neglected, due to it being, occasionally, a monotonous and tiring job.

This is an area thoroughly investigated by researchers who try to automate some parts of this process, while also reducing the time and resources required for this task. In particular, we highlight the proposal of test reduction techniques and tools that implement them, with the goal of detecting unnecessary test cases without compromising the coverage given by the original test suite.

The main objective of this work consists in answering two questions: "What is the best Test Suite Reduction tool?" (according to some criterion) and "Can a Test Case Prioritization tool be adapted to effectively replace a Test Suite Reduction tool?". To answer these questions, we developed a framework that integrates a set of Test Suite Reduction and Test Cases Prioritization tools, with the possibility to integrate more, and that uses them, given a set of open source Java projects, to apply each of its techniques.

We then gather test execution data about these projects and compare them to compute a score for each Test Suite Reduction tool, which is used to rank them.

This score is achieved by applying a Multi-Criteria Decision-Making methodology, the Analytic Hierarchy Process, to weigh several chosen code metrics on the final score.

To answer the second question we integrated a Test Case Prioritization tool into our framework, with which we will get a list of the less important test cases, that we will then remove, simulating a reduction.

Keywords: Software testing, Test suite reduction, Test suite optimization

Contents

List of Figures

List of Tables

Glossary

Test Suite A set of Test Cases.

Test Case A set of instructions executed to verify a functionality of a software application.

Acronyms

AHP Analytic Hierarchy Process. , 5, 18, 20, 37, 41, 43, 44, 47, 57

ILP Integer Linear Programming. , 10, 13, 26

MCDM Multi-Criteria Decision Making. , 3, 5, 18, 20, 37, 57

TCP Test Case Prioritization. , 2, 3, 5, 14, 15, 17, 25, 31, 33, 39, 66, 69, 70

TSR Test Suite Reduction. , 2, 3, 5, 6, 9, 10, 13, 14, 17, 21, 25, 26, 28, 31, 33, 38, 39, 41, 46, 47, 48, 55, 63, 66, 69, 70

Chapter 1

Introduction

In this chapter, an introduction to the topic of this dissertation will be given, presenting the reasons that motivated its development and explaining what are the goals to be achieved as well as the contribution given by this work. The structure of the remaining chapters of the document is provided at the end.

1.1 Motivation

The inclusion of a testing stage in the software development process has proven to be a fundamental requirement for the industry over the years, and nowadays it takes up approximately 50% of the time and more than half of the development costs of a system [1], since it is this activity that will ensure the quality and reliability of our software. This stage is already defined in several software development models, such as the waterfall model or test driven development.

Poor management of this stage can - and most likely will - cause the software implementation to fail, i.e., to deviate from the expected behavior. The existence of such failures becomes more serious when we talk about critical software systems (hospitals, military, space exploration, ...), where the existence of problems in software may translate into major economic or commercial image damage, or even loss of human lives [2].

As the software is developed and changed, it may be necessary to implement new test cases, either to test new functionalities or to have test suites that offer a better coverage of the code. Testing allows us to assure the quality of our software during development, or after a modification [3] (e.g. implementation of a new functionality). In other words, after making any changes to the software, in addition to creating test cases that cover the newly written code, one must also run all the previously created test cases in order to ensure that the last changes did not have a negative impact on what was originally made and tested.

Obviously, it is to be expected that as the software grows, so do the test suites associated with it. This results in ever-increasing software under test and test cases to run, which translates into more time and more resources spent on this task, making regression testing

one of the most costly types of software testing [4]. These tests focus on verifying that a new functionality does not modify the expected behavior of the previously implemented functionalities.

For these reasons, regression tests are often research targets with the aim of developing methodologies to reduce their high costs. With this problem in mind, several methodologies began to emerge to try to cover the cost of this stage of development [5, 6, 7, 8].

One of these methodologies is the application of Test Suite Reduction (TSR), or Test Suite Minimization, techniques, which will be one of the focus of this work, and which aim to reduce the size of test suites, usually by identifying and eliminating redundant test cases, converting the original test suite into a reduced one [7], thus saving execution time and facilitating the maintenance of these tests, preferably without compromising their effectiveness in detecting bugs.

Another one of these methodologies, and also the focus of this work, although not so thoroughly, is the application of Test Case Prioritization (TCP) techniques. As opposed to TSR, Test Case Prioritization (TCP) will not reduce the suite, but will reorganize it into a more efficient one, which will prioritize the most likely test cases to identify a fault [9].

1.2 Goals

The main goal of this work is to carry out an experimental study that allows us to compare different techniques for reducing test suites along with some to prioritize test cases. In order to complete this goal, a set of tools was studied and a subset of those was selected to conduct this study. The criteria to filter these tools will be explained further ahead in this dissertation. Using these tools and test suites from several open-source projects, the efficiency of the reduced test suites will be measured - through a selected set of metrics - and compared between each reduced suite generated by each tool.

To simplify this analysis, and also as a goal of this work, a tool was developed that facilitates the execution of the previous selected tools, and provides a comparative report between the efficiency of the original and reduced test suites.

With the accomplishment of this work, we intend to answer two questions: First, which of the Test Suite Reduction tools that we used is the best, according to some set of criteria. Second, if we take a Test Case Prioritization tool and apply some kind of cut (for instance, erase the least important 30% test cases in a test suite), will the TCP tool be more efficient than the Test Suite Reduction ones?

1.3 Contributions

The main contribution of this work is an experimental approach that compares and evaluates a set of TSR and TCP tools, regarding their performance in the task of test suite

reduction.

This approach was also implemented in the form of an open source modular framework that integrates a set of predefined TSR and TCP tools that reduces the test suites of given projects. Using a code analysis tool, it then extracts a set of code metrics, which are then aggregated through a Multi-Criteria Decision Making (MCDM) methodology to ease the comparison and evaluation of the tools.

The framework offers the possibility to configure its execution, whether by easily adding or removing other available tools (as long as they respect the requirements and with a small programming effort), as well as the code metrics measured (as long as they are offered by the used code coverage tool).

1.4 Document organization

The remaining chapters of this dissertation are organized as follows:

- **Chapter 2 - Related work**: in this chapter, a state of the art analysis will be made, considering the related work, focusing on the techniques used to reduce test suites, as well as the categorization of such techniques and the main existing tools that implement them. In a more summarized way it will also be given an explanation about existing TCP techniques and tools, along with a brief explanation of the code metrics commonly used to study the efficiency of Test Suite Reduction, and a brief study on MCDM;

- **Chapter 3 - Selected frameworks and tools**: in this chapter, we detail the work done and problems found with each studied TSR tool and TCP tool. We also define the choices, and rationale behind them, to form the final set of tools to integrate in our framework;

- **Chapter 4 - Design**: in this chapter, an overview of the general approach of our framework and its evolution throughout time will be made. We will also detail the choices made regarding the implementation of our framework, such as the code analysis tool to choose, how to process the metrics measured and how to use them to achieve the answer to our goals;

- **Chapter 5 - Implementation**: in this chapter, the implementation of our framework will be described, explaining its structure and summarizing each part of the code and its purpose, as well as the changes we made to the necessary integrated tools;

- **Chapter 6 - Results**: in this chapter, we will explain the conducted experiments, as well as the results obtained by them and a discussion regarding them;

- **Chapter 7 - Conclusion**: in this chapter, a summary of the results will be given, as well as the possible future work to be made, related to this work, and the topic it presents. We also describe the problems found during this work.

Chapter 2

Related work

The study of the state of the art carried out in the first months of the work focused mainly on three aspects: the proposed TSR techniques and the existing tools that implement some of these techniques, an overview of existing TCP techniques, and the metrics that could be used to evaluate the efficiency of these techniques.

A brief study on MCDM and some of its approaches, specifically the Analytic Hierarchy Process (AHP), was also conducted and will be described in the end of this chapter.

2.1 Test Suite Reduction

Test Suite Reduction allows a more efficient and easier test suite maintenance, which in turn reduces the cost of the testing phase of software development, although in this process the ability to detect faults may be compromised [10].

These techniques work by identifying and erasing test cases that became obsolete or redundant, usually by some change in the code, e.g. the implementation of a new functionality or the modification of an old one [5, 11, 12].

According to M. Harrold et al. [11], the problem of Test Suite Reduction can be formally defined as the following:

Given:

- TS - a test suite;
- r_1, r_2, ..., r_n - set of test case requirements that must be met to ensure the desired test coverage of the program;
- T_1, T_2, ..., T_n - subsets of TS, each associated with one of the r_i, with $1 \leq i \leq n$, such that any test case $t_j \in T_t$ can be used to test r_i.

Problem:

- Find a set of test cases in TS that satisfy all r_i.

2.1.1 Test Suite Reduction techniques

There are several techniques proposed and thoroughly studied [3, 4, 5, 8, 11, 13].

Binkley et al. [3] suggest that a TSR technique should have the following characteristics:

- **Inclusiveness**, i.e. the reduced test suite must provide the same test coverage as the original test suite, according to some criteria.
- **Precision**, i.e. the algorithm should be able to find a subset of minimum, or approximate minimum, test cases.
- **Efficiency**, i.e. the reduction is only worth if the cost of the analysis needed to perform it is less than what we saved by reducing the test suite.
- **Generality**, i.e. the technique should be applicable to a comprehensive set of programs.

2.1.2 Classification of techniques

According to Alian et al. [13], we can classify TSR techniques into eight categories:

- ***Requirement based***

  Some of the proposed techniques solve the problem of TSR by looking at test requirements rather than test cases. These techniques usually provide a good reduction in the percentage of redundant tests, but on the other hand they are generally more time consuming.

  Fraser et al. [14] propose a technique of this category that consists of choosing subsets of test cases that meet the requirements. A model-checker then receives as input a finite state model and a temporal logic property that will result in a counterexample if the property is not met.

  Chen et al. [15] propose a technique that uses a requirement relation graph to optimize the set of requirements, by applying graph contraction methods.

- ***Genetic algorithm***:

  As already mentioned, one of the software testing problem lies in the effort, time and cost required to develop good test suites. In an attempt to reduce this problem, the use of evolutionary algorithms, in particular genetic algorithms, has been researched in order to automatically generate test cases.

  The techniques of this category take, as the initial population, the existing test cases, using them to create the next generations by using mutation, crossover and fitness functions that use the information collected after the tests are executed (for example, information on test coverage) and generate the next populations until they find the minimized test suite. Although this method reduces the number of test cases and the runtime of the test suite, there is still some work to be done in relation to the ability to detect faults [16].

  Nachiyappan et al. [17] propose a genetic algorithm technique, in which the initial population is based on test history, i.e. the tests' previous execution time and coverage, and the fitness function depends on these values.

  You and Lu [18] propose a technique that reduces a test suite taking into account the tests' execution time that follows the generic implementation of a genetic algorithm. The initial population is generated according to a proposed representation scheme, and parent selection, crossover and mutation is then applied as to generate better candidates, according to a fitness function that uses tests' execution time as an evaluation.

  Mohapatra and Pradhan [19] propose a genetic algorithm technique that generates the initial population by using a binary matrix with columns representing test cases and rows the test requirements to be met. It then uses single point crossover and mutation until it can achieve a subset of the initial test suite that covers all the specified requirements.

  Ma et al. [20] propose a technique that uses test history according to covered blocks of code, i.e. sequence of statements, to generate the initial population. Selection, crossover and mutation are then applied to evolve the individuals until one who achieves the desired coverage with minimal cost is found.

- ***Clustering***:

  The techniques of this category, as the name implies, take advantage of well-known clustering techniques, but lack the ability to detect faults.

  Wang et al. [21] propose a technique that divides test cases into clusters according to their similarity in profiling. It provides improvements by making three types of profiles: File execution sequence, function call sequence, and function call tree.

Subashini and JeyaMala [22] propose the use of data mining approach on clustering, joining the redundant and similar test cases into clusters, and reducing the testing effort by testing these clustered test cases in turn of the whole test suite. A similar approach is also proposed by Parse et al. [23].

- ***Fuzzy logic***:

  The techniques in this category are based on the use of fuzzy logic, i.e., logic based on true values (as opposed to boolean logic - 0 or 1), thus allowing to optimize test suites not only for one objective, but several (multi-objective selection criteria). These techniques reduce the size and execution time of the tests.

  Haider et al. propose several techniques using fuzzy logic. In [24] the technique consists of an expert system that uses a technique based on a defined objective function, similar to human judgement, using a classification based on fuzzy logic.

  In [25] they study some computational intelligence based approaches such as evolutionary computation, fuzzy logic and neural networks, and conclude that only fuzzy logic is adequate for test regression. Finally, in [26] they propose an approach that extracts the optimization parameters from the test cases, develops a model for the problem under testing and optimizes the test suite using fuzzy logic.

- ***Coverage based***:

  Something that should always be considered when reducing a test suite is the coverage provided by the tests. These techniques ensure that the given tests, even when reduced, cause the program to be tested to run according to most of the run paths (paths that the program flow takes from start to finish) defined for that program. The rate of reduction of test cases is quite high, but the type of coverage chosen may be inefficient depending on the size of the software under test.

  Murphy [27] proposes an example of a coverage-based technique, which consists of an algorithm that covers all attainable states. It focuses on path coverage as it generates test cases by accessing the source code. It then identifies test cases that cover all sub-paths in the program based on code implementation, and removes any test cases that cover already covered sub-paths.

  Pringsulaka and Daengdej [28] propose a technique called coverall algorithm, that uses algebraic conditions to define the values of some variables. This will limit the values within a definite range, which in turn results in fewer test cases to process.

  Roongruangsuwan and Daengdej [29] propose a technique using case based reasoning [30]. This technique defines three methods: Test Case Complexity for Filtering, that finds a complexity for each test case, determined by the number of test cases in a test suite related to the average, and then removes the test cases with fewer

complexity; Test Case Impact for Filtering, that is similar to the previous one, with the difference that it measures the impact of each test case, which is related to the faults found by a test case; and Path Coverage for Filtering, which removes the test cases that achieve less coverage.

Khan and Nadeem [31] propose a technique called TestFilter. This technique finds a weight for each test case, which is related to the requirements they cover. The test cases are then assigned to the reduced test suite, starting from the one with the higher weight value, until all requirements are met.

- ***Program slicing***:

  This methodology works by identifying parts of a program that are relevant to the values of a predefined set of variables at some point in the program. A slice of the program is built by removing all parts of the program that are irrelevant to these values. These techniques reduce the size of the test suite and its execution time, but there is still some study to be done on the ability to detect faults.

  Arasteh [32] proposes an example of a technique that uses program slicing, which focuses on parts of the code that have a significant impact on its output, while those parts of the program that have no effect are eliminated from the testing process.

  Binkley [33] proposes an algorithm that identifies test cases that produce the same output when executed on a given program, before and after a modification, as well as the existing test cases that test the new components added after a modification, thus avoiding the rerunning of tests that produce the same output and providing a more efficient test suite.

- ***Greedy algorithm***:

  Techniques in this category generally select the test case that satisfies the maximum number of unsatisfied requirements, and make a random choice if there is a tie. Provides a significant reduction in test cases, although the random choice on those specific situation is not ideal.

  Tallam [34] proposes a technique in which the test cases are considered objects and the requirements their attributes. Using concept analysis frameworks, the maximum groupings of objects and attributes are identified and called contexts. Reduction rules are used to reduce the size of the context table by applying object and attribute reduction rules.

  Xu et al. [35] propose a weighted greedy algorithm. This algorithm determines if a test case covers all of the testing requirements, selecting it for the reduced suite if it does. If not, it eliminates the redundant test cases, updating the set of test cases and the uncovered requirements until all requirements are met. Zhang et al. [36] apply this technique to JUnit test suites while studying a set of existing TSR techniques.

- ***Hybrid algorithm***:

  Some algorithms try to reduce the number of test cases using more than one type of technique, for example, the combination of genetic algorithms and the greedy algorithm. These techniques provide a significant reduction in the number of test cases, but are also highly complex.

  Suri [37] proposes a technique that combines genetic algorithms and colonies of artificial bees. Colonies consists of three groups of bees: employed, onlookers and scouts. By using bees as agents, the algorithm can exploit the minimum set of test cases.

  Sampath et al. [38] suggest the standardization of the use of hybrid criteria by proposing three ways to combine criteria: Rank, which combines criteria by order of importance and applies them in series, i.e. when the first criteria fails, the second one is applied; Merge, which combines criteria simultaneously; and Choice, which applies the criteria in a series but, unlike rank, follows some criterion for the selection, for instance, the code coverage.

  Yoo and Harman [39] introduce a hybrid multi-objective genetic algorithm. It uses a modified version of the greedy algorithm that uses knowledge about the cost of the test cases regarding computational effort and statement coverage.

2.1.3 Test Suite Reduction tools

In order to ease the whole method of Test Suite Reduction, several researchers proposed and developed tools and frameworks that allowed to speed up this process [5].

Tool classification

With the increase of research in this area, and the constant development of TSR tools, it became necessary to have a way to classify them.

Khan et al. [5] proposed a categorization of the existing tools, based on the following parameters:

- ***Approach type***

  This parameter represents the main category of approaches taken to conduct reduction on test suites, and is divided into four attributes: *coverage-based*, *search-based*, *Integer Linear Programming (ILP)* and *Similarity-based*.

  Coverage-based approaches are based on the use of greedy algorithms in order to select test cases that cover a maximum number of instructions from the program to be tested. In search-based, search algorithms, such as genetic algorithms, are used to find several test cases from an initial population. ILP based finds a minimal

solution to the problem of TSR based on the defined objectives and constraints. Finally, the similarity-based try to solve this problem by making use of similarity matrices for all pairs of test cases.

- ***Testing paradigm***

  Represents the programming paradigm associated with the language in which the tool was implemented. It is divided into three attributes: *structured*, *object-oriented* and *aspect-oriented*.

  The structured ones were developed using a language that allows a type of structured programming, using the flow control and divides the code by procedures and functions. Object-oriented were written in languages that allow the use of object-oriented programming, which generally treats the data as if they were objects. Finally, aspect-oriented were implemented using languages in which aspect oriented programming can be practiced, which aims to increase and improve the modularity of the written code.

- ***Optimization type***

  Represents the amount of tool optimization goals, which can be either *single-objective* or *multi-objective*. Single-objective only focuses on generating a reduced test suite or calculating the effectiveness of fault detection. On the other hand, the multi-objective ones focus on both objectives.

- ***Coverage source***

  Splits into *source code* and *test execution profile*. Source code determines the reduced suite by choosing test cases that cover as many elements of the program's source code as possible. The test execution profile captures the test execution profiles by running the test suite and then the tool determines the representative test cases based on the coverage score achieved by the execution profiles.

- ***Execution platform***

  Defines the number of servers required to run the tool to determine the reduced suite. This parameter is divided into *single server* and *multiple server*. The main improvement brought by multiple servers is the ability to divide a problem into smaller problems, in this case, divide a test suite into small test suites that are then run on each server, accelerating the process of reducing test suites.

- ***Computational mode***

  Represents the type of processing supported by the tool. Splits into *online* and *offline*. The online type receives several small test suites sequentially to produce the reduced suite, while the offline type receives the entire suite in order to find an optimal solution.

- ***License type***

  Defines the type of license of the tool, which can be *commercial*, *academic research* or *free*. Commercials can be purchased through some form of payment, academic research is developed by laboratories or academic research groups and is usually accessible in the academic field and free is available to any user, without any kind of payment.

- ***Evaluation***

  Represents the type of evaluation used to evaluate the tool and is divided into *internal* and *external*. While the internal ones were evaluated by their designers and developers, in the same environment in which they were developed, the external ones are evaluated outside their development environment.

- ***Customizability***

  Defines the ease of any user in modifying the tool and categorizes itself in *full*, *partial* and *in support for customizability*. The tools considered full allow any type of change made, and are usually those that provide the source code. The partial only admit minor changes, for example the integration of the tool in other developed software. No support does not allow any kind of modification.

- ***Support***

  Represents the availability of support provided for the tool, i.e. the existence of documentation, or whether the executable or the source code is available.

In addition to these parameters, the tools can also be organized into five different classes, which defines the paradigm where these tools will be most useful:

- ***Randomized unit testing***

  Tools that focus on unit tests, in which random elements are involved in the selection of parameters or methods to be tested, which allows the generation of different test cases in an easy and fast way, facilitating the discovery of faults in the code.

  This methodology is propitious to the generation of very similar test cases, creating some redundancy and therefore raising the need to reduce them.

- ***User session testing***

  In case of software with a high degree of interactivity, which requires constant interaction from the user, as is the case of web applications, it becomes necessary to take a different approach when we want to test them.

  In these cases, the users' session data, for example, a sequence of users' actions, is usually used to test the software. However, it is very likely that this data is highly redundant.

  These tools intend to reduce this degree of redundancy present in this data.

- ***Re-targeted compilers testing***

  When there is a need to redesign a processor, it is also indispensable to build new compilers for these processors. An efficient approach to this task is reusing the existing compilers. Generally, a large number of test cases are generated using source code employing grammatical coverage criteria.

  The tools in this class focus on back-end testing of a compiler to reduce the test suite.

- ***Integer linear programming***

  This approach allows for the determination of optimal solutions based not only on the reduction of the test suite while maintaining the coverage of the original suite, but also on other objectives, such as minimizing the suite execution time and maximizing the fault detection capability.

  The tools of this class formulate the problem of Test Suite Reduction through a ILP problem.

- ***Automated fault-localization***

  These tools focus on finding the exact location of a gap in the software to be tested, trying to improve the effectiveness of this process.

Existing tools

Table 2.1, based on the one prepared by Khan et al. [5], presents some developed tools and frameworks and their taxonomy, i.e., the class and some parameters of each, considered the ones with most potential to use in this work.

The support and customizability parameters were merged and presented in the 'availability' column, considering that if a tool has full customizability, it is because its source code (Src) is available. If it has partial customizability, it has an executable (Exe) available and if it has no customizability, then it has no source code or executable available. The

Class	Tool	Approach type	Availability		
			Doc	Src	Exe
Randomized Unit-testing	**ATAC** [40]	Coverage-based	✓	✓	✓
	Rostra [41]	Coverage-based	✗	✗	✗
	Raspect [42]	Coverage-based	✗	✗	✗
	RUTE-J [43]	Coverage-based	✓	✓	✓
	Randoop [44]	Coverage-based	✓	✓	✓
	GenRed [45]	Coverage-based	✗	✗	✗
	JTOP [46]	-	✓	✗	✓
	TOBIAS [47]	Coverage-based	✓	✗	✓
	TEMSA [48]	Search-based	✓	✗	✓
	Open-SourceRed [49]	Coverage/Search-based	✓	✓	✓
	EvoSuite [50]	Coverage/Search-based	✓	✓	✓
	Testler [51]	Similarity-based	✓	✓	✓
User session testing	UsbRed [52]	Similarity-based	✗	✗	✗
	CPUT [53]	Coverage-based	-	-	-
Retargeted compilers	RTL [54]	Coverage-based	-	-	-
	PLOOSE [55]	Coverage-based	-	-	-
Integer Linear Porgramming	**MINTS** [56]	ILP-based	✓	✓	✓
	EDTSO [57]	ILP-based	-	-	-
Automated fault-localization	SrTC [58]	Coverage-based	-	-	-
	JINSI [59]	Coverage-based	-	-	-
	GZoltar [60]	Coverage-based	✓	✓	✓

Table 2.1: Summary of the taxonomy of some existing TSR tools.

documentation (Doc) column represents the availability of any written documentation for each tool.

The tools in bold represent those that were primarily selected for use in the development of this work. The criteria for this selection will be elaborated further in this dissertation.

The ones that do not have any information in the availability column, marked with an hyphen (-), represent the situation where no documentation has been found, or was found written in a foreign language (other than English), or no executable or source code capable of producing an executable binary was found, whereas the (✗) means it was not available and (✓) means the opposite.

2.2 Test Case Prioritization

As opposed to TSR, Test Case Prioritization techniques do not discard test cases. However, they reorganize them in a more efficient order that can achieve some goal related to the software under test in the fastest way possible [6, 7, 61, 62].

These goals can be quite diverse, and depend on what the testers want to achieve when

they execute their tests. Usually, some of these objectives are: increasing the rate of fault detection, increasing the rate of detection of high-risk faults, increasing the probability of finding bugs related to specific changes in the code, increasing the coverage of the software under test or increasing confidence in the reliability of the system [63].

Rothermel et al. [6] define the TCP problem as such:

Given:

- TS - a test suite
- PT - the set of all permutations of TS
- f - a function from PT to $\mathbb{R}$, the set of Real numbers

Problem:

- Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

2.2.1 Test Case Prioritization techniques

Elbaum et al. [9] suggest and use a set of TCP techniques, taking advantage of some already defined by Rothermel et al. [6, 63].

These techniques are divided in three groups: the comparator group, the statement level group and the function level group.

Table 2.2 is based on the one from [9], and shows a brief description of each technique. After the table, each group will be described, as well as a more extensive description of each technique.

Group	Name	Prioritization goal
Comparator	random	random order
	optimal	maximize rate of fault detection
Statement-level	st-total	maximize coverage of statements
	st-addtl	maximize coverage of statements not yet covered
	st-fep-total	maximize probability of exposing faults
	st-fep-addtl	maximize probability of faults, considering previous test cases
Function-level	fn-total	maximize coverage of functions
	fn-addtl	maximize coverage of functions not yet covered
	fn-fep-total	maximize probability of exposing faults
	fn-fep-addtl	maximize probability of faults, considering previous test cases
	fn-fi-total	maximize probability of fault existence
	fn-fi-addtl	maximize probability of fault existence, considering previous test cases
	fn-fi-fep-total	maximize combined probabilities of fault existence and fault exposure
	fn-fi-fep-addtl	maximize combined probabilities of fault existence and exposure, considering previous coverage
	fn-diff-total	maximize probability of fault existence
	fn-diff-addtl	maximize probability of fault existence, considering previous test cases
	fn-diff-fep-total	maximize combined probabilities of fault existence and fault exposure
	fn-diff-fep-addtl	maximize combined probabilities of fault existence and exposure, considering previous coverage

Table 2.2: Test Case Prioritization Techniques.

- **Comparator group**

  The techniques in this group were considered in [9, 63] to allow a comparison for the study being conducted in these papers. These are the simpler techniques.

  - *random*: Not as much of a technique as the others, this one is simply a random ordering of the test cases, with no goal in mind, and only serves as an experimental control.
  - *optimal*: This technique is also used as a comparison, and consists of ordering the test cases in the optimal order to detect faults earlier. This was possible in [9] because the programs used had known faults and it was known which test cases could trigger each fault. In the real world, when working with software for which there are no known faults, such technique is near impossible to replicate.

- **Statement Level group**

  These techniques focus on coverage and fault detection regarding the statements of the software under test, and have goals related to those.

  - *st-total*: This technique tries to order the test cases in an order that will give the most coverage of the program's statements in the less time possible. To achieve this, it uses program instrumentation.
  - *st-addtl*: This technique is similar to the previous one, but uses information about the coverage obtained so far to determine the statements that were not yet covered. To achieve this, a test case that covers the most statements is selected, the not covered statements are updated, and it repeats this process until all statements are covered by at least one test case.
  - *st-fep-total*: This technique tries to order the test cases according to an award value, assigned to each test case, defined by Elbaum et al. [9]. This award value equals the Fault-Exposing Potential of the test case, and is the sum of all mutation scores, which in turn is the ratio, for each test case, of mutants of each statement exposed by that test case, to total mutants of that statement.
  - *st-fep-addtl*: Similar to the previous technique, this one takes into account each test case already executed. When a test case is executed, the award values of other test cases that cover the same statements as the former. After this, the next test case is selected and the process repeats itself until all test cases have been ordered.

- **Function Level group**

  These techniques are similar to the ones that belong in the statement level group, but now regarding the functions of the software under test.

  As the number of functions in a software will usually be much smaller than the number of statements, these techniques are less expensive than the ones in the statement level group, although they may not be the most efficient in exposing faults.

- *fn-total*: This technique, similarly to *st-total*, orders the test cases according to the total number of functions executed by each of them.
- *fn-addtl*: Similar to *fn-total*, with the addition of considering the functions that are yet to be executed, analogously to *st-addtl* with the statements.
- *fn-fep-total*: Similar to *sn-fep-total*, but adapting the fault-exposing potential to the function-level in a method analogous to the one in *sn-fep-total* but using functions instead of statements.
- *fn-fep-addtl*: This technique enhances *fn-fep-total* in the same way *sn-fep-addtl* enhances *sn-fep-total*.
- *fn-fi-total*: This technique orders the test cases similarly to *fn-total*. For each test case, a sum of Fault Index is calculated and assigned to it, which then serves as a comparator to sort the test cases. Fault index is used to measure the fault proneness of a function in the program.
- *fn-fi-addtl*: This technique is analogous to *fn-fi-total*, but uses a set of functions that were previously called by executed test cases. When all the functions are in that set, it is emptied. After that, the sum of the fault indexes are calculated for each test case, not counting the ones from functions that are on the set of covered functions.
- *fn-fi-fep-total*: This technique is a combination of *fn-fi-total* and *fn-fep-total*, as the method is the same and is considered the product of the fault index and fault-exposure potential, which is then summed for each test case and used in the sorting.
- *fn-fi-fep-addtl*: Analogous with the previous technique, with the difference that the values for the functions already called are reset.
- *fn-diff-total* / *fn-diff-addtl* / *fn-diff-fep-total* / *fn-diff-fepp-addtl*: These techniques works the same way as *fn-fi-total* / *fn-fi-addtl* / *fn-fi-fep-total* / *fn-fi-fep-addtl*, respectively, but use data collected by the Unix command diff, when applied to each function of the program in two different versions.

2.2.2 Test Case Prioritization tools

Since the main focus of our work was the TSR methodology, the study conducted on TCP tools was less thorough. As such, we only considered two TCP: Kanonizo [64] and Ekstazy [65]. Seeing that Ekstazy was only available as binary and Kanonizo provided four different techniques of prioritization, we opted to pick the latter, which will be further described in the next chapter.

2.3 Code metrics

The metrics that may be considered most important in the case of TSR are the size of the reduced suite, and the ability to detect faults [36].

The size of a test suite, which allows us to define the percentage of reduction in its size, can be defined, for instance, in relation to lines of code, which is regarded as the most commonly used code metric [66].

The fault detection capability of a test suite, which is useful for comparing the number of faults found by the original suite and the reduced one, can be measured to analyze the efficiency of the reduced test suite in favor of the original one.

In addition to these metrics, it may also be interesting to analyze some metrics related to the source code as well as the execution of the test suite, to study its efficiency, for instance the number of methods covered per class in comparison to the total number of methods per class, or the execution time of the test suite.

2.4 Multiple-Criteria Decision-Making

Multi-Criteria Decision Making (MCDM), or Multi-Criteria Decision Analysis, is a discipline of Operational Research that defines methods considering several criteria to achieve a solution to a problem, generally the choice of the best alternative among many. Although, above this, it is a specific perspective to deal with specific problems [67].

According to Belton et al. [68], it seeks to integrate objective measurement with value judgment and make subjectivity explicit. In other words, it helps us structure and understand the problem, and its priorities and objectives.

There are several methods of multiple-criteria decision-making [69], but we focused on the one that would be used in this work: the Analytic Hierarchy Process [70].

2.4.1 Analytic Hierarchy Process

The Analytic Hierarchy Process is a method that aims to structure the problems' criteria hierarchically. As Figure 2.1 depicts, and Saaty [70] denotes, the general decision problem will descend from a main focus or objective, which is then divided in criteria, which in turn are divided into subcriteria, that define the aspects one should have in consideration when making the decision, and lastly, the alternatives taking into account. The goal of this process is to obtain the priorities that each subcriteria will have when taking the main focus into account.

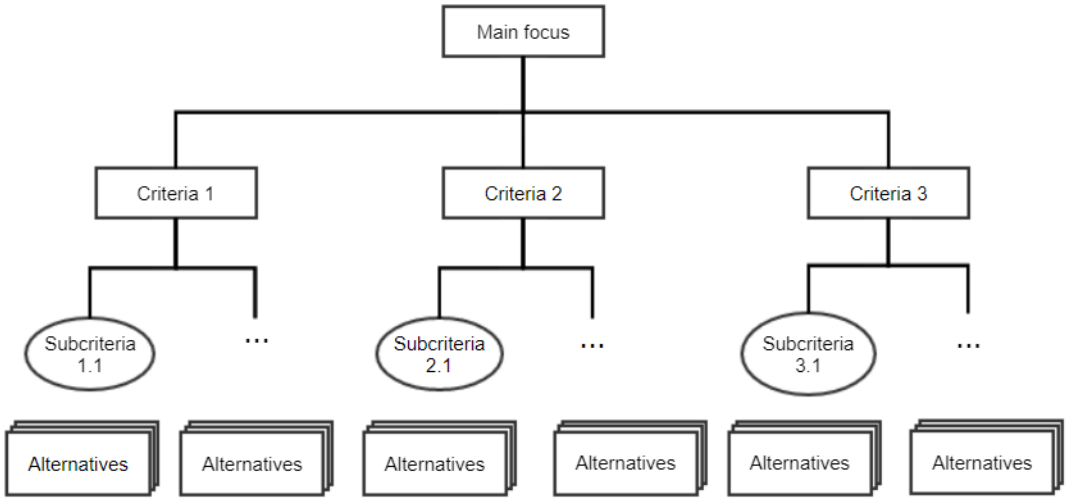

Figure 2.1: General hierarchical model.

Saaty [71] states that the steps to obtain these priorities are, briefly speaking, as follows:

- Define the problem;
- Structure the decision hierarchically;
- Construct a set of pairwise comparison matrices to obtain the priorities;
- Use the priorities of each level to weigh the priorities of the level immediately below. Repeat this process until there are no more levels.

The pairwise comparison matrices are build by assigning to each column i and each row j one of the criteria (in one level), where the value (i, j) will represent the importance of criteria i in accordance to criteria j. Naturally, when $i = j$ that value is 1.

According to [71], to compare two criteria i and j, one must answer the question "How much more is the i criteria important than the j criteria?". The values should be assigned in accordance to the ones shown in Table 2.3, based on the ones from [71, 72].

Once one has the pairwise comparison matrix for a set of criteria, the next step would be to find the geometric mean for each row, sum up all of them, and divide each by the total sum, which will give us, per row, the priority of the criteria assigned to that row, i. e., the weight one specific criteria has on the decision.

2.4.2 Normalizing values

Since the criteria can be of any nature, it is natural that the values assigned to each one can have different units of measure, and can have different orders of magnitude. As such,

Intensity of Importance	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance	Experience and judgement slightly favour one activity over another
5	Strong importance	Experience and judgement strongly favour one activity over another
7	Very strong or demonstrated importance	An activity is favoured very strongly over another, its dominance demonstrated in practice
9	Extreme importance	The evidence favouring one activity over another is one of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values	
Reciprocals of above (1 / x)	If activity i has one of the above non-zero numbers assigned to it when compared with activity j, then j has the reciprocal value wen compared with i	A reasonable assumption
1.1-1.9	If the activities are very close	May be difficult to assign the best value but when compared with other contrasting activities the size of the small numbers would not be too noticeable, yet they can still indicate the relative importance of the activities

Table 2.3: Fundamental scale of importance.

it is essential to have a way to normalize these values, i.e., transform them into a common numeric range, so that we can aggregate them into a final score [73].

There are some different normalization techniques that can be applied to MCDM, and therefore to AHP [73, 74, 75].

According to [73], the best technique to use in the AHP, is the Linear: max, followed by the application of Linear: sum to normalize the values, with their formulas for benefit and cost criteria presented in Table 2.4.

Technique	Condition of use	Formula
Linear: Max	Benefit criteria	$n_{ij} = {}^{r_{ij}}/_{r_{max}}$
	Cost criteria	$n_{ij} = 1 - {}^{r_{ij}}/_{r_{max}}$
Linear: Sum	Benefit criteria	$n_{ij} = {}^{r_{ij}}/_{\sum_{i=1}^{m} r_{ij}}$
	Cost criteria	$n_{ij} = \frac{{}^{1}/_{r_{ij}}}{\sum_{i=1}^{m} {}^{1}/_{r_{ij}}}$

Table 2.4: Normalization techniques.

We can describe a benefit criteria as one in which an higher value means an higher reward, and a cost criteria the opposite: an higher value means a lower reward.

Chapter 3

Selected frameworks and tools

When the related work study was finished, there was a need to filter out the tools and frameworks that were not usable in this work.

This chapter will explain the rationale behind the chosen tools, describe the work made with each one of those tools, along with the difficulties found.

3.1 Initial set of Test Suite Reduction tools

As Table 2.1 shows, twenty-one TSR tools were initially considered for this work. These tools were part of the state of the art study phase, and would then be integrated into the developed framework, so we tried to thouroughly study each of them, to learn how they worked and how could we integrate them in our framework.

Consequently, the tools in which developers did not provide any executable were discarded, as well as those in which no information about that was found.

As it could be necessary to change these tools, to facilitate their integration into the framework to be developed, it would also be important to have access to their source-code, so all those tools that did not have available source code were also discarded,

Taking these decisions into account, the tools initially selected to be integrated in the framework were ATAC [40], RUTE-J [43], Randoop [44], Open-SourceRed [49], EvoSuite [50], Testler [51], MINTS [56] and GZoltar [60].

It is important to notice that, although only those that provided source code were considered, the tools that provided an executable were also tested, to determine if they could be useful for this work without undergoing changes, these being JTOP [46], TOBIAS [47] and TEMSA [48].

3.1.1 ATAC

ATAC is a tool developed by Horgan and London [40]. This tool analyses the coverage achieved by tests on C and C++ programs, which helps its user to program a more complete test suite. This is achieved essentially by three steps: the instrumentation of the

source-code to be tested, which is achieved via the ATAC compiler, atacCC, with the goal to gather several data regarding the program execution; the execution of the tests, which will allow the information gathering; and the coverage analysis, which will use the gathered information to provide the tester with useful information about the executed tests, such as line and branch coverage.

Since our interest is the reduction of a test suite, and not exactly test management, we can replace the last step with the test minimization one, using the files generated by the execution of the tests to try and present a minimum list of test cases that achieve the maximum coverage possible.

Figure 3.1 illustrates the whole process. The elements that belong to the instrumentation step are underlined in green, the ones that belong to the execution of tests in blue and the ones from the coverage analysis and test minimization in red.

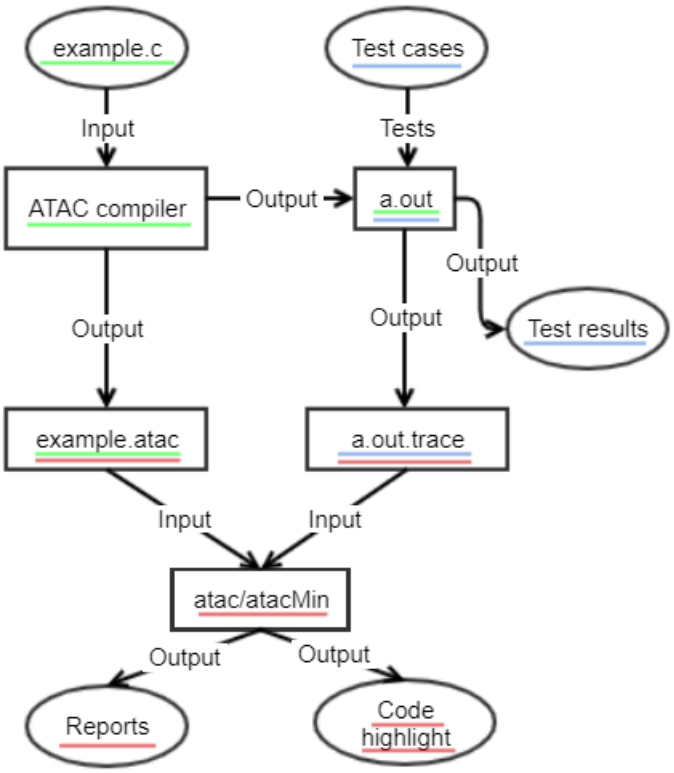

Figure 3.1: ATAC workflow.

Work done

To test this tool, its tutorial[1] was followed. It consists of a small word counting program written in C that receives text documents - to count the words from - as input. The input used was fairly simple. It consisted of three text documents provided with the source code of ATAC.

[1] https://invisible-island.net/atac/

As Figure 3.1 depicts, we first compiled the program using the ATAC compiler, and then executed the program with the input given. This generated a trace file, as well as .atac files for each source code file, which were next used as input to the command *atac* to generate the report that showed us how good the tests were, in terms of coverage, C-uses and P-uses. If given as input to the command *atacMin*, a minimal set of test names is given as output, that ideally has the same coverage as all the tests, thus discarding the redundant ones.

Problems

As we will see along this chapter, in the end the overwhelming majority of the tools were written in Java, and worked with JUnit tests, and as such we discarded ATAC from our work, as it was developed in C.

3.1.2 RUTE-J

RUTE-J, Randomized Unit Testing Engine for Java, was developed by Andrews et al. [43] in an effort to solve some of the most common problems in randomized unit testing, such as the correct definition of arguments, whether they are scalar or complex, and specifying correct behaviour of tests.

To work with RUTE-J, the user must write a Test Fragment Collection, which is essentially a java class that extends the RUTE-J class TestFragmentCollection that will contain test fragments, methods whose names have "tf_" as prefix and that denote a base for the test cases that will be generated by RUTE-J (generally, one test fragment per each method under test).

Once we have the Test Fragment Collection set up, we must run RUTE-J UnitDriver, which in turn will initialize a graphical user interface, where the user can generate test cases for the executed test fragment collection. The interesting part here is when RUTE-J finds a failing test case and halts the generation, giving the user the opportunity to minimize the failed test case, discarding the unnecessary statements to achieve the fail state.

Work done

To test RUTE-J, we followed the tutorial of RuteMoneyTest.java provided in the RUTE-J 1.2 source code, with the explanation in its readme file. We executed RUTE-J on RuteMoneyTest, a Test Fragment Collection, and were able to randomly generate test cases for the test fragments provided in that class, in which we see the name of the test fragment (without the prefix) and the parameters used, and if it passed or not, as shown in Figure 3.2.

Problems

Although the approach to minimize the failing test cases provided by RUTE-J was interesting, the extra effort it needed, such as the writing of a Test Fragment Collection, in other words, the impossibility of using existing test suites, made us discard this tool in the end, even though there was some thought on a possibility of somehow converting a standardized JUnit test suite into a Test Fragment Collection.

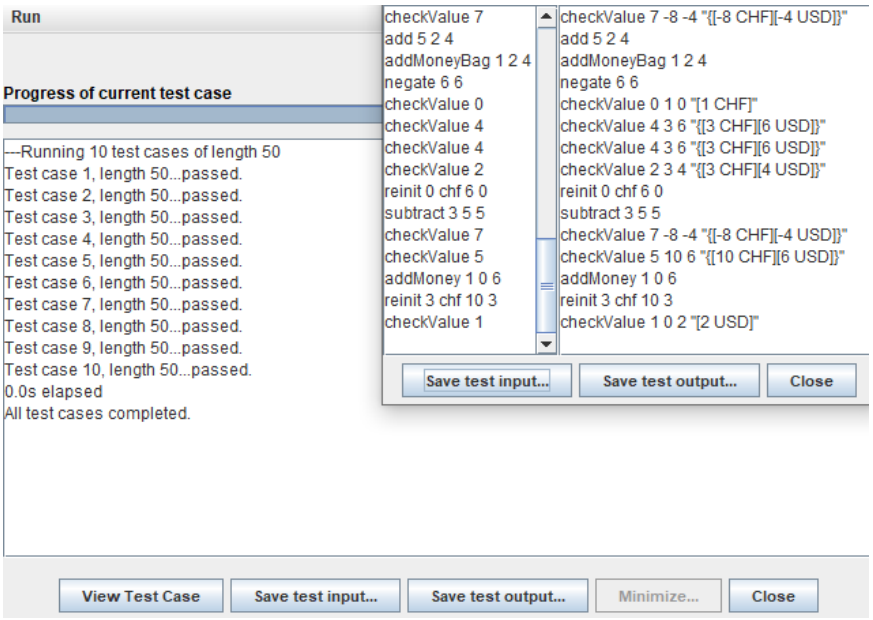

Figure 3.2: RUTE-J Graphic User Interface.

3.1.3 Randoop

Randoop was developed by Pacheco and Ernst [44], and is a tool that generates JUnit test cases and also provides the reduction of failing test cases.

To generate test cases, Randoop uses feedback directed random testing [76], a technique that gathers feedback from the execution of tests as they are generated, so that it can generate better and more reliable tests. The functionality that is of interest to us is the minimization of failing test cases, in which Randoop receives as input a JUnit test suite and tries to reduce every test that fails by analyzing each of its statements, and then providing a simplification to it, not compromising the output of the test.

The simplifications made to a statement can be, at least, one of the following:

- If a statement is represented by null, it is removed;
- Replacing the right hand side expression with 0, false or null, according to the left hand side expression;

- Replacing the right hand side expression with a calculated value obtained from a passing assertion;
- Removing the left hand assignment of a statement.

Given a JUnit file suite.java and the class files of the units tested by *suite*.java, we can use the Randoop minimize command to reduce the tests in this file, which will be saved in a new file named *suite*Minimized.java, so that the original tests are not lost.

Work done

We tried the minimize command with the test suites provided with lambdaj2.4.1[2] project and got a minimized suite for each JUnit file. There were some changes we had to make to the Randoop source code, which will be discussed in Chapter 5.

Problems

In some of the minimized tests, Randoop erased important "import" statements, which caused some compilation errors in the minimized suite.

3.1.4 Open-SourceRed

Open-SourceRed is a framework developed by Kauffman and Kapfhammer [49] that implements TSR and TCP algorithms. It consists of two tools, Proteja and Modificare.

Proteja is written in Java and collects coverage information on some software by running its JUnit test suites, outputting some coverage reports and files, which are then used as input in Modificare, to run the reduction and prioritization algorithms.

Modificare is written in R and reads the reports generated by Proteja to apply one of the six implemented TSR and TCP algorithms: random, adaptive random, greedy, hill climbing, simulated annealing, and genetic.

Work done

We followed the tutorials provided in the readme files of Proteja[3] and Modificare[4], using the example Sudoku project provided in the source code of Proteja. First we had to configure Proteja, specifying the source code and test suites, then waiting for it to run and use its output as input in Modificare. The algorithm suggested in the tutorial was the random.

In the end, Modificare presented a list with the test cases reorganized, which in turn could be saved into a file that could again be used in Proteja to run again and gather new coverage information.

[2] https://github.com/mariofusco/lambdaj
[3] https://github.com/kauffmj/proteja
[4] https://github.com/kauffmj/modificare

Problems

As we have learned, Modificare only needs the coverage reports to apply its algorithms. This means that no source code is analyzed by Modificare itself, and theoretically, if we have other program written in another language and are able to generate reports with the format required by Modificare, we can use it to to apply TSR and TCP techniques. This also means that it is impossible for Modificare to generate any kind of test suites in the form of files, since its output is meant to be used directly on Proteja, so we can gather more coverage information. This, and our preference to keep the tools under study written in java and specifically for JUnit tests, made us discard this framework.

3.1.5 MINTS

MINTS is a tool developed by Hsu and Orso[56] in C++ that deals with multi-objective reduction TSR problems. It models the multi-objective problem as an ILP problem and then uses renowned ILP solvers to solve the problem, thus achieving a reduced test suite.

MINTS receives a set of test-related data, that can be coverage or fault detection rate data, and a set of minimization criteria, that must be specified by the testers. This minimization criteria consists of two parts: a set of criteria that acts as a constraint for the minimization, i.e., the minimization of the test execution times; and a minimization policy that defines how the different criteria should be combined to achieve the optimal solution.

This input is given as a set of text files: one *priority* file, at least one *batch* file, at least one *relative_criteria* file, at least one *absolute_criteria* file, and at least one *relative_criterion* file.

The *priority* file contains a path to a *batch* file for each line. The first line of the *batch* file contains the number of test cases in the test suite we wish to optimize, the second line is a path to a *relative_criteria* file, the third line is the number of absolute criteria to be considered, and the fourth and last line is the path to the *absolute_criteria* file. The *relative_criteria* file contains a path to a *relative_criterion* file for each line. The *relative_criterion* file contains the weight assigned to this criterion in the first line, and the coefficients of this criterion in the second line. Finally, the *absolute_criteria* file contains information about the absolute criteria considered in the problem. Each absolute criterion is specified by three lines: the first one denotes the binary operator of the criterion, the second one the right hand side value of the criterion and the last one the left hand side.

Work done and problems

We tried to reproduce the example given in the readme file that was provided in the source code of MINTS, but failed to put any of the compatible solvers into use, or in some cases to find them available.

Despite theoretically being possible to use JUnit test suites since the input received are text files containing criteria and test data, the unavailability of the ILP solvers and the difference of the input, such as its configurability in terms of criteria and policies used and the need of test-related data, caused us to discard MINTS in the end.

3.1.6 GZoltar

GZoltar is a tool developed by Campos et al. [60] and is available as a command line interface, Ant task, Maven plugin and Eclipse plugin. Its most important feature is the generation of fault detection reports and, when integrated with Eclipse, interactive and intuitive graphics, that aid the testers in identifying several faults in shorter time.

At the time of this work we struggled to find information about the test reduction functionality. After sudying the source code of all GZoltar components, we contacted the main developer, who notified us that this functionality was only implemented in the Eclipse plugin, which had some usability and installation issues, and so its development was incomplete and not actively maintained. Given the circumstances, we discarded this tool too.

3.1.7 EvoSuite

EvoSuite was developed by Fraser and Arcuri [50]. It implements search-based and mutation techniques to generate test suites as small as possible and that achieve high coverage according to some criterion.

To generate test suites, EvoSuite only needs the bytecode of the specific class it will generate tests for and its dependencies. After analyzing and instrumenting the bytecode, it will generate a test suite maximizing branch coverage.

While generating the tests, EvoSuite uses a search-based approach, considering a population of candidate solutions and generating new solutions by reproduction of the best individuals, according to some fitness function. In the case of EvoSuite, the candidate solutions are test suites, each of them with a set of test cases. The reproduction of two candidates is done by exchanging some of their test cases between each other. In addition to this, mutation in the candidates is also employed, by adding, removing or changing individual statements or parameters in some test cases. The fitness function in EvoSuite is defined according to some coverage criterion, as branch coverage.

Work done

The work done with EvoSuite was straightforward. We followed the tutorial provided in its website for the command line interface[5]. This consisted in setting up a small simple

[5] `http://www.evosuite.org/documentation/tutorial-part-1/`

project with only a java class representing a Stack, and using EvoSuite to generate a test suite for this class.

Since our idea was to have tools that reduced existing test suites, we faced a minor setback with EvoSuite, as it only reduced its generated test suites. Therefore we also thoroughly studied a major part of its source code, so we could try to better understand the whole EvoSuite process. EvoSuite had an internal representation for the test suites it generated, so our first thought was to try to convert existing test suites into this internal representation and then use the reduction process on them. This did not work as the Object that represented the test suites contained some more information defined in generation time.

Our solution to this problem was to use the tests generated by EvoSuite before and after it performed the reduction, and use the not reduced tests as input to the other TSR tools, thus changing the approach of our framework. Therefore we had to change the EvoSuite source code. All these changes are described and discussed more thoroughly in Chapter 5.

3.1.8 Testler

Testler was developed by Vahabzadeh and Stocco [51], and implements a fine-grained minimization technique to achieve Test Suite Reduction. It analyzes the behaviour of the test cases at statement level to infer a model that represents the relationship between the test statements and test states, which is the information about a test at the time of each statement executed, such as the defined variables, their values and the production method calls. Based on this model, Testler can detect any fine-grained redundancy between test cases in the same test suite, and reorganizes the suites so that it removes these redundancies.

There are four steps to achieve Test Suite Reduction with Testler. The first one is to configure Testler, this can be done in the class ca.ubc.salt.model.utils.Settings, where the user can specify the path to the system under test, a java project. The second step is to execute the instrumenter, which will create a copy of the original project, but with its source code instrumented. The next step is executing the command *mvn test* on the instrumented project, which will generate some trace files, which are XML files containing information about the test run. The last step is to execute a merger class provided by Testler, that will process the data on the trace files and create another project, copied from the original but with the reduction applied in its test suites.

Work done

We followed the steps given in the readme file provided with Testler source code. We tried Testler in a set of Maven projects, which was the same used by the developers in the

study conducted by Vahabzadeh et al. [51]. We later decided to use this set of projects as the study case of our own framework. A detailed information about these projects is given in the next chapter. This experiment revealed some issues to us that we dealt with by changing the source code of Testler. The description of these issues is given next, and the changes made to the source code will be described in Chapter 5.

Problems

Testler was the tool that caused more problems. To try it, the steps described in the readme file provided with Testler were followed. However, the file was outdated and mentioned a class that had some compilation errors, therefore being impossible to carry out the given tutorial. We then contacted one of the developers, who helped us by providing some updated information.

The last step, which was executing the merger class to create the reduced test suite, caused an out of memory error in some of the projects. In some cases, the problem was caused by insufficient memory on the testing machine; in others, it was when Testler tried to load the trace files' content into a String. In the bigger projects, some trace files could take as much space as 3 Gigabytes or more, and so it was impossible to store its content in a single String object.

After solving this issue, the reduced test suites obtained had some compilation errors, such as the use of variables that were not declared or wrong method calls (with wrong parameters), which needed the user intervention.

Despite this, we managed to solve most of the problems and used Testler in our study.

3.1.9 JTOP

JTOP is tool developed by Zhang et al. [46]. It is built as a Eclipse plugin that helps managing JUnit test suites by statically analyzing the software under tests and conducting test reduction, prioritization and selection.

Since JTOP was only available as an Eclipse plugin, we tried to find its source code, so that we can use its tests reduction functionality in our framework, but we did not find it. As such, the only way to use this tool was through Eclipse and we had no way to automate its test reduction process to integrate it in our framework, so no further experimentation was conducted with this tool.

3.1.10 TOBIAS

TOBIAS was developed by Dadeau et al. [47] and is available in a website [6]. Its main functionality is combinatorial test suite generation, which receives an input file from the user, written in a specific language to be interpreted by TOBIAS. This file describes the

[6] `http://tobias.liglab.fr/`

methods the test suite should test, and the inputs it should use to do it. Since TOBIAS generates these test suites combinatorially, it is safe to assume that the number of test cases generated could be excessive and, potentially, with too many redundant test cases. To mitigate this, TOBIAS employs some reduction mechanisms.

Work done

We contacted the developer about the possibility to get the source code for TOBIAS, but the answer was negative. While we were waiting for an answer, we tried the available tutorial to try to better understand the TOBIAS process. Using an input file provided in the aforementioned website. The content of the file is displayed in A.1.

This file contains a call to the constructor of the type that will be tested, calls to each of the methods that should be tested, and a set with values to use when calling these methods.

After uploading the file to the website, the user must input his/her e-mail address, where the the test suite will be received, and can also specify the version of JUnit of the generated test suite.

The generated file is received shortly after via e-mail, and its test cases are a combination of the methods called with the inputs specified.

Problems

Since the tool used a specific language developed for its use as input, and was mainly used to generate tests and had no source code available, which could have make it possible to isolate the test reduction functionality and use it in existing test suites, we decided not to use this tool.

3.1.11 TEMSA

TEMSA (TEst Minimization using Search Algorithms) was developed by Wang et al. [48] and is a web-based application available at its website[7].

The user must give a XML file as input, according to some specific XML schema downloadable in the website, that specifies the features and test cases to test a product. TEMSA then applies some cost measures, like Overall Execution Time, and effectiveness measures, like fault detection capability or test minimization percentage, that can also be set by the user in the website. Based on this input, it then generates a set of XML files representing the minimized test suite.

[7] `http://zen-tools.com/TEMSA/`

Problems

We tried to follow the instructions that are available in the website, but could not upload any input file to the server. When we tried to upload some file, the website would just put a placeholder path in the "input file" field and would not actually upload our file. We also tried to contact the developer to get the source code but got no answer, so we could not use this tool either.

3.2 Final set of Test Suite Reduction tools

While we studied the initial set of TSR tools, we were also defining a set of characteristics for the tools that we want to integrate to have. In order to be able to make the most impartial and equal comparison, we decided that we should integrate only tools that were written in Java and, more importantly, that could reduce JUnit test suites. For this reason, and taking into account all the problems reported in the last section, the final set of TSR tools contained Evosuite, Testler and Randoop.

At this point, we thought about the studied TCP techniques and how they would perform comparatively to the TSR ones. As such, we decided to enrich the main objective of this work by adding a second goal, described in the first chapter. Our idea was to integrate a TCP tool to get reorganized test suites, followed by the removal of the less important test cases, according to some cut value. The chosen tool for this goal and the work done with it is described in the next section.

3.3 Test case prioritization tool: Kanonizo

Kanonizo[8] was developed by Paterson and implements several algorithms to achieve TCP on test suites, receiving as input the path to the folder containing the byte code of the system under test, the algorithm to use, and a set of some optional parameters.

After execution, Kanonizo generates several reports, the most interesting being the ordering one, a comma-separated values file containing a header as the first line, and each following line a test case name, the class it belongs to, its execution time, whether it passed or not, and the total number of covered lines, sorted by descending order of importance, according to the chosen Kanonizo algorithm.

Algorithms

Kanonizo implements several algorithms from a lot of methodologies. Random search and genetic algorithm are search-based; random, greedy, additional greedy and schwa

[8] `https://github.com/kanonizo/kanonizo`

work at the test case level; marijan, huang, cho and elbaum are history-based, meaning that they need an history file as input. A brief description of each algorithm is given next:

- Random: reorganizes the test cases by a random order;
- Greedy: prioritizes test cases that cover the maximum number of lines;
- Huang: prioritizes tests according to the severity of its failures;
- Schwa: prioritizes tests according to the likelihood of each class and method containing a fault;
- Marijan: prioritizes test cases by processing an history file with information related to previous test executions;
- Random search: uses a fitness function to determine the important tests according to a criterion;
- Elbaum: prioritizes tests that have not failed in a long time;
- Additional greedy: Similar to greedy, but excludes the already covered lines;
- Cho: prioritizes tests according to the number of consecutive fails;
- Genetic algorithm: prioritizes tests by applying mutations and crossovers between individuals (test cases).

Since our framework will execute tests for the first time, we had to discard the history-based algorithms. Schwa needed a lot of additional configuration as input, as well as Python[9] installed on the system, so we also discarded this algorithm, as well as the genetic algorithm, as it did not seem to work.

The chosen algorithms to use with Kanonizo were Random, Random search, Greedy and Additional greedy.

Work done

We tried Kanonizo with one of the projects used in Testler and later in our own study. The behaviour was the expected one and it generated the needed reports.

In order to conduct the study we wanted for this work, we programmed additional behaviour to the Kanonizo tool. Using the ordering reports generated, the goal was to reduce a given test suite by erasing the least important test cases of each test suite, according to some given cut off value. Details of this implementation are given in Chapter 5.

[9] `https://www.python.org/`

Chapter 4

Design

Along with the work described in the last chapter, we started planning and designing the solution to our problem of finding the best TSR tools and techniques.

This chapter will explain the decisions made about the implementation of our framework, such as the technologies, tools to be used, as well as the evolution of the approach from the beginning to the end.

4.1 Initial approach

The initial approach was thought as a framework that integrated several TSR tools, and that would use them to reduce a set of test suites, obtaining a different set of test suites for each tool used.

Therefore, the indispensable aspects of the basic idea behind our framework would be that its input would be a set of test suites, its output a set of reduced test suites for each TSR tool implemented, and data about those suites.

As learned during the work described in the last chapter, not every TSR tool had as input the same format of test suites, same for the output. Therefore it would be needed some sort of converter from a standardized format of test suites to each specific format for each TSR tool for the input, and vice versa for the output, as well as some interface to extract the code metrics to analyse the original test runs vs. the reduced test runs. Figure 4.1 shows the initial approach for the implementation of the framework.

As we defined our working set of TSR tools, we made changes to the initial approach of our framework. Since the final selected tools no longer had different kinds of input formats, the initial thought of a converter normalizing all the input was discarded. Furthermore, since we were going to use the Evosuite tool to also generate tests, its behaviour was going to be somewhat different than the one of the others TSR tools. We also added the TCP tool Kanonizo.

Figure 4.2 depicts our approach after the study and decision of the TSR to integrate.

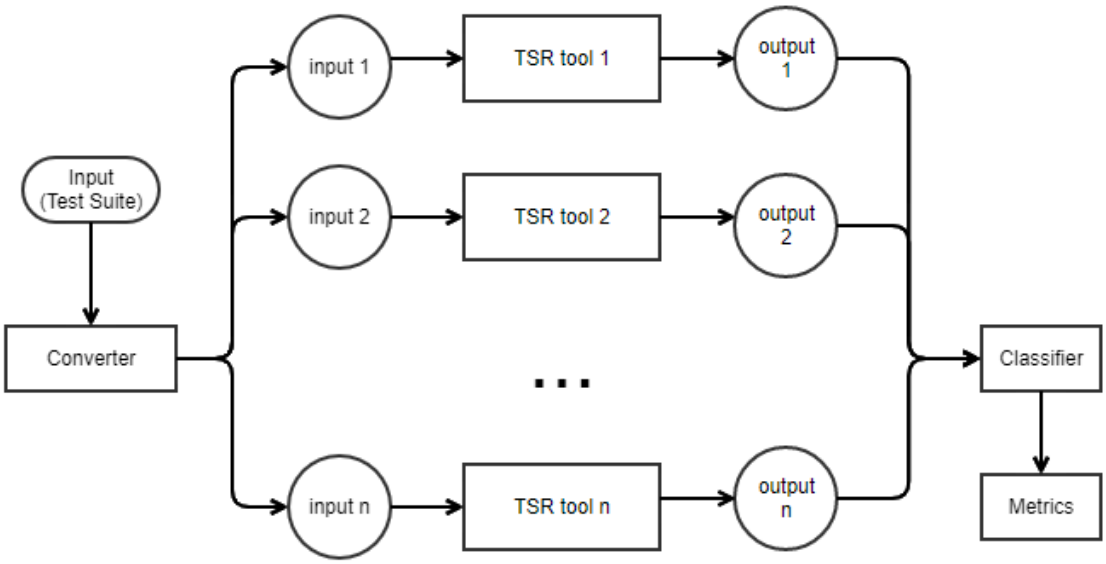

Figure 4.1: Overview of our initial approach.

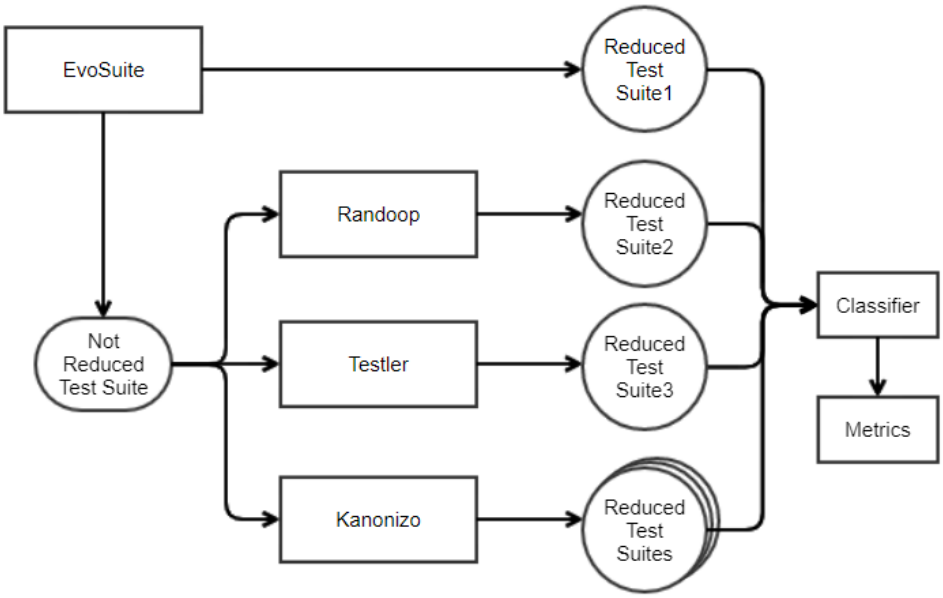

Figure 4.2: Overview of our midterm approach.

To implement our framework, we decided to use Java[1] since all the TSR and TCP tools focused were also developed in Java. Furthermore it is a language with a great amount of support in the web, it is platform independent and it has a lot of libraries available.

At this point, we needed to figure out which open-source projects should be used to conduct this study, and define the way to collect their test suites data and what to do with it. These steps are discussed in the next sections.

4.2 Case studies

Although we wanted our sample to be large and heterogeneous, in order to conduct a more vast and complete study, there were some properties that we wished to maintain, and as such, all the projects used were Maven Java projects, with JUnit test suites.

Since Vahabzadeh et al. [51] used a set of projects as a case study with these properties, we decided to use a subset of it as the case study of our work, those projects being commons-lang[2], commons-email[3], pmd-core[4], tudu-lists[5], lambdaj[6], jfreechart[7], java-library[8], crunch-core[9], crunch-kafka[10], tika-xmp[11] and xml-sec[12].

As we can see in Table 4.1, this set of projects is quite diverse dimension-wise, since it has number of lines of code (LOC) ranging from 1,644 to 221,637 and number of classes from 10 to 637. This allows us to verify if the tools that we will integrate work well when applied on test suites with different sizes (since the size of the test suite is directly influenced by the number of classes, as we are using EvoSuite to generate them).

We can also notice, by the number of dependents taken from each project's github repository, that we have a wide range of other projects that depend on the ones we will test, meaning that these are active and useful projects.

[1] https://www.java.com
[2] https://github.com/apache/commons-lang
[3] https://github.com/apache/commons-email
[4] https://github.com/pmd/pmd/tree/master/pmd-core
[5] https://github.com/jdubois/Tudu-Lists
[6] https://github.com/mariofusco/lambdaj
[7] https://github.com/jfree/jfreechart
[8] https://github.com/urbanairship/java-library
[9] https://github.com/apache/crunch/tree/master/crunch-core
[10] https://github.com/apache/crunch/tree/master/crunch-kafka
[11] https://github.com/apache/tika/tree/master/tika-xmp
[12] https://github.com/apache/santuario-java

Project	Dimension			# Dependents	Version	% Branch Coverage	% Total Coverage	Tests' Runtime (s)
	LOC	# Classes	# Test Classes					
commons-lang	78 000	151	172	121 866	3.7	40.11	76.97	10.14
commons-email	6 182	23	26	14 219	1.5	60.84	94.45	17.55
pmd-core	48 221	350	102	160	6.20	22.99	47.70	16.36
tudu-lists	4 497	49	20	0	3.0	23.67	45.41	0.21
lambdaj	8 325	95	61	3 813	2.4.1	36.03	77.27	1.85
jfreechart	221 637	637	350	6 213	1.6.0	3.20	55.71	2.26
java-library	36 298	437	157	9	2.1.0	25.06	74.32	5.03
crunch-core	37 954	297	185	6	0.15.0	18.08	34.51	2.64
crunch-kafka	2 449	14	16	6	0.15.0	51.43	86.90	0.23
tika-xmp	1 644	10	2	97	1.18	34.54	68.47	0.09
xml-sec	70 530	392	198	1 745	2.0.8	35.98	65.64	20.81

Table 4.1: Case studies projects.

4.3 Code analysis tools

In order to conduct our comparative analysis, we needed to gather some test execution data, so we could compare the one from the original test execution with the reduced ones. Hence we searched for code analysis tools and considered JaCoCo[13] and OpenClover[14].

These tools instrument the source code of a project so they can gather a set of information, generally configurable, about its execution.

4.3.1 JaCoCo

JaCoCo is a tool that provides code coverage analysis on several coverage metrics, such as instruction, branch, line and method coverage. It only needs the bytecode of the system under test and generates coverage reports in different formats like HTML, XML and CSV. It is available as an Ant task, a Maven plugin and a command line interface.

We carried out a simple example with the Maven plugin (version 0.8.6). First of all we had to configure JaCoCo on the System Under Test. This can be made by adding the JaCoCo plugin in the pom.xml file, and configure it with two executions: one that will be made before the execution of the tests, that will generate the execution data; and another after the tests, that will generate the code coverage report. After this, we must configure the Maven surefire plugin so that we make sure the JaCoCo agent will be executed when we are running the test suites.

JaCoCo gathers information about several code metrics, such as the number of instruction, branches, lines and methods missed and covered and saves them in a report with the specified format.

[13] `https://www.jacoco.org/`
[14] `https://openclover.org/`

4.3.2 OpenClover

OpenClover, similarly to JaCoCo, is a code coverage analysis tool. It provides the measure of more than twenty metrics, with the ability for the user to define its own code metrics. Some of it are the cyclomatic complexity, which can roughly be described as the number of unit test cases to fully cover a certain piece of software[77], the covered methods, branches and statements, with the option to be presented as a percentage or raw value, the execution time for the running tests, among others. It also gives the possibility to generate reports in several formats, such as HTML, XML, PDF, JSON and simple text. It also has a lot of possibilities of integration, being available as a Maven plugin, Ant task, Gradle plugin and Eclipse plugin. OpenClover can, and must, be configured directly in the project's pom file, along with an additional report configuration XML file, where the user specifies the metrics that should be measured.

We tried a simple example with the Maven plugin (version 4.3.1) available in the OpenClover source code, where the pom file was already configured and there was an available report configuration file.

4.3.3 Chosen tool

In the end we chose OpenClover as the code coverage analysis tool to use in our framework, as it offered more generated reports formats and had the possibility of gathering data about more code metrics than JaCoCo.

We defined the code metrics to analyze to be the cyclomatic complexity, covered branches, methods and statements, total branches, methods and statements, and total percentage covered for the source code, and not commented lines of code, test suites' execution time, number of passing, failing and total tests, and size of file in bytes, for the tests source code.

4.4 Code metrics processing and tools evaluation

Now that we had defined the process to obtain the coverage information about the original and reduced test suites, we needed to define a way to compare them in order to evaluate the tools according to these metrics.

Our main idea was to use these metrics to compute a score for each tool, which will then be used to compare them. We decided to use the AHP, a MCDM approach, that was detailed before in Chapter 2.

Figure 4.3 defines the hierarchy model for our approach. It descends from the Score we want to obtain for each pair (Tool, Case Study), then we chose the three main criteria to evaluate each tool: Dimension, Time and Coverage, since these ones represented the values we were measuring with OpenClover. Finally we divided Dimension in sub-criteria

file sizes and number of test cases, Time in test suites' execution time, and Coverage in percentage of branches covered and percentage of total coverage. The alternative scenarios would be each of the tools integrated. We chose not to take the time each tool took to reduce the tests into account for the calculation of the score, as this would be an unfair evaluation, since EvoSuite takes time to also generate the test suites.

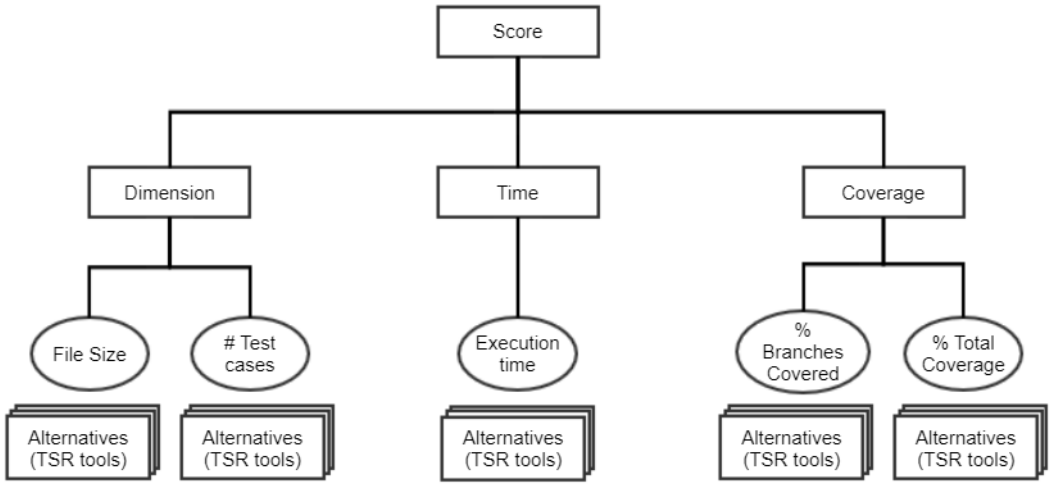

Figure 4.3: Hierarchy model for our AHP approach

As such, given:

- T the set of tools integrated,
- $t_i \in T, 1 \leq i \leq |T|$ an integrated tool,
- P the set of projects to be tested,
- $p_j \in P, 1 \leq j \leq |P|$ a project to be tested,
- $s_{i,j}, 1 \leq i \leq |T|, 1 \leq j \leq |P|$ the score t_i got for p_j

The final score for tool t_i, $S_i, 1 \leq i \leq |T|$, will be computed by calculating the average of all scores $s_{i,j}, 1 \leq j \leq |P|$.

4.5 Final concept

Having made all these decisions we were now ready to start the implementation of our framework, which will be described in the next chapter. We also defined the functional requirements of the framework as such:

- Executes a given test suite and collects coverage and execution data about it;

- Reduces the given test suite at least as many different times as the number of TSR tools integrated;
- Executes the reduced test suites, collecting the same data about them as the original ones;
- Conducts a comparative analysis using the collected data and a multi-criteria decision-making approach;
- Ranks the integrated TSR and TCP tools.

The framework configuration will be centralized in a configuration XML file.

Figure 4.4 depicts an high-level view of the final approach for our framework. The basic concept is the same as the one from the initial and middle approaches, but now with clearer information about its execution flow for each tested project.

We also thought about implementing the possibility for the user to choose whether to use the EvoSuite generated test suites or the ones provided with the source code of the tested project, making it possible for the framework to use existing test suites and removing the dependence on EvoSuite.

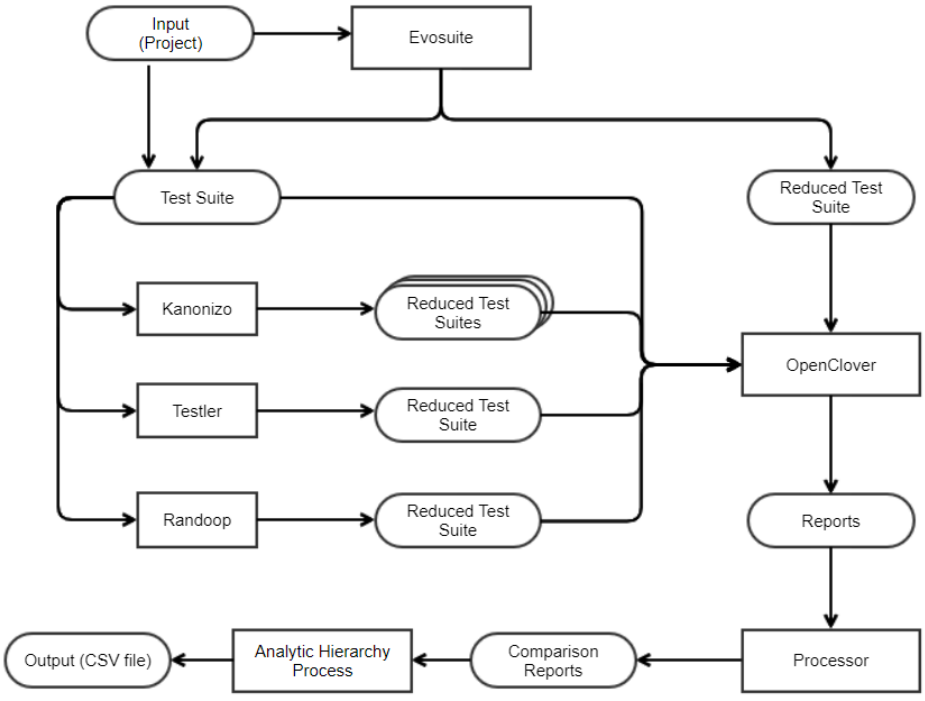

Figure 4.4: Final approach for framework

Input and Output

Our framework will receive a set of Maven java projects, with its paths specified in the configuration file. These projects must follow the standard Maven directory structure, as

some relative paths are assumed along the execution of the framework (for instance, the path where the source files or test files are).

As output, the framework will:

- create, for each project, $n + m * k$ copies of the project where n is the number of integrated tools (except for Kanonizo), m is the number of Kanonizo algorithms used and k is the number of cut off values defined. Each one of these copies has the reduced test suites in turn of the original;

- generate, for each project, the same amount of comparison reports (one for each copied project). These reports are CSV files where each pair of rows represent a source class and the information gathered according to the metrics measured by OpenClover, defined in this chapter, as well as the reduced version of that class. In the end of these files, the same information is presented for the test classes as well;

- generate a single CSV file where each row represents a tool and the scores it achieved for each tested project, as well as the total score for this tool. The tools are sorted in descending order by score.

The first two outputs are merely auxiliary files to achieve the interesting output of the framework, which is the file containing the ranking of the analyzed tools.

Chapter 5

Implementation

The last part of this work was to implement the framework that would allow us to make the comparative analysis between the selected tools.

This chapter explains in detail the implementation process of our framework, such as its code organization, the role of each developed class in the framework, how we handle its configuration, as well as the modifications we had to make in each tool to seamlessly integrate it in our framework, and all the problems found along the way.

5.1 Code structure

The behaviour of our framework can be divided into three parts: code analysis, tool execution, and score computation. This division is well explicit in the code structure, as it is divided in packages that separates each function. The code is organized between four packages, all subpackages from the main package *pt.ul.fc.di.pei42103*. Package *clover* contains classes related to the work done by OpenClover for code analysis; package *tools* contains the classes that represent each of the integrated TSR tool and Kanonizo; package *ahp* has classes that implement the AHP; and package *utils* contains several utilities classes.

For better understanding of the code structure, some class diagrams were also made, as we can see in Figures B.1, B.2, B.3, B.4 and B.5. The class diagram was split into five parts, so it was possible to show detailed information in each package, as well as a general view of the whole framework.

A description of each class from each package will be given next.

Package pt.ul.fc.di.pei42103

- Main

  This is the main class of our framework. It starts by loading the configuration values set by the user in the configuration file. Then, for each project, it does a pre-processing job that prepares this project for the experiment. This includes modifying its pom to include the needed plugins to execute OpenClover, and make copies of its structure in which the reduced test suites will be saved. It then uses EvoSuite to generate test suites, uses OpenClover on both the original and the EvoSuite-reduced suites, writes the comparative report for EvoSuite and project, and, finally, computes the score that EvoSuite got for the project. If the user sets the configuration file to not use EvoSuite, this step is skipped.

  Following this, it will, for each integrated tool, reduce the original tests suites, use OpenClover on the reduced suites and write the comparative report for this tool and project, as well as computing the score that the tool got for the project.

  The next step, if Kanonizo is set in the configuration file, is repeating the same process for Kanonizo, which is treated as several tools, according to the algorithms and cut off values defined by the user in the configuration.

  Finally, the execution times that each tool took to reduce each project are displayed in the output, and a CSV file containing the sorted tools by final score obtained.

  If, for some reason, something fails, the framework skips the execution of a tool in some project, and continues executing for the rest.

Package pt.ul.fc.di.pei42103.clover

- CloverRunner

  This is a simple class that contains the command to run OpenClover and a method to execute it given a project root path.

- TestReportRow

  The objects from this class represent a row in the comparative report with the data gathered with OpenClover for the test classes. Hence, this class has fields that represent the metrics measured, as well as a boolean that indicates if this class belongs to the original or the reduced suites, and appropriate getters and setters.

- SrcReportRow

  Similarly with TestReportRow, the objects from this class represent a row in the comparative report for source classes. It has fields that represent the metrics measured.

- ReportRowBuilder

  This class contains methods that receive the information from the OpenClover reports and creates SrcReportRow and TestReportRow objects.

- CloverReportReader

  This class contains methods that read the OpenClover reports and create lists of SrcReportRow and TestReportRow from it, which will then be used to write the comparative reports.

Package pt.ul.fc.di.pei42103.tools

- TSRTool

  This is an abstract class that all the integrated tools must extend. Along with specific fields for the name and path to jar, it also contains methods for the general behaviour of a tool, like creating a copy of the current project to save the reduced suite and appropriate getters and setters.

- EvoSuite

  This class calls the main method from the EvoSuite class of EvoSuite, passing it the appropriate arguments. After its execution, it makes some changes to the generated test suites, which will be explained further ahead in the EvoSuiteHelper class.

- Testler

  This class reproduces the Testler process studied in Chapter 3. It calls the Instrumenter class, runs the command *mvn test* in the instrumented project, and finally it runs the BackwardTestMerger class.

- Randoop

  This class calls the main method from the Main class of Randoop, passing the appropriate arguments.

- Kanonizo

  This class will call the main method from the Main class of Kanonizo for each algorithm set in the configuration file. Then, it will process the Kanonizo reports with the test cases orderings and, for each cut off value, will copy the test suite into a new directory and erase the least important test cases in each class, according to the cut off value.

Package pt.ul.fc.di.pei42103.ahp

- Criterion

  The objects of this class represent a criterion that belongs to the AHP. It is a simple encapsulation class containing the name of the criterion, its priority, a list of its sub-criteria and a list of its importance related to other criterion.

- PairwiseComparisonMatrix

  The objects of this class represent the pair-wise comparison matrices used to compute the priorities of criteria within the AHP, as depicted in Chapter 2.

  Given a list of criterion, it will build the pairwise comparison matrix between them and compute the priority for each criterion. It has a method that returns the computed priorities.

- Ahp

  This class deals with the calculation of a score. It receives a list of all the most high-level criteria, and computes the priorities for all of them and their sub-criteria, using PairwiseComparisonMatrix. It has a method that returns the score according to these priorities and the metrics retrieved from the source code.

- Score

  This class follows the singleton pattern and encapsulates a map of String, representing a tool's name, to a list of pairs of String and double, that represent the scores that the tool got for each project.

  It features methods to put a score a certain tool got for a project in the map, as well as getting a score by a tool's and project's name. It also provides a method to get a sorted list by score with all the tools and the final scores they got.

Package pt.ul.fc.di.pei42103.utils

- Pair

  This is a simple class that encapsulates two objects of two different types.

- Subject

  The objects of this class represent the subjects of the experiment, in other words, the projects to be tested. It has fields to represent its name, path to root and its importance to the calculation of the final score of a tool, getters and setters, as well as a method to prepare a project for the experiment, in this case, the addition of information to its pom.

- Triple

  An utility class that encapsulates three Strings defining the name of a tool, a subject and a criterion. It is used to simplify the saving of measured metrics.

- Logger

  This class is a simple logging class, with methods to print information or error messages to both the System.out and a file.

- CommandRunner

  This class contains a method to execute a command through the ProcessBuilder java class. It is used in the framework to execute OpenClover, as well as some of the tools.

- EvoSuiteHelper

  The tests generated by EvoSuite are meant to execute in a predefined environment and depend on some EvoSuite classes. In order to have regular JUnit tests we identified all the changes we would need to do in these tests and implemented this class to rewrite the test classes with these changes. Given a path to the tests folder, this operation will remove all the scaffolding files generated by EvoSuite, rearrange the way the tests deal with thrown exceptions, remove all the imports and statements related to EvoSuite, and replace all the calls and initializes to Mock objects from EvoSuite to their Java API counterparts.

- TestlerSuiteHelper

  The test cases generated by testler were always written in a single line, and since most of them had compilation errors, this made it very difficult to identify and solve all these errors. This class rewrites these tests with the appropriate line breaks and indentation, and solves a common compilation error found, which was the use of a variable not previously defined.

- FileUtils

  This class contains methods that make operations with files, that will be used in the framework to write the comparative reports, change project's pom files, erase test methods and copying directories.

- XMLUtils

  This class has methods that help with the parsing of the OpenClover XML reports.

- Configuration

  This class contains all the configuration information defined by the user in the configuration file (which content will be detailed in the next section). It sets all of

its fields when the class is loaded, and it provides getters so that it can be used throughout the whole framework.

5.2 Framework modifiability and configuration

One of the goals with the implementation of this framework was for it to be as easy as possible to modify. We achieved this by centralizing all of this information in a configuration XML file.

Before using our framework, the user must set a group of configurations required for its execution, that must all be defined in this file, that is saved in the src/main/resources folder of the project. An example to the content of this file can be seen in B.1. The needed configurations are as follows:

- **Mode of execution**: We implemented two modes of execution in the framework, which the user can specify by their names: full and analyze. Full mode is the one specified in the description of the Main class in the previous section. The analyze mode assumes that the reductions were already made and that the comparative reports already exist, and therefore it will load the information stored in these reports to compute the scores for each tool. We chose to implement this mode to facilitate the testing of the framework, so that we could analyze how the criteria and importances between them influenced the score of each tool, without the need to re-run all of the framework behaviour, since it was a long process.

- **Reduced Projects Path**: The path where the reduced projects would be saved, which will be the original project with reduced test suites by some tool. If this path is *path*, given a tool name *t* and a project name *p*, the framework will use this path to copy the project to be tested to a folder denoted by the path *path/p/t*. In Kanonizo case, given an algorithm *a* and cut off value *c*, the framework will copy the project to be tested into all the combinations of *path/p/Kanonizo/a/c*.

- **Reports path**: The path where the comparative reports with the OpenClover gathered data will be saved. Given a project name *p* and a tool name *t*, a report will be named as *p_t_report.csv*. In Kanonizo case, given an algorithm *a* and a cut off value *c*, the file will be named *p_kanonizo#a#c_report.csv*.

- **Output path**: The path where the output CSV file will be saved. This file will be named as ReductionScores.csv.

- **Evosuite usage**: A simple true or false value to indicate if the framework should use EvoSuite or not.

- **Kanonizo usage**: The user can define if the framework will use Kanonizo or not, the same way as EvoSuite.

- **Integrated tools**: The set of TSR tools that the framework integrates and will use for the experiment (besides EvoSuite and Kanonizo).

- **Projects to be tested**: The set of projects to be used in the experiment, specified by their name, path to the root folder and relative weight to the final score. Since not all the projects have the same importance, as some of them are widely used and others not so much, the score that a tool gets for a less important project should not contribute to the final score as much as a score that a tool gets for a more important project. As such, the user can define, for each project, the weight, as a natural number, that the scores for this project will weigh in the calculation of a final score for a tool.

- **Criteria for AHP**: The set of criteria that will be used for the AHP to determine the score for a tool, along with the values of importance between them.

5.2.1 Modifiability of the framework

As time passes, the development of other TSR tools is imminent. In order to keep this framework relevant in the study of these tools, we implemented it in such a way that it provides a rather easy way of integrating new tools. To do so, the user must provide a tool's name in the configuration file, and code a class in the package *pt.ul.fc.di.pei42103.tools* that extends the class TSRTool, where the behaviour of this new tool will be implemented. Through Java Reflection the tool will then be loaded in the same way as the ones implemented by default.

Regarding EvoSuite, the user can disable its use. In this case, the framework will use the existing test suites for each project. As for Kanonizo, in addition to the possibility of disabling its use, the user can also specify the algorithms that should be used, as well as the cut off values for the reduction.

The set of projects to use in the experiment is really simple to modify too, since it is only necessary to change the configuration file accordingly, providing that the path given exists.

As for the criteria used to compute the tools' score, the user can also add new ones or modify the existing ones, such as the values of the importance between them, by following the structure presented in the configuration file, which contributes to getting different evaluations for each tool, considering the specified criteria. Additionally, if there is a new Criterion added, the user must also code the way to calculate its values in the AHP class, method saveValues.

5.3 Analytic Hierarchy Process implementation

When the framework starts, it loads the criteria from the specification in the configuration file and computes the priority for each base-level criteria (in our case being file size, number of test cases, test suites' execution time, percentage of branch covered and percentage of total coverage), as it is defined in the AHP. The next step would be, using the measured values for each criterion and tool, to deduce the pairwise comparison matrix between the alternatives (the reduction tools), and compute their priorities, which would then be used to rank the tools. However, we decided to diverge from the AHP in this step, as we wanted the scores of each tool to be independent from one another. Instead, we use the sub-criteria priorities and the values measured for the criteria for each tool to compute the scores for each tool.

For each criterion, the values are a combination of the measurement of the related metric for the original test suites and the reduced one, in such a way that, for:

- Cost criteria (such as file size, number of test cases and test suites' execution time): we take $v_{original}$ and $v_{reduced}$ and calculate $(1 - \frac{v_{reduced}}{v_{original}}) * 100$. In the case that $v_{original}$ is 0, we define the whole value to be 0.

- Benefit criteria (such as percentage of branch coverage and percentage of total coverage): we take $v_{original}$ and $v_{reduced}$ and calculate $(\frac{v_{reduced}}{v_{original}}) * 100$. In the case that $v_{original}$ is 0, we define the whole value to be 0.

Once we collect all these values, we normalize them for each pair (Project, Criterion), so that we can relate each value obtained for the different tools, for the same project and criterion. We normalize them applying the formula $\frac{v_i}{v_{max}}$, where v_i is the value obtained for a given tool, in the pair (Project, Criterion), and v_{max} is the maximum value from all the values for this criterion in this project.

The final step in getting the score is multiplying each of these values for the respective criteria priority obtained, and sum the results obtained for each criterion. This ensures that the score for a given tool will always be a value between 0 and 1. In addition, the closer it is to 1, the better, so we can rank the TSR tools by sorting the scores by descending order.

5.4 Modified Test Suite Reduction Tools

In order to seamlessly integrate the tools in our framework, we needed to change or add some code to their source code, sometimes just so its execution could provide us the results we needed, and others to correct some errors from the tools themselves.

EvoSuite

The original EvoSuite (version 1.0.7 used in this work) takes as parameters (for the execution profile we needed) the flag *-target* and the path to a project's bytecode. Since we changed EvoSuite so it produced two sets of test suites, we modified it to accept extra arguments that represented the path in which to save the original test suites, and the path in which to save the reduced test suites.

We then modified the EvoSuite class TestSuiteGenerator, so that it saves the generated test suite before and after reduction, saving each one in the paths given as an argument.

Testler

The original Testler (version 0.0.1 used in this work) has all of its configuration hard-coded in its Settings class, such as the project, instrumented project and merged project paths. Since we needed to use Testler for a lot of different projects within the same execution, we changed this so that these values were passed as arguments to its classes.

As stated in Chapter 3, there was a problem with Testler when we tried to reduce some bigger projects, since in those cases they generated trace files with more than 3Gb, which caused an out of memory error. To solve this we changed the way Testler processed these trace files, instead of loading the file content to a single String object, we changed it to load its content to a LinkedHashSet, where each entry would be a line of the file. We then proceeded to make all the necessary alterations to Testler source code to integrate this change. After these changes we ran Testler on a set of projects we had already tested, and confirmed that the results were the same, as a way to evaluate if these changes had altered the output in any way.

Randoop

The original Randoop (version 4.1.2 used in this work) saves the reduced suites with the suffix *Minimized* in their name. This caused the command *mvn test* to not find any test cases (since by default it looks for classes ending in 'Test'), so we changed that suffix to *MinimizedTest*. We also modified the code so it would save the reduced test suites to another path, given as an argument, as opposed to the default one, that was the original test suite's path.

5.5 Problems found

The main problems found related to the implementation were the integration of the reduction tools into our framework, as we can see in the last section since all of them had to suffer modifications.

The biggest problem we dealt with was with Testler, as it was little documented and the tests it generated always had some kind of compilation errors. Since we wanted our framework to execute automatically without any user interaction, we tried to implement some ways to solve those errors with the class TestlerSuiteHelper, but there can be some errors that this class does not solve, since there are always errors specific to each project code.

For the sake of our experiment, this kind of errors were solved manually, and Testler was left as a part of our framework. However, this means that the user must intervene in order to solve the errors in the reduced test suites.

Chapter 6

Results

When the development of the framework was finished, we conducted a series of experiments in order to answer the questions made at the beginning of this work and to analyze the influence each criteria had on the final score for each tool.

Each experiment was repeated twice, one using tests generated by EvoSuite, and another with the test suites that come included with each studied project (henceforth referred to as EvoSuite run and Normal run, respectively). With this, we intend to analyze if the use of automatically generated tests influence the efficiency of the tested tools.

After generating the OpenClover reports, when then proceeded to configure the criteria (the values of importance between them) to analyze the change this caused in the scores of each tool.

The next sections describe the working environment, the obtained results, and a discussion regarding these results.

6.1 Testing environment

All the experiments were done within the same environment: a laptop with a processor Intel Core i7 7700HQ and 16Gb RAM with Windows 10. The whole framework was implemented and tested using Eclipse IDE[1], version 4.8.0, and using Java version 1.8.0_231 and Maven version 3.5.3.

6.2 Experiments and results

We can separate each experiment in two parts (same as our framework's process): the first one is the generation and reduction of test suites, with their subsequent analysis; the second one is the interpretation of the test suites' analysis, and calculation of scores for each tool.

[1] `https://www.eclipse.org/`

It is important to note that we configured Kanonizo to use four algorithms and cut off values of 10, 15, 20, 25 and 30, and to remember that each of this combination is treated as a separated test suite "reduction" tool.

6.2.1 Generating, reducing and analyzing tests

This part of the experiment is just the generation and preparation of data. We can compare the times each tool took to reduce the test suites of each project.

Tables 6.1 and 6.2 show the execution times for each tool across all projects, for the EvoSuite run and the Normal one, respectively. In Table 6.1, the column "Total (w/o EvoSuite)" shows us the executing times without considering the time took by EvoSuite, so it is possible to compare them with the ones from the Normal Run. The Kanonizo column represents every execution of Kanonizo with all the configured algorithms and appliances of cut off values.

Project	**Execution time (hh:mm:ss)**					
	EvoSuite	**Testler**	**Randoop**	**Kanonizo**	**Total**	**Total (w/o EvoSuite)**
commons-lang	04:10:26	00:38:30	00:09:31	00:29:30	05:27:47	01:17:31
commons-email	00:14:07	00:11:00	00:00:12	00:08:37	00:33:56	00:19:49
pmd-core	07:06:57	07:36:50	00:06:28	00:33:44	15:23:59	08:17:02
tudu-lists	00:43:59	00:31:03	00:00:28	00:05:59	01:21:29	00:37:30
lambdaj	01:50:06	01:10:42	00:01:19	00:13:48	03:15:55	01:25:49
jfreechart	03:31:54	00:45:32	00:00:09	00:18:16	04:35:51	01:03:57
java-library	03:26:49	00:37:12	00:03:28	00:22:10	04:29:39	01:02:50
crunch-core	01:40:13	00:51:07	00:01:34	00:14:02	02:46:56	01:06:43
crunch-kafka	00:02:50	00:04:31	00:00:05	00:06:20	00:13:46	00:10:56
tika-xmp	00:00:54	00:00:11	00:00:01	00:06:24	00:07:30	00:06:36
xml-sec	09:47:21	01:10:42	00:15:40	07:01:11	18:14:54	08:27:33
Total	32:35:36	13:37:20	00:38:55	09:40:01	56:31:52	23:56:16

Table 6.1: Execution times each tool took for each project (with EvoSuite).

Project	**Execution time (hh:mm:ss)**			
	Testler	**Randoop**	**Kanonizo**	**Total**
commons-lang	00:09:55	00:00:24	00:28:03	00:38:22
commons-email	00:08:26	04:55:31	00:23:29	05:27:27
pmd-core	00:03:42	00:29:56	00:06:17	00:39:55
tudu-lists	00:04:27	00:00:34	00:06:04	00:11:05
lambdaj	00:01:06	00:06:03	00:07:44	00:14:53
jfreechart	00:48:03	01:18:48	00:25:24	02:32:15
java-library	00:26:47	02:29:15	00:17:45	03:13:47
crunch-core	00:02:39	00:08:58	00:09:42	00:21:19
crunch-kafka	00:02:32	00:06:49	00:26:47	00:36:08
tika-xmp	00:00:14	00:04:04	00:03:26	00:07:44
xml-sec	00:18:17	00:20:20	00:07:46	00:46:23
Total	02:06:08	10:00:42	02:42:27	14:49:17

Table 6.2: Execution times each tool took for each project (without EvoSuite).

As we can see from the tables, the EvoSuite run took more time than the one without it (even discarding the EvoSuite execution times). Aside from Randoop, all the others tools took longer to execute with automatically generated tests.

A possible explanation for this can be the difference in the size of test suites in both experiments. All the test suites generated by EvoSuite are, approximately, 15.95 Mb with 16910 test cases in total, while the original test suites are 9.4 Mb with 9834 test cases in total, as inferred from the values shown in Tables 6.4 and 6.5. As the integrated tools work directly on the test suites, their size can influence the time needed for the tool to complete the reduction. However, Randoop took approximately 9 hours and 20 minutes longer reducing the original test suites than the EvoSuite ones. This can be because Randoop only reduces failing tests. Since the EvoSuite ones are executed while being generated to try and generate passing test cases, it would be expected for them to have fewer failing tests.

Regarding Kanonizo, Table 6.3 shows us the times Kanonizo took to execute each of the algorithms across all projects. These values do not consider the time took reducing the suites and running OpenClover on the reduced suites, which are considered in the Kanonizo column of Tables 6.1 and 6.2. We notice that the algorithm that took longer was Random Search in both runs, executing for 7 minutes and 1 second in the EvoSuite Run, and 6 minutes and 50 seconds in the Normal Run, in contrast with the other algorithms, where they all took less than a minute to execute.

EvoSuite Run				
Algorithm	Random	Random Search	Greedy	Additional Greedy
Total time (hh:mm:ss)	00:00:32	00:07:01	00:00:15	00:00:28

Normal Run				
Algorithm	Random	Random Search	Greedy	Additional Greedy
Total time (hh:mm:ss)	00:00:32	00:06:50	00:00:11	00:00:16

Table 6.3: Kanonizo algorithms total executing time.

Tables 6.4 and 6.5 show the cumulative data gathered by OpenClover for each one of the original projects (before reduction) and for all the considered metrics, in both runs. The values that contribute to this cumulative data will be the $v_{original}$ in the formula applied to calculate the comparative values between the original and the reduced test suites, seen in Section 5.3.

EvoSuite Run					
Projects	**FileSize (bytes)**	**# Test Cases**	**Test Suites' Exec. Time (s)**	**Branch Cov. (%)**	**Total Cov. (%)**
commons-email	123808	143	0.926	20.854	16.468
commons-lang	5471991	5541	5.794	59.078	83.353
crunch-core	413051	424	0.595	3.503	5.794
crunch-kafka	15676	24	0.25	2.806	5.953
java-library	1671313	1620	5.36	16.836	36.293
jfreechart	71087	51	0.626	0.761	2.971
lambdaj	680351	795	0.781	17.812	42.718
pmd-core	3155247	3589	11.324	21.99	42.361
tika-xmp	1197	2	0.015	0	0.580
tudu-lists	221805	252	0.281	22.959	29.323
xmlsec	4124237	4469	80.508	23.009	38.262

Table 6.4: Test execution data gathered by OpenClover (EvoSuite run).

Normal Run					
Projects	**FileSize (bytes)**	**# Test Cases**	**Test Suites' Exec. Time (s)**	**Branch Cov. (%)**	**Total Cov. (%)**
commons-email	191814	187	10.139	40.11178571	76.976746
commons-lang	2617380	4032	17.553	60.8388125	94.45088
crunch-core	270600	330	2.64	18.07762069	34.51355
crunch-kafka	32356	55	0.23	51.42538462	86.90779
java-library	730171	678	5.029	25.0566895	74.324326
jfreechart	2447639	2175	2.261	33.204961	55.711197
lambdaj	160660	265	1.846	36.02743802	77.27039
pmd-core	466264	1009	16.356	22.9977561	47.707268
tika-xmp	16766	33	0.09	34.54	68.471954
tudu-lists	63491	50	0.208	23.6727451	45.41258
xmlsec	2451059	1020	20.811	35.97684086	65.64482

Table 6.5: Test execution data gathered by OpenClover (Normal run).

6.2.2 Calculating scores

From the moment we have the reduced suites and the comparative reports, we can use them to generate different results, according to the configured values of importance between each criteria and sub-criteria.

We wanted to carry out a comprehensive study, that analyzed the tools in several contexts and regarding different necessities. As such, we generated results for a configuration that tried to replicate a real-world scenario, and a configuration focusing each of the base-level criteria: File size, Number of test cases, Test suites' execution time, Percentage of branch coverage and Percentage of total coverage.

These results are shown next. The following tables represent a summarized table from the one generated by our framework, presenting only the tool and its final score, whereas in the framework output we can see the score obtained in each project, besides the final one (an example of this can be seen in C.1).

Besides this, we also established a baseline score in each scenario. This value rep-

resents the score of a tool that does not perform any reduction. As such, applying the formula specified in Section 5.3 of this document, we will have $v_{original} = v_{reduced}$. This causes the benefit criteria values to be 100, and the cost criteria ones to be 0 in all scenarios. The baseline score is influenced by the weights of each criteria. We can then use this default value to evaluate if a tool is of any use in a given scenario, by comparing its score with the baseline one.

Replicating a real-world necessity

As we saw in Chapter 2, the use of Test Suite Reduction is many times associated with the fact that the testing stage is long and costly, so we tried to evaluate the tools by giving greater importance to Time, high importance to Coverage and the lowest importance to Dimension. The pairwise comparison matrices can be seen in C.2 and C.3.

Table 6.6 shows the obtained scores for each tool in the EvoSuite and Normal runs. Figure 6.1 and Figure 6.2 provide us a visual representation of the obtained scores for each tool for this scenario in the form of a chart, for EvoSuite Run and Normal Run, respectively. For better presentation, the kanonizo name was removed from the axis values, being represented only with the algorithm and cut off value (e.g. random#10).

Comparing both tables, we can see that, for the EvoSuite run, the scores range between 0.57 and 0.83, approximately. For the Normal run, the scores range between 0.56 and 0.82, approximately. This similarity in the scores' range shows us that the general behaviour of the tools in both runs does not have a great difference, meaning that, in general, the tools have the same efficiency when reducing tests coded by humans, as well as automatically generated tests.

As for the tools compared, Testler and Randoop were ranked as the worst ones in both runs, with the biggest difference being the several combinations of Kanonizo. EvoSuite came up as the third best tool to reduce its own generated tests, having a score really close to the seconds best. We can also notice that, in both tables, the worst Kanonizo combinations - henceforth denoted by (Kanonizo, algorithm, cut off value) - are the ones with the lowest cut off values. Additionally, we can verify that every tool got a greater score than the baseline.

We can now use these values and compare them with experiments in which we change the importance of each sub-criteria and analyze how it influences the scores of each tool, which we will show next.

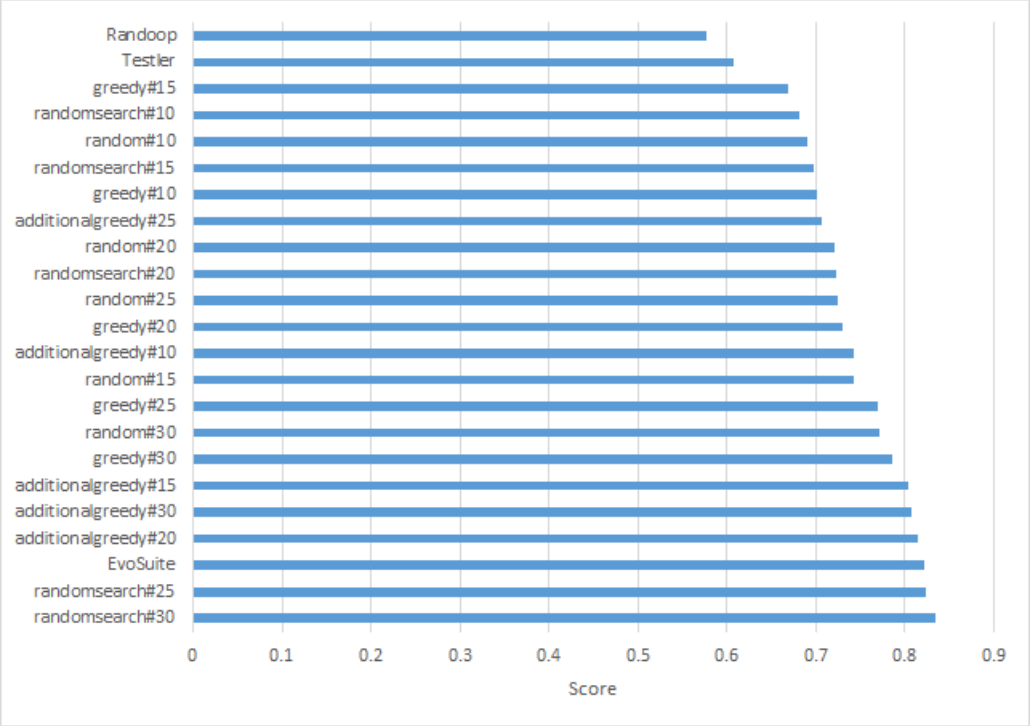

Figure 6.1: Chart for real world scenario results (EvoSuite Run).

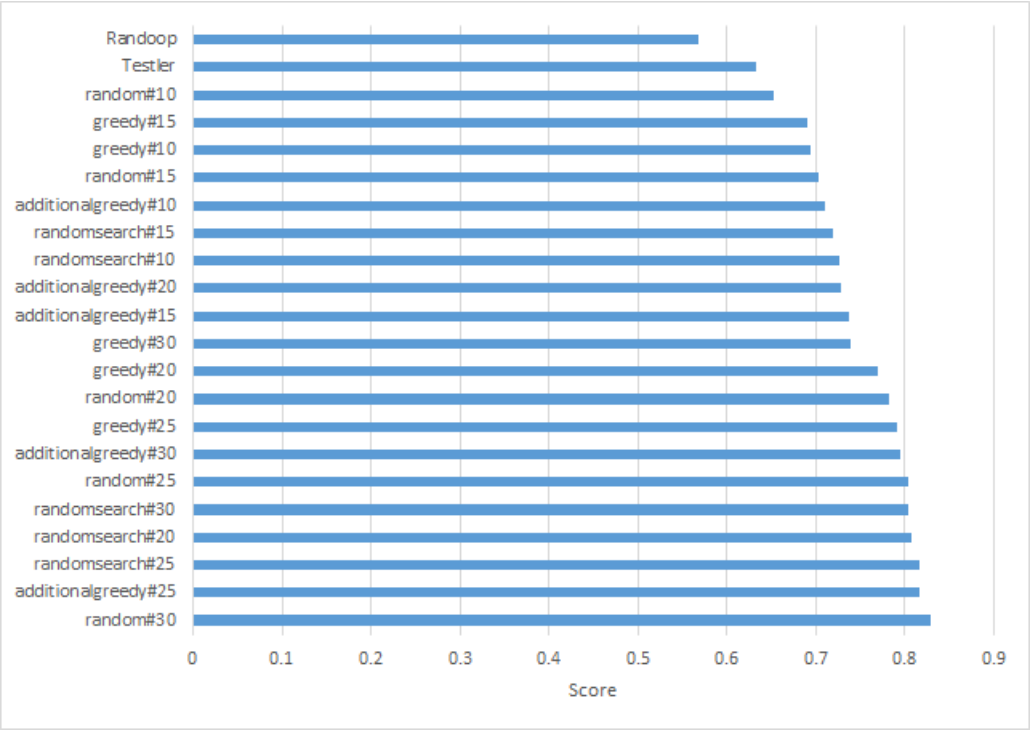

Figure 6.2: Chart for real world scenario results (Normal Run).

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random Search	30	0.83444
Kanonizo	Random Search	25	0.82338
EvoSuite			0.82227
Kanonizo	Additional Greedy	20	0.81469
Kanonizo	Additional Greedy	30	0.80751
Kanonizo	Additional Greedy	15	0.80366
Kanonizo	Greedy	30	0.78591
Kanonizo	Random	30	0.77183
Kanonizo	Greedy	25	0.76982
Kanonizo	Random	15	0.74349
Kanonizo	Additional Greedy	10	0.74295
Kanonizo	Greedy	20	0.73113
Kanonizo	Random	25	0.72421
Kanonizo	Random Search	20	0.72291
Kanonizo	Random	20	0.72126
Kanonizo	Additional Greedy	25	0.70652
Kanonizo	Greedy	10	0.70131
Kanonizo	Random Search	15	0.69871
Kanonizo	Random	10	0.69019
Kanonizo	Random Search	10	0.6811
Kanonizo	Greedy	15	0.66897
Testler			0.60783
Randoop			0.57772
Baseline			0.29464

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random	30	0.82875
Kanonizo	Additional Greedy	25	0.81676
Kanonizo	Random Search	25	0.81664
Kanonizo	Random Search	20	0.80687
Kanonizo	Random Search	30	0.80494
Kanonizo	Random	25	0.8039
Kanonizo	Additional Greedy	30	0.79447
Kanonizo	Greedy	25	0.79134
Kanonizo	Random	20	0.78337
Kanonizo	Greedy	20	0.77041
Kanonizo	Greedy	30	0.7386
Kanonizo	Additional Greedy	15	0.73717
Kanonizo	Additional Greedy	20	0.72834
Kanonizo	Random Search	10	0.72668
Kanonizo	Random Search	15	0.72
Kanonizo	Additional Greedy	10	0.70977
Kanonizo	Random	15	0.7041
Kanonizo	Greedy	10	0.69483
Kanonizo	Greedy	15	0.69044
Kanonizo	Random	10	0.653
Testler			0.63351
Randoop			0.568
Baseline			0.29464

Table 6.6: Scores for real-world scenario.

Focusing specific criterion

We made an evaluation focusing on each of the sub-criteria we evaluated (File size, number of test cases, test suites' execution time, percentage of branch covered and percentage of total coverage). Our goal with these different configurations was to find the best tools specific for each of these metrics and to compare the obtained values with the previously obtained. The pairwise comparison matrices of these experiments all follow the templates shown in Table C.4 and Table C.5, in which the *Evaluated criteria* is the focused criteria, and the *Other criteria* are the remainder. For instance, if we want to evaluate the better tool in reducing the file size of the test suites, we will define the importance of the Dimension criteria related to Time and Coverage to be 9 in both. Additionally, we would define the importance of File size sub-criteria related to Number of test cases to be 9. All the other criteria and sub-criteria will have the same importance between them.

As this is done using a MCDM approach, this obviously means that the values measured for all the other metrics will also be taken into account, although to a minimum, when computing the value of the final score for each tool. By applying the AHP, this would mean that the evaluated sub-criteria will have a weight of 73.(63)% in the final score (0.9 from the file size and number of tests pairwise comparison matrix * 0.(81) from the dimension-time-coverage one), unless it is the evaluated time, given that it is the only sub-criteria of Time, it will not share the weight of Time with any other sub-criteria, and so it will have a weight of 81,(81)%.

Finally, we verify that the established score for the baseline is the same between the

scenarios that focus on cost criteria (0.0909) and between the scenarios that focus on benefit criteria (0.81818). This was expected, since the weight for the focused criterion will always be the same. Additionally, the lower value of the cost criteria can be explained by the fact that file size, number of test cases and test suites' execution times values will be 0, as explained previously, and will therefore weigh more on the final score, bringing it to a final value closer to 0. Contrarily. for the cost criteria we have a score closer to 1.

Figure 6.3 and Figure 6.4 present a chart, for EvoSuite Run and Normal Run respectively, that makes a global appreciation of the results obtained for each tool across all the specific criteria.

We will make an analysis for each of the criteria measured, where we show the tables with the scores for each one, and finish with a general analysis comparing all of the obtained results.

- Dimension

  Tables 6.7 and 6.8 show the results obtained for the sub-criteria of Dimension.

  Regarding file size, we can see that EvoSuite was by far the better tool for the EvoSuite run, while (Kanonizo, Random, 30) was the better tool for the Normal run. We can also notice that the kanonizo combinations are ordered by cut off value, which was expected since a greater cut off value means a smaller file size, therefore a bigger score on file size reduction. A noticeable difference is the position of Randoop in the two runs. As suggested above, the fact that Randoop did better in the normal run might be due to the lack of failing test cases compared with the Normal run.

  Testler's low scores regarding this criterion might seem odd, since its premise is merging several test cases into one. However, what Testler does is marking the merged test cases with the annotation "@Ignore", and so it creates a new big test case, increasing the value of the file sizes.

  EvoSuite's big divergence between its score and the other tools' can be explained by the fact that in almost every test case it will make reductions, drastically reducing the file sizes.

  As for the number of test cases, we can see that, similarly with the file size case, the Kanonizo combinations are sorted by cut off value, for the same explanation we gave before. As expected, Randoop is the least efficient tool for this criterion, since the number of test cases will always be the same in the reduced suite. We can see that Testler got a better position comparing with file size, since now the test cases with "@Ignore" are not considered towards the test cases counting.

  We can also see that every tool got a better score than the baseline. This was expected since the baseline assumes that the file size stays the same. However, regarding file size, we see that Testler (that increases the file size) still got a better score.

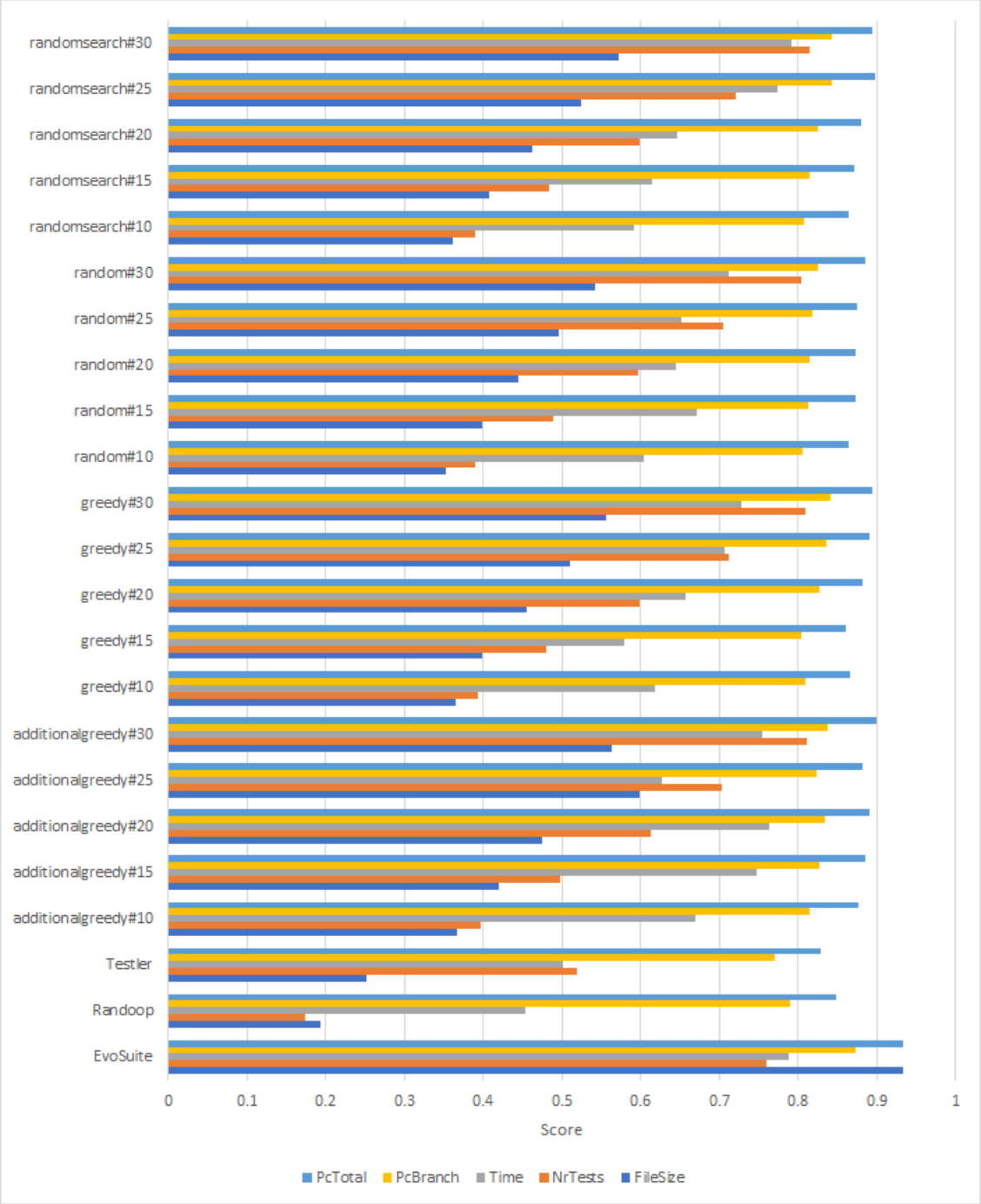

Figure 6.3: Chart with specific criterion results (EvoSuite Run).

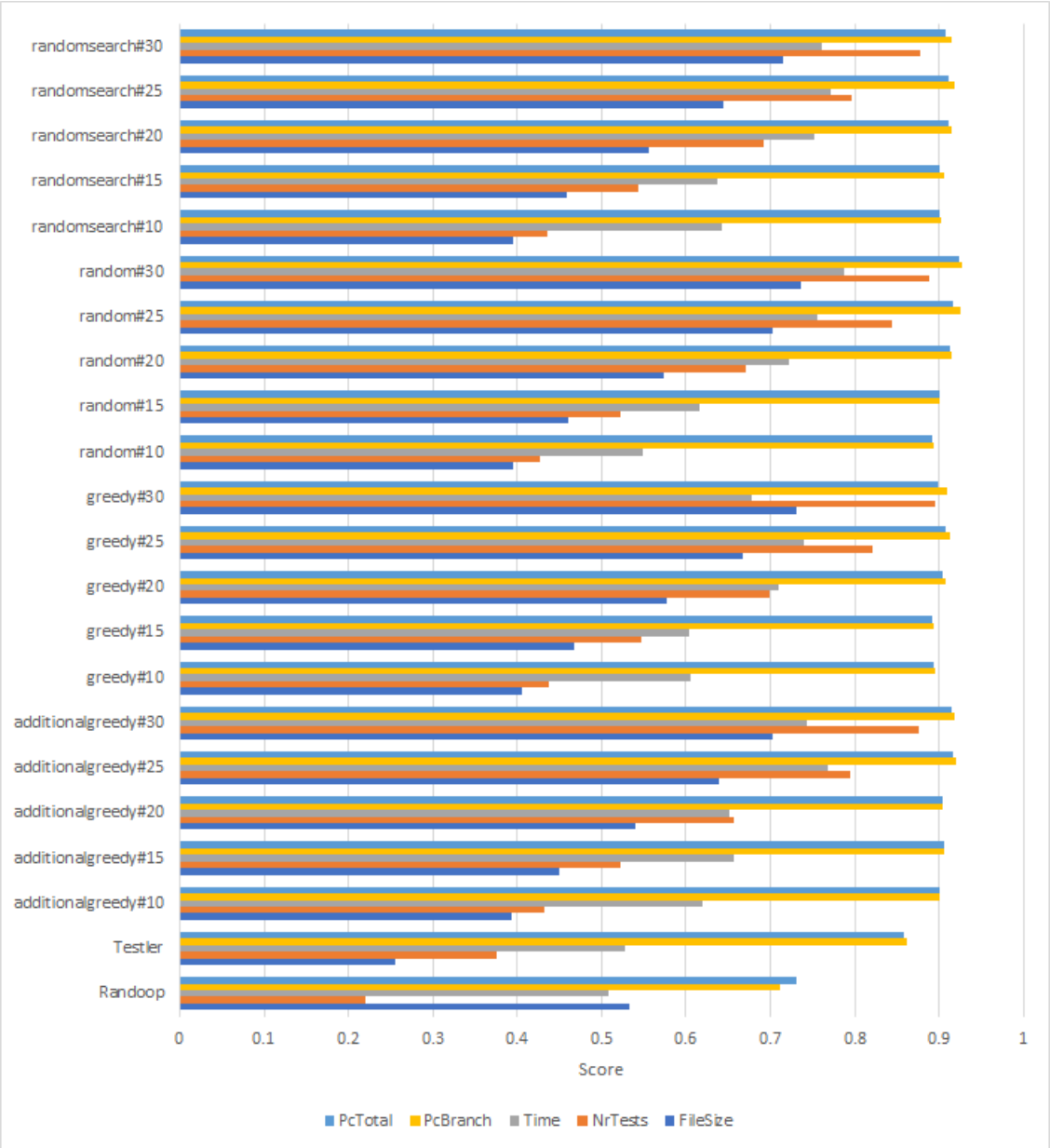

Figure 6.4: Chart with specific criterion results (Normal Run).

This is explained by the contribution of the other criterions, which is also taken into account, as stated before. Similarly, this also explains the better score of Randoop for the test cases scenario.

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
EvoSuite			0.93398
Kanonizo	Random Search	30	0.57147
Kanonizo	Additional Greedy	30	0.56384
Kanonizo	Greedy	30	0.5567
Kanonizo	Random	30	0.54271
Kanonizo	Random Search	25	0.5241
Kanonizo	Greedy	25	0.51078
Kanonizo	Additional Greedy	25	0.50872
Kanonizo	Random	25	0.49528
Kanonizo	Additional Greedy	20	0.4749
Kanonizo	Random Search	20	0.46224
Kanonizo	Greedy	20	0.45622
Kanonizo	Random	20	0.44443
Kanonizo	Additional Greedy	15	0.42083
Kanonizo	Random Search	15	0.40746
Kanonizo	Greedy	15	0.39863
Kanonizo	Random	15	0.39807
Kanonizo	Additional Greedy	10	0.36707
Kanonizo	Greedy	10	0.36496
Kanonizo	Random Search	10	0.36176
Kanonizo	Random	10	0.35269
Testler			0.25189
Randoop			0.1936
Baseline			0.0909

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random	30	0.7353
Kanonizo	Greedy	30	0.73096
Kanonizo	Random Search	30	0.71527
Kanonizo	Additional Greedy	30	0.70228
Kanonizo	Random	25	0.70197
Kanonizo	Greedy	25	0.66706
Kanonizo	Random Search	25	0.64501
Kanonizo	Additional Greedy	25	0.63875
Kanonizo	Greedy	20	0.57652
Kanonizo	Random	20	0.57287
Kanonizo	Random Search	20	0.55658
Kanonizo	Additional Greedy	20	0.54036
Randoop			0.53326
Kanonizo	Greedy	15	0.46718
Kanonizo	Random	15	0.46001
Kanonizo	Random Search	15	0.45852
Kanonizo	Additional Greedy	15	0.44952
Kanonizo	Greedy	10	0.40555
Kanonizo	Random Search	10	0.39566
Kanonizo	Random	10	0.39525
Kanonizo	Additional Greedy	10	0.39253
Testler			0.25468
Baseline			0.0909

Table 6.7: Scores for file size scenario.

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random Search	30	0.81463
Kanonizo	Additional Greedy	30	0.81034
Kanonizo	Greedy	30	0.80876
Kanonizo	Random	30	0.80345
EvoSuite			0.76059
Kanonizo	Random Search	25	0.72044
Kanonizo	Greedy	25	0.71203
Kanonizo	Random	25	0.70491
Kanonizo	Additional Greedy	25	0.70373
Kanonizo	Additional Greedy	20	0.61217
Kanonizo	Greedy	20	0.59915
Kanonizo	Random Search	20	0.59862
Kanonizo	Random	20	0.59619
Testler			0.51984
Kanonizo	Additional Greedy	15	0.49851
Kanonizo	Random	15	0.48822
Kanonizo	Random Search	15	0.4841
Kanonizo	Greedy	15	0.48007
Kanonizo	Additional Greedy	10	0.3976
Kanonizo	Greedy	10	0.39298
Kanonizo	Random	10	0.38956
Kanonizo	Random Search	10	0.38921
Randoop			0.17433
Baseline			0.0909

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Greedy	30	0.89492
Kanonizo	Random	30	0.88764
Kanonizo	Random Search	30	0.87769
Kanonizo	Additional Greedy	30	0.87563
Kanonizo	Random	25	0.84423
Kanonizo	Greedy	25	0.82153
Kanonizo	Random Search	25	0.79657
Kanonizo	Additional Greedy	25	0.79491
Kanonizo	Greedy	20	0.69857
Kanonizo	Random Search	20	0.69122
Kanonizo	Random	20	0.67098
Kanonizo	Additional Greedy	20	0.65624
Kanonizo	Greedy	15	0.54777
Kanonizo	Random Search	15	0.54405
Kanonizo	Random	15	0.52299
Kanonizo	Additional Greedy	15	0.52253
Kanonizo	Greedy	10	0.4366
Kanonizo	Random Search	10	0.43592
Kanonizo	Additional Greedy	10	0.43194
Kanonizo	Random	10	0.42709
Testler			0.37637
Randoop			0.21983
Baseline			0.0909

Table 6.8: Scores for number of test cases scenario.

- Time

  Table 6.9 shows the results when focusing the test suites' Execution time sub-criterion.

  In the case of test suites' execution time, it could be expected that Testler and Randoop could achieve a better score, as they effectively reduce the test cases. However, the raw cut made by Kanonizo was enough to achieve the best reduction on the test suites' execution time.

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random Search	30	0.79258
EvoSuite			0.78763
Kanonizo	Random Search	25	0.77418
Kanonizo	Additional Greedy	20	0.76266
Kanonizo	Additional Greedy	30	0.75473
Kanonizo	Additional Greedy	15	0.74761
Kanonizo	Greedy	30	0.72863
Kanonizo	Random	30	0.71238
Kanonizo	Greedy	25	0.70597
Kanonizo	Random	15	0.6722
Kanonizo	Additional Greedy	10	0.66892
Kanonizo	Greedy	20	0.65634
Kanonizo	Random	25	0.65149
Kanonizo	Random Search	20	0.64634
Kanonizo	Random	20	0.64555
Kanonizo	Additional Greedy	25	0.6267
Kanonizo	Greedy	10	0.61829
Kanonizo	Random Search	15	0.61523
Kanonizo	Random	10	0.60335
Kanonizo	Random Search	10	0.59248
Kanonizo	Greedy	15	0.57954
Testler			0.50142
Randoop			0.45305
Baseline			0.0909

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random	30	0.78711
Kanonizo	Random Search	25	0.77092
Kanonizo	Additional Greedy	25	0.7683
Kanonizo	Random Search	30	0.76082
Kanonizo	Random	25	0.75566
Kanonizo	Random Search	20	0.75261
Kanonizo	Additional Greedy	30	0.74382
Kanonizo	Greedy	25	0.74007
Kanonizo	Random	20	0.72269
Kanonizo	Greedy	20	0.70899
Kanonizo	Greedy	30	0.6786
Kanonizo	Additional Greedy	15	0.65721
Kanonizo	Additional Greedy	20	0.65185
Kanonizo	Random Search	10	0.64249
Kanonizo	Random Search	15	0.63732
Kanonizo	Additional Greedy	10	0.62007
Kanonizo	Random	15	0.61663
Kanonizo	Greedy	10	0.60465
Kanonizo	Greedy	15	0.60289
Kanonizo	Random	10	0.54954
Testler			0.52768
Randoop			0.50781
Baseline			0.0909

Table 6.9: Scores for time scenario.

- Coverage

  Tables 6.10 and 6.11 show the results for the sub-criteria of Coverage.

  We can see that the better tools for each run stay the same in both scenarios, being EvoSuite for the EvoSuite run and (Kanonizo, Random, 30) for the Normal run.

  Once again we see Testler and Randoop at the bottom of the table, but with scores closer to some of the Kanonizo combinations, as opposed to what we verified in some of the other scenarios where they both had the lowest scores. In general this means that every tool got a good performance when maintaining the coverage obtained by the original test suites.

  Regarding the percentage of branch coverage scenario, we see that approximately half of the tools have a lower score than the baseline in the EvoSuite run. This tells us that in this specific scenario, those tools did not bring any improvement to the test suites, which is normal, since the decrease of coverage is a known issue of Test Suite Reduction (TSR).

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
EvoSuite			0.87329
Kanonizo	Random Search	30	0.84363
Kanonizo	Random Search	25	0.84354
Kanonizo	Greedy	30	0.84055
Kanonizo	Additional Greedy	30	0.83793
Kanonizo	Greedy	25	0.83637
Kanonizo	Additional Greedy	20	0.83471
Kanonizo	Additional Greedy	15	0.82685
Kanonizo	Greedy	20	0.82674
Kanonizo	Random	30	0.82516
Kanonizo	Random Search	20	0.82493
Kanonizo	Additional Greedy	25	0.8227
Baseline			0.81818
Kanonizo	Random	25	0.81811
Kanonizo	Additional Greedy	10	0.81505
Kanonizo	Random Search	15	0.81473
Kanonizo	Random	20	0.81451
Kanonizo	Random	15	0.81291
Kanonizo	Greedy	10	0.80997
Kanonizo	Random Search	10	0.80713
Kanonizo	Random	10	0.80663
Kanonizo	Greedy	15	0.80354
Randoop			0.79073
Testler			0.77056

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random	30	0.92671
Kanonizo	Random	25	0.92493
Kanonizo	Additional Greedy	25	0.92067
Kanonizo	Additional Greedy	30	0.91882
Kanonizo	Random Search	25	0.91827
Kanonizo	Random Search	30	0.91531
Kanonizo	Random Search	20	0.91512
Kanonizo	Random	20	0.91505
Kanonizo	Greedy	25	0.91278
Kanonizo	Greedy	30	0.90874
Kanonizo	Greedy	20	0.90778
Kanonizo	Additional Greedy	15	0.90536
Kanonizo	Random Search	15	0.90505
Kanonizo	Additional Greedy	20	0.90485
Kanonizo	Random Search	10	0.90166
Kanonizo	Random	15	0.90031
Kanonizo	Additional Greedy	10	0.89996
Kanonizo	Greedy	10	0.89561
Kanonizo	Greedy	15	0.89334
Kanonizo	Random	10	0.89284
Testler			0.86143
Baseline			0.81818
Randoop			0.71218

Table 6.10: Scores for percentage of branch coverage scenario.

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
EvoSuite			0.93304
Kanonizo	Additional Greedy	30	0.9
Kanonizo	Random Search	25	0.89846
Kanonizo	Random Search	30	0.89497
Kanonizo	Greedy	30	0.89392
Kanonizo	Additional Greedy	20	0.89164
Kanonizo	Greedy	25	0.89161
Kanonizo	Additional Greedy	15	0.88539
Kanonizo	Random	30	0.88496
Kanonizo	Additional Greedy	25	0.88201
Kanonizo	Greedy	20	0.88141
Kanonizo	Random Search	20	0.88047
Kanonizo	Additional Greedy	10	0.87577
Kanonizo	Random	25	0.87499
Kanonizo	Random	20	0.87315
Kanonizo	Random	15	0.87284
Kanonizo	Random Search	15	0.87114
Kanonizo	Greedy	10	0.86589
Kanonizo	Random	10	0.86464
Kanonizo	Random Search	10	0.86379
Kanonizo	Greedy	15	0.86097
Randoop			0.84898
Testler			0.82834
Baseline			0.81818

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Random	30	0.92435
Kanonizo	Random	25	0.91693
Kanonizo	Additional Greedy	25	0.91692
Kanonizo	Additional Greedy	30	0.91475
Kanonizo	Random	20	0.91253
Kanonizo	Random Search	20	0.91124
Kanonizo	Random Search	25	0.91087
Kanonizo	Greedy	25	0.9083
Kanonizo	Random Search	30	0.90827
Kanonizo	Additional Greedy	15	0.90631
Kanonizo	Additional Greedy	20	0.90406
Kanonizo	Greedy	20	0.90404
Kanonizo	Random	15	0.90127
Kanonizo	Additional Greedy	10	0.90125
Kanonizo	Random Search	15	0.90123
Kanonizo	Random Search	10	0.90091
Kanonizo	Greedy	30	0.89938
Kanonizo	Greedy	10	0.89289
Kanonizo	Random	10	0.89243
Kanonizo	Greedy	15	0.89134
Testler			0.85772
Baseline			0.81818
Randoop			0.73104

Table 6.11: Scores for percentage of total coverage scenario.

Overall, comparing the results obtained for each criteria and the ones from the real world scenario, we see that the tools that obtained the best scores in the latter: (Kanonizo, Random search, 30) for EvoSuite run, (Kanonizo, random, 30) for Normal run; reflect the ones we got in the specific criterion scenarios, since, in the case of the EvoSuite run, (Kanonizo, Random search, 30) obtained the best score for 2 out of the 5 criteria, and the second best for another 2; in the case of the Normal run, (Kanonizo, Random, 30) obtained the best score in 4 out of 5 categories.

Changing the project's weight

Besides the described experiments, we also tried to change each project weight on the final score, to see if this influenced the ranking of the tools. We wanted to replicate the projects usefulness in this weight, and as such we defined the weight as the number of dependents (seen in Table 4.1). The results are shown in Table 6.12.

As the project with the greatest number of dependents was commons-lang by a wide margin, the results we see here reflect the scores obtained when reducing this project.

Judging by all the results, we can also assume that the studied tools worked similarly in both runs, and, as such, the fact that the tests are automatically generated or written by humans probably have no influence in the general efficiency of these tools.

EvoSuite Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Greedy	30	0.917864205
Kanonizo	Greedy	25	0.901134847
Kanonizo	Additional Greedy	30	0.900890049
Kanonizo	Greedy	20	0.88595514
Kanonizo	Random Search	25	0.881493084
Kanonizo	Random Search	30	0.878056736
Kanonizo	Additional Greedy	25	0.871722278
Kanonizo	Additional Greedy	15	0.842900159
Kanonizo	Greedy	15	0.842759073
Kanonizo	Random Search	20	0.836249533
Kanonizo	Additional Greedy	20	0.835796222
EvoSuite			0.829700137
Kanonizo	Random Search	15	0.820821786
Kanonizo	Random	30	0.803754629
Testler			0.779436033
Kanonizo	Random	15	0.755862711
Kanonizo	Random	20	0.75386665
Kanonizo	Random	25	0.741018508
Kanonizo	Additional Greedy	10	0.734939692
Kanonizo	Random Search	10	0.728486757
Kanonizo	Greedy	10	0.720150547
Kanonizo	Random	10	0.692061793
Randoop			0.325831898
Baseline			0.29464

Normal Run			
Tool	**Algorithm**	**Cut off**	**Score**
Kanonizo	Greedy	20	0.836798281
Randoop			0.830177957
Kanonizo	Greedy	30	0.824732249
Kanonizo	Greedy	25	0.821115728
Kanonizo	Random Search	20	0.803329693
Kanonizo	Random Search	25	0.797382353
Kanonizo	Random Search	30	0.792960079
Kanonizo	Additional Greedy	30	0.78878095
Kanonizo	Greedy	15	0.782450928
Kanonizo	Random Search	15	0.77842572
Kanonizo	Greedy	10	0.759499704
Kanonizo	Random	30	0.756159272
Kanonizo	Random	25	0.75443151
Kanonizo	Additional Greedy	25	0.736019717
Kanonizo	Random	20	0.730513078
Kanonizo	Additional Greedy	15	0.712528468
Kanonizo	Random Search	10	0.709789467
Kanonizo	Additional Greedy	10	0.706911089
Kanonizo	Random	10	0.706339709
Kanonizo	Random	15	0.698103489
Kanonizo	Additional Greedy	20	0.673654884
Testler			0.388814691
Baseline			0.29464

Table 6.12: Scores with changed project weights.

6.3 Discussion

Our main goal for this work was to answer two questions: “What is the best Test Suite Reduction tool?” (according to some criterion) and “Can a Test Case Prioritization tool be adapted to effectively replace a Test Suite Reduction tool?” (according to some criterion and threshold value).

Table 6.13 shows us the average of scores obtained in both runs for each tool. We chose to agglomerate all the Kanonizo’s variations into the same row, for better visualization. After conducting several experiments, we can safely say that the tool that was more efficient was Kanonizo in the Normal Run, having an average score difference of 0.16295 to Testler’s and of 0.20283 to Randoop’s.

As for the EvoSuite Run, we see that the order between Kanonizo, Testler and Randoop is maintained. However, EvoSuite has the greatest average score, having a difference of 0.1538 to Kanonizo’s, 0.27802 to Testler’s and 0.3454 to Randoop’s. Although EvoSuite has a greater average score than Kanonizo, we must notice that in three out of six scenarios of our exeperiment - real-world, number of test cases and execution time - some variation of Kanonizo was better than EvoSuite, and that the average score of Kanonizo is heavily influenced by the lowest scores obtained in its less efficient variations. Taking these factors into account, and considering that the real-world scenario is undoubtedly more common than any other one, and that EvoSuite only reduces tests generated by it, we could argue that there is a variation of Kanonizo that is more efficient than EvoSuite, for a real-world usage.

Considering just Test Suite Reduction tools, Testler achieved a greater score than Ran-

EvoSuite Run	Average Score
EvoSuite	0.8518
Kanonizo	0.69800
Testler	0.57998
Randoop	0.50640

Normal Run	Average Score
Kanonizo	0.74818
Testler	0.58523
Randoop	0.54535

Table 6.13: Average scores in both runs.

doop in both runs, which is no surprise, since Randoop only focuses on failing test cases. On average, we see that Testler got a score that exceeds the Randoop's score by 14.53% in the EvoSuite run, and by 7.31%, approximately, in the Normal run.

It is important to notice that we could only use EvoSuite to reduce tests generated by it, which could influence its effectiveness, since its reduction is made considering test data gathered in generation time. If EvoSuite had the same performance reducing original test suites, we could say that it was the best TSR tool, compared to Testler and Randoop.

Regarding our second question, and speaking in the context of our experiments, all Kanonizo combinations were better than any reduction tool in the normal run, and aside from EvoSuite and Testler in one scenario (number of test cases), the same happened in the EvoSuite run. Table 6.14 shows us the average score obtained for each Kanonizo variation, in both runs. It is clear that, of all the cut off values tried, the 30% one was the best, as the three best variations in both runs had a cut off value of 30%, representing a considerable reduction in the test suite size without compromising the coverage obtained. As for the chosen algorithm, the one that achieved a better score on average was the Random Search for the EvoSuite run, with a value of 0.79195, and Random for the Normal run, with a value of 0.84831. Although we could argue that the results obtained by Random are too volatile, and we could consider Random Search to be more trustful, with an average score of 0.83038 in the Normal run.

Looking at Chapter 2, we said that the TSR tools should have a set of characteristics: inclusiveness, precision, efficiency and generality. Since Kanonizo is not a TSR tool, and it was our changes that made it possible to use it as such, we would like to analyze if its behaviour confirms these properties, using the results gathered for each of the scenarios. As for inclusiveness, we saw that all combinations of Kanonizo reductions achieved a good branch coverage, not perfect but still better than Testler and Randoop. Regarding precision, since the Kanonizo reductions are made in a rather raw method, and we can always set a greater cut off value, this property does not apply to this tool. For efficiency, since the Kanonizo execution times are near Testler's or Randoop's, it is safe to say that it has this characteristic. Lastly, regarding generality, we can say that Kanonizo also has this property since we tested it on a very comprehensive set of programs (11 projects in each run, which gives us 22 differents sets of test suites).

EvoSuite Run		
Algorithm	**Cut off**	**Average Score**
Random Search	30	0.79195
Additional Greedy	30	0.77906
Greedy	30	0.76908
Random Search	25	0.76402
Random	30	0.75675
Greedy	25	0.73776
Additional Greedy	20	0.7318
Random	25	0.7115
Additional Greedy	25	0.7084
Additional Greedy	15	0.69714
Greedy	20	0.69183
Random Search	20	0.68925
Random	20	0.68252
Random	15	0.66462
Random Search	15	0.64856
Additional Greedy	10	0.64456
Greedy	15	0.63195
Greedy	10	0.62557
Random	10	0.61784
Random Search	10	0.61591

Normal Run		
Algorithm	**Cut off**	**Average Score**
Random	30	0.84831
Random Search	30	0.83038
Additional Greedy	30	0.82496
Random	25	0.8246
Random Search	25	0.80971
Additional Greedy	25	0.80939
Greedy	30	0.80853
Greedy	25	0.80685
Random Search	20	0.77227
Random	20	0.76292
Greedy	20	0.76105
Additional Greedy	20	0.73095
Additional Greedy	15	0.69635
Random Search	15	0.69436
Random	15	0.68422
Greedy	15	0.68216
Random Search	10	0.66722
Additional Greedy	10	0.65925
Greedy	10	0.65502
Random	10	0.63503

Table 6.14: Average scores of Kanonizo variations in both runs.

Having these aspects in mind, with the tools we compared, we can clearly say that the TCP tool is a viable option, and could even replace the TSR tools if we were to remove the least important test cases from each test suite. However, this depends on the tool that is used. For instance, using Kanonizo with the random algorithm, we are blinding removing test cases, since the ordering of the test cases is completely random, and there is always a risk of removing test cases that are more important than others. As such, it is preferable to use a prioritization algorithm that prioritizes tests according to some criterion, as this is more similar with some of the TSR approaches.

Chapter 7

Conclusion

In this chapter, a summary about the whole work done is presented, regarding the obtained results, the problems found throughout this work and the future work we could do regarding this framework and study.

As we stated before, our main objective with this work was to answer two questions: "What is the best Test Suite Reduction tool?" and "Can a Test Case Prioritization tool be adapted to effectively replace a Test Suite Reduction tool?". The framework was designed with the first question in mind since the very beginning, and later with the second one too. As such, we tried to implement it while always thinking in the best way to make it extensible to other tools, projects and criteria.

As far as we know, the development of this framework, with all of its functionalities, is something that has not been done yet. The possibility to compare a set of reduction tools, and even the study about the efficiency of the TCP tool with a cut on the test suites, is important to the actual state of the art, since in the last years there has been a huge increase in software development, with the growing need to spend less and less time testing it. This calls for a need to know which are the better tools to reduce the time spent testing. Although this framework works, its usability depends on the correctness of the integrated tools, which is the field that gave us more trouble.

7.1 Results

With the several experiments we conducted, described in the last chapter, we found the answer to the questions we proposed to answer in the beginning of this work. The results showed us that some Kanonizo variation performed better than the other considered tools in all scenarios, which gives us the confidence to state that a TCP tool can apply a reduction in a more effective way than some TSR tools, when we filter out some of the most unimportant test cases. We also verified that our changed Kanonizo complies with three out of the four presented characteristics that every TSR tool should have.

Regarding just the TSR tools, we noticed that EvoSuite got the best score in all sce-

narios. Since it can only reduce its own generated tests, it is also important to notice that Testler got the best score when compared to Randoop.

7.2 Problems

One of the biggest problems faced in this work was choosing and integrating the TSR tools to conduct the experiment. As we saw in Chapter 3, there are a lot of tools that are simply not made available by the developers, and some that are available but only in a specific platform, like as a web application or an IDE plugin. Those that were available, sometimes did not work as expected and were not actively maintained by their developers, and had little to no documentation, which delayed the whole process of understanding how it worked, and why it did not work when it was giving problems.

Most of the studied tools had reduction integrated as a part of a bigger function, for instance, the automatic generation of test suites. This was a problem, given that our interest was in the reduction of existing test suites. As such, we can say that although there are a lot of research being done in this area, there are some lack of robust and supported TSR tools and frameworks, that focus entirely on this function, as is the case of Testler (besides all the problems found).

This all had a negative impact in the duration of this work, since we had to adapt to the very few options we had available and try and get the maximum number of tools to work. Hence the creation of a second goal with the TCP tool.

7.3 Future work

As for future work, there are some additions to the framework that could be done.

For instance, the integration of other code analysis tools, besides OpenClover, and give the user the option to choose which one to use. This could also make for an interesting comparative study about the way the analysis of each one of these tools influence the final score obtained for a TSR tool. In addition to this, we could also implement other methods to compute the score for each tool, aside from AHP.

Apart from this, there is always the option of integrating future TSR tools, which is easily done through the configuration file.

Finally, we expect to publish a paper where we make an analysis of the results obtained from our study and to make the implemented framework available online as an open source project.

Bibliography

[1] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, (New York, NY, USA), p. 85–91, Association for Computing Machinery, 2006.

[2] "10 historical software bugs with extreme consequences - pingdom royal." `https://royal.pingdom.com/10-historical-software-bugs-with-extreme-consequences/`. Accessed: 23-08-2019.

[3] D. Binkley, "The application of program slicing to regression testing," *Information and Software Technology*, vol. 40, no. 11, pp. 583–593, 1998.

[4] X.-y. Ma, B.-k. Sheng, and C.-q. Ye, "Test-suite reduction using genetic algorithm," in *Advanced Parallel Processing Technologies* (J. Cao, W. Nejdl, and M. Xu, eds.), (Berlin, Heidelberg), pp. 253–262, Springer Berlin Heidelberg, 2005.

[5] S. U. R. Khan, S. Peck Lee, R. Ahmad, A. Akhunzada, and V. Chang, "A survey on test suite reduction frameworks and tools," *International Journal of Information Management*, vol. 36, pp. 963–975, 12 2016.

[6] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929–948, Oct 2001.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, 03 2012.

[8] J. A. Jones, M. Harrold, and I. Computer Society, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, 06 2003.

[9] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, pp. 159–182, Feb 2002.

[10] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *1995 17th International Conference on Software Engineering*, pp. 41–41, April 1995.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," in *Proceedings. Conference on Software Maintenance 1990*, pp. 302–310, Nov 1990.

[12] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Softw. Test., Verif. Reliab.*, vol. 12, pp. 219–249, 2002.

[13] M. Alian, D. Suleiman, and A. Shaout, "Test case reduction techniques - survey," *International Journal of Advanced Computer Science and Applications*, vol. 7, pp. 264–275, 06 2016.

[14] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, (Berlin, Heidelberg), p. 291–305, Springer-Verlag, 2007.

[15] Z. Chen, B. Xu, X. Zhang, and C. Nie, "A novel approach for test suite reduction based on requirement relation contraction," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, (New York, NY, USA), p. 390–394, Association for Computing Machinery, 2008.

[16] R. Singh and M. Santosh, "Test case minimization techniques : A review," *IJERT*, vol. 2, pp. 1048–1056, 12 2013.

[17] S. Nachiyappan, A. Vimaladevi, and C. B. SelvaLakshmi, "An evolutionary algorithm for regression test suite reduction," in *2010 International Conference on Communication and Computational Intelligence (INCOCCI)*, pp. 503–508, 2010.

[18] L. You and Y. Lu, "A genetic algorithm for the time-aware regression testing reduction problem," in *2012 8th International Conference on Natural Computation*, pp. 596–599, 2012.

[19] S. K. Mohapatra and M. Pradhan, "Finding representative test suit for test case reduction in regression testing," in *2015 International Conference on Computer, Communication and Control (IC4)*, pp. 1–6, 2015.

[20] Xue-ying MA, Zhen-feng He, Bin-kui Sheng, and Cheng-qing Ye, "A genetic algorithm for test-suite reduction," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, pp. 133–139 Vol. 1, 2005.

[21] R. Wang, B. Qu, and Y. Lu, "Empirical study of the effects of different profiles on regression test case reduction," *IET Software*, vol. 9, no. 2, pp. 29–38, 2015.

[22] B. Subashini and D. JeyaMala, "Reduction of test cases using clustering technique," in *International Journal of Innovative Research in Science, Engineering and Technology, 2014 International Conference on Innovations*, vol. 3, pp. 1993–1996, 2014.

[23] S. Parsa, A. Khalilian, and Y. Fazlalizadeh, "A new algorithm to test suite reduction based on cluster analysis," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pp. 189–193, 2009.

[24] A. Haider, S. Rafiq, and A. Nadeem, "Test suite optimization using fuzzy logic," *2012 International Conference on Emerging Technologies*, pp. 1–6, 2012.

[25] A. A. Haider, A. Nadeem, and S. Rafiq, "Computational intelligence and safe reduction of test suite," in *2013 IEEE 9th International Conference on Emerging Technologies (ICET)*, pp. 1–6, 2013.

[26] A. A. Haider, A. Nadeem, and S. Rafiq, "On the fly test suite optimization with fuzzyoptimizer," in *2013 11th International Conference on Frontiers of Information Technology*, pp. 101–106, 2013.

[27] C. Murphy, Z. Zoomkawalla, and K. Narita, "Automatic test case generation and test suite reduction for closed-loop controller software," 01 2013.

[28] P. Pringsulaka and J. Daengdej, "Coverall algorithm for test case reduction," in *2006 IEEE Aerospace Conference*, pp. 8 pp.–, 2006.

[29] S. Roongruangsuwan and J. Daengdej, "Test case reduction methods by using cbr," *CEUR Workshop Proceedings*, vol. 646, pp. 75–82, 01 2010.

[30] I. Watson and F. Marir, "Case-based reasoning: A review," *The Knowledge Engineering Review*, vol. 9, no. 4, p. 327–354, 1994.

[31] S. Khan, A. Nadeem, and A. Awais, "Testfilter: A statement-coverage based test case reduction technique," in *2006 IEEE International Multitopic Conference*, pp. 275–280, 2006.

[32] B. Arasteh, "Using program slicing technique to reduce the cost of software testing," *Journal of Artificial Intelligence in Electrical Engineering*, vol. 2, pp. 24–33, 11 2013.

[33] D. W. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Software Eng.*, vol. 23, pp. 498–516, 1997.

[34] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *SIGSOFT Softw. Eng. Notes*, vol. 31, p. 35–42, Sept. 2005.

[35] S. Xu, H. Miao, and H. Gao, "Test suite reduction using weighted set covering techniques," in *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 307–312, 2012.

[36] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, (USA), p. 170–179, IEEE Computer Society, 2011.

[37] B. Suri, I. Mangal, and V. Srivastava, "Regression test suite reduction using an hybrid technique based on bco and genetic algorithm," *Special Issue of International Journal of Computer Science & Informatics (IJCSI)*, pp. 2231–5292, 01 2011.

[38] S. Sampath, R. Bryce, and A. M. Memon, "A uniform representation of hybrid criteria for regression testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1326–1344, 2013.

[39] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689 – 701, 2010.

[40] J. R. Horgan and S. London, "A data flow coverage testing tool for c," in *[1992] Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, pp. 2–10, May 1992.

[41] Tao Xie, D. Notkin, and D. Marinov, "Rostra: a framework for detecting redundant object-oriented unit tests," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 196–205, 2004.

[42] T. Xie, J. Zhao, D. Marinov, and D. Notkin, "Detecting redundant unit tests for aspectj programs," in *2006 17th International Symposium on Software Reliability Engineering*, pp. 179–190, 2006.

[43] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in *Proceedings of the 1st International Workshop on Random Testing*, RT '06, (New York, NY, USA), p. 36–45, Association for Computing Machinery, 2006.

[44] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, (New York, NY, USA), p. 815–816, Association for Computing Machinery, 2007.

[45] H. Jaygarl, K. Lu, and C. K. Chang, "Genred: A tool for generating and reducing object-oriented test cases," in *2010 IEEE 34th Annual Computer Software and Applications Conference*, pp. 127–136, 2010.

[46] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Jtop: Managing junit test cases in absence of coverage information," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (USA), p. 677–679, IEEE Computer Society, 2009.

[47] F. Dadeau, Y. Ledru, and L. Du Bousquet, "Directed random reduction of combinatorial test suites," in *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, (New York, NY, USA), p. 18–25, Association for Computing Machinery, 2007.

[48] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software*, vol. 103, pp. 370 – 391, 2015.

[49] J. M. Kauffman and G. M. Kapfhammer, "A framework to support research in and encourage industrial adoption of regression testing techniques," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 907–908, April 2012.

[50] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 416–419, ACM, 2011.

[51] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 210–221, ACM, 2018.

[52] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter Greenwald, "Applying concept analysis to user-session-based testing of web applications," *IEEE Transactions on Software Engineering*, vol. 33, pp. 643–658, Oct 2007.

[53] S. Sampath, R. C. Bryce, S. Jain, and S. Manchester, "A tool for combination-based prioritization and reduction of user-session-based test suites," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 574–577, 2011.

[54] G. Woo, H. S. Chae, and H. Jang, "An intermediate representation approach to reducing test suites for retargeted compilers," in *International Conference on Reliable Software Technologies*, pp. 100–113, Springer, 2007.

[55] H. S. Chae, G. Woo, T. Y. Kim, J. H. Bae, and W.-Y. Kim, "An automated approach to reducing test suites for testing retargeted c compilers for embedded systems," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2053 – 2064, 2011.

[56] H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (USA), p. 419–429, IEEE Computer Society, 2009.

[57] D. Li, C. Sahin, J. Clause, and W. G. J. Halfond, "Energy-directed test suite optimization," in *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pp. 62–69, 2013.

[58] X. Zhang, Q. Gu, X. Chen, J. Qi, and D. Chen, "A study of relative redundancy in test-suite reduction while retaining or improving fault-localization effectiveness," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, (New York, NY, USA), p. 2229–2236, Association for Computing Machinery, 2010.

[59] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), p. 221–231, Association for Computing Machinery, 2011.

[60] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, 09 2012.

[61] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *Journal of King Saud University - Computer and Information Sciences*, 2018.

[62] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74 – 93, 2018.

[63] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pp. 179–188, Aug 1999.

[64] D. Paterson, G. Kapfhammer, G. Fraser, and P. McMinn, "Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization," in *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*, pp. 57–63, May 2018.

[65] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 713–716, May 2015.

[66] P. Meirelles, C. Santos Jr., J. Miranda, F. Kon, A. Terceiro, and C. Chavez, "A study of the relationships between source code metrics and attractiveness in free software projects," in *2010 Brazilian Symposium on Software Engineering*, pp. 11–20, Sep. 2010.

[67] S. Greco, J. Figueira, and M. Ehrgott, *Multiple criteria decision analysis*. Springer, 2016.

[68] V. Belton and T. Stewart, *Multiple criteria decision analysis: an integrated approach*. Springer Science & Business Media, 2002.

[69] M. Velasquez and P. Hester, "An analysis of multi-criteria decision making methods," *International Journal of Operations Research*, vol. 10, pp. 56–66, 05 2013.

[70] R. Saaty, "The analytic hierarchy process—what it is and how it is used," *Mathematical Modelling*, vol. 9, no. 3, pp. 161 – 176, 1987.

[71] T. L. Saaty, "Decision making with the analytic hierarchy process," *International journal of services sciences*, vol. 1, no. 1, pp. 83–98, 2008.

[72] M. Martínez, D. D. Andrés, and J.-C. Ruiz, "Gaining confidence on dependability benchmarks' conclusions through "back-to-back" testing (practical experience report)," in *Proceedings of the 2014 Tenth European Dependable Computing Conference*, EDCC '14, (USA), p. 130–137, IEEE Computer Society, 2014.

[73] N. Vafaei, R. A. Ribeiro, and L. M. Camarinha-Matos, "Normalization techniques for multi-criteria decision making: analytical hierarchy process case study," in *doctoral conference on computing, electrical and industrial systems*, pp. 261–269, Springer, 2016.

[74] A. Çelen, "Comparative analysis of normalization procedures in topsis method: With an application to turkish deposit banking market," *Informatica, Lith. Acad. Sci.*, vol. 25, pp. 185–208, 2014.

[75] A. Jahan and K. L. Edwards, "A state-of-the-art survey on the influence of normalization techniques in ranking: Improving the materials selection process in engineering design," *Materials & Design (1980-2015)*, vol. 65, pp. 335 – 342, 2015.

[76] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84, May 2007.

[77] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*, vol. 500. US Department of Commerce, Technology Administration, National Institute of . . . , 1996.

[78] Y. Ledru, "Tobias : a tool for combinatorial testing." `http://tobias.liglab.fr/`. Accessed: 22-10-2018.

[79] "Atac – automatic test analysis for c programs." `https://invisible-island.net/atac/`. Accessed: 17-11-2018.

Appendix A

Selected frameworks and tools

```
group SimpleSequence[us=true] {
 Purse p = new Purse();
 p.credit([10, 50, 0]);
 p.debit([5,15]);
 p.getBalance();
}
```

Listing A.1: Tobias input file example

Appendix B

Implementation

```
<?xml version="1.0" encoding="UTF-8"?>
<config>

  <mode>Full</mode>

  <!-- The path where the reduced projects will be written -->
  <reducedProjectsPath>D:\Reduced\</reducedProjectsPath>

  <reportsPath></reportsPath> <!-- The path where the reports will be
    written -->

  <outputPath>D:\Estudo</outputPath> <!-- The path where the output
    will be saved -->

  <evosuite>false</evosuite> <!-- should use evosuite -->

  <kanonizo> <!-- Kanonizo configuration -->
    <use>true</use> <!-- should use kanonizo -->
    <algorithms> <!-- algorithms to use -->
      <algorithm use="true">random</algorithm>
      <algorithm use="false">greedy</algorithm>
      <algorithm use="false">randomsearch</algorithm>
      <algorithm use="false">additionalgreedy</algorithm>
    </algorithms>
    <cutOffs> <!-- percentages to cut the test Suites -->
      <cutOff>15</cutOff>
      <cutOff>20</cutOff>
      <cutOff>25</cutOff>
    </cutOffs>
  </kanonizo>

  <!-- The TSR tools to run in the experiment -->
  <TSRtools>
    <tool>
      <name>Testler</name>
      <jar>path/to/jar</jar> <!-- optional -->
    </tool>
    <tool>
      <name>Randoop</name>
      <jar>path/to/jar</jar>
```

```
  </tool>
</TSRtools>

<!--
  The values of 'importance' must be given as a
  comparison between criteria and according to
  these values:

  1 - Equal importance
  3 - Moderate importance
  5 - Strong importance
  7 - Very strong importance
  9 - Extreme importance

  2,4,6,8 values can be used for
  intermediate importance values

  if X has an importance z related to Y,
  then Y should have an importance 1/z related to X
 -->
<ahp>
  <criteria>
    <name>Dimension</name>
    <importance>2</importance>
    <importance>1/2</importance>
    <subcriteria>
      <criteria>
        <name>File Size</name>
        <importance>1</importance>
      </criteria>
      <criteria>
        <name># of Test Cases</name>
        <importance >1</importance>
      </criteria>
    </subcriteria>
  </criteria>
  <criteria>
    <name>Coverage</name>
    <importance>1/2</importance>
    <importance>1/5</importance>
    <subcriteria>
      <criteria>
        <name>% Branches Covered</name>
        <importance>1</importance>
      </criteria>
      <criteria>
        <name>% Total Coverage</name>
        <importance>1</importance>
      </criteria>
    </subcriteria>
  </criteria>
  <criteria>
    <name>Time</name>
    <importance>2</importance>
    <importance>5</importance>
    <subcriteria>
```

```
        <criteria>
          <name>Tests Execution Time</name>
        </criteria>
      </subcriteria>
    </criteria>
  </ahp>

  <projects> <!-- The subjects of the experiment -->
    <project>
      <name>project_name_1</name>
      <pathToPom>Path/to/pom/1</pathToPom>
      <weight>1</weight> <!-- Weight for the final score -->
    </project>
    <project>
      <name>project_name_2</name>
      <pathToPom>Path/to/pom/2</pathToPom>
      <weight>1</weight>
    </project>
      <project>
      <name>project_name_3</name>
      <pathToPom>Path/to/pom/3</pathToPom>
      <weight>1</weight>
    </project>
  </projects>
</config>
```

Listing B.1: config.xml

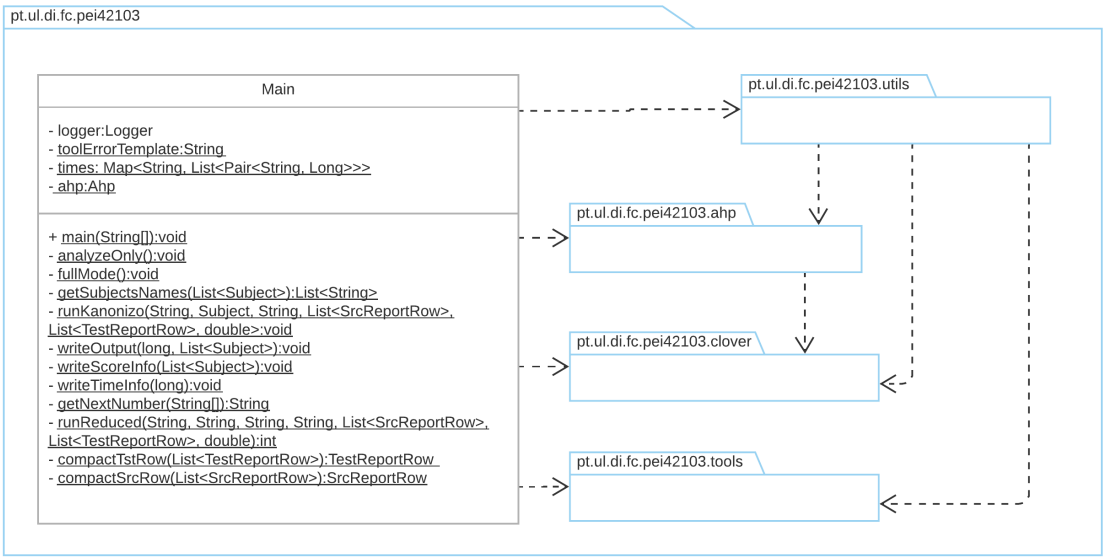

Figure B.1: Package pt.ul.di.fc.pei42103 class diagram.

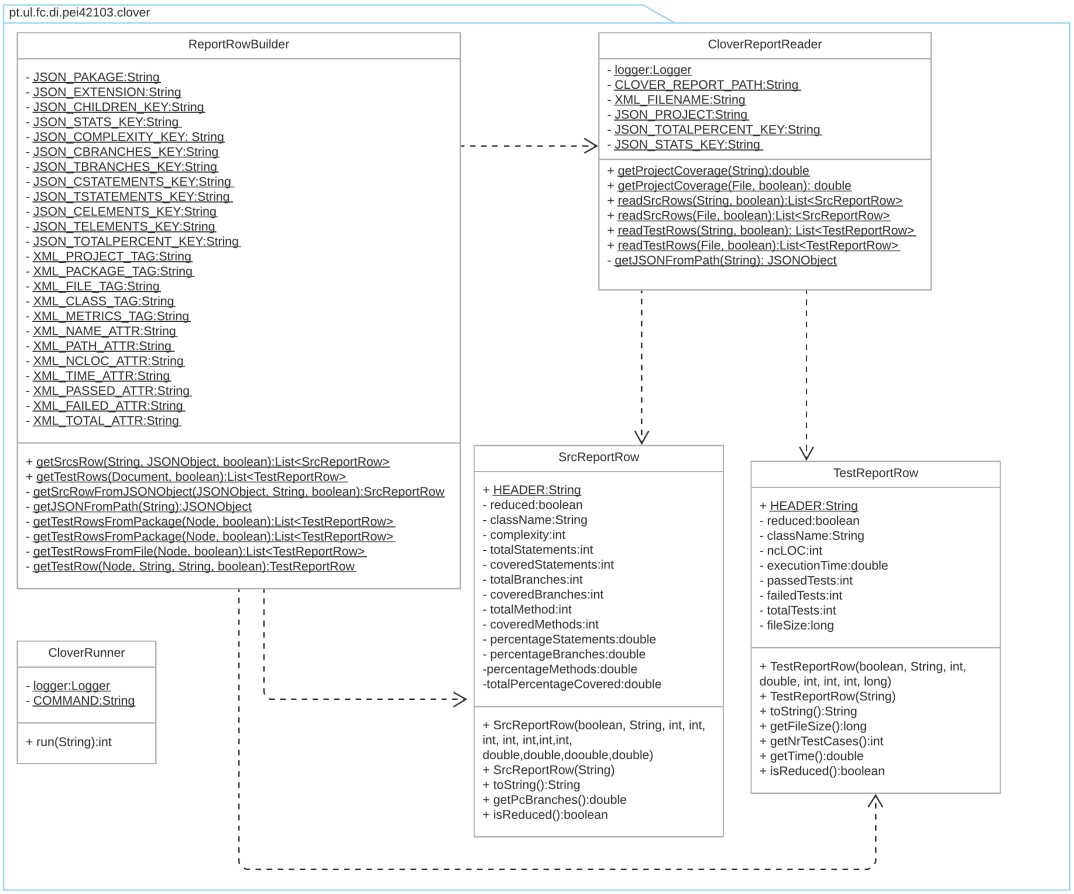

Figure B.2: Package pt.ul.di.fc.pei42103.clover class diagram.

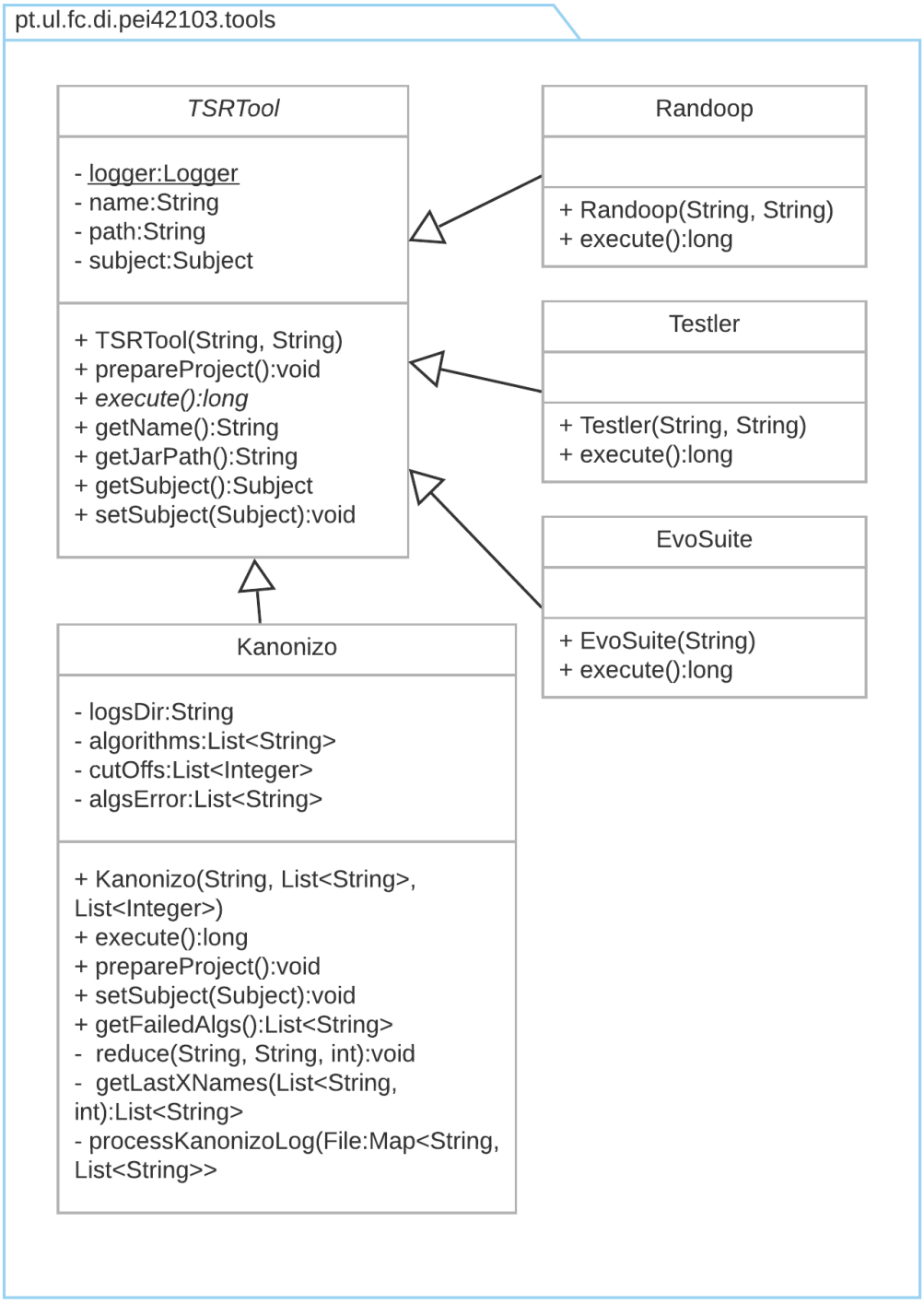

Figure B.3: Package pt.ul.di.fc.pei42103.tools class diagram.

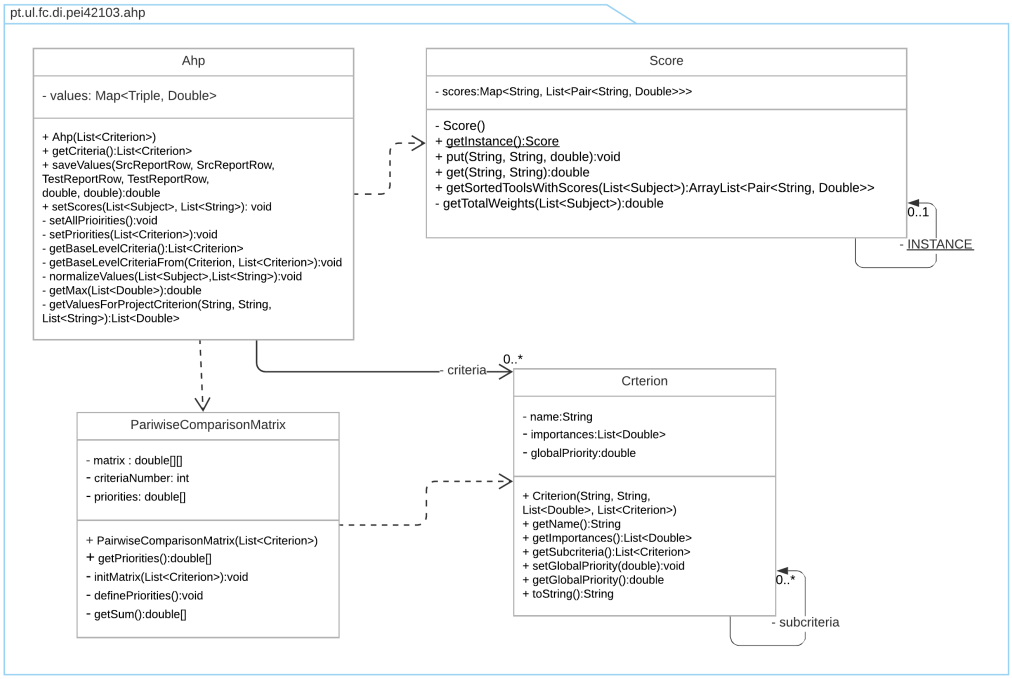

Figure B.4: Package pt.ul.di.fc.pei42103.ahp class diagram.

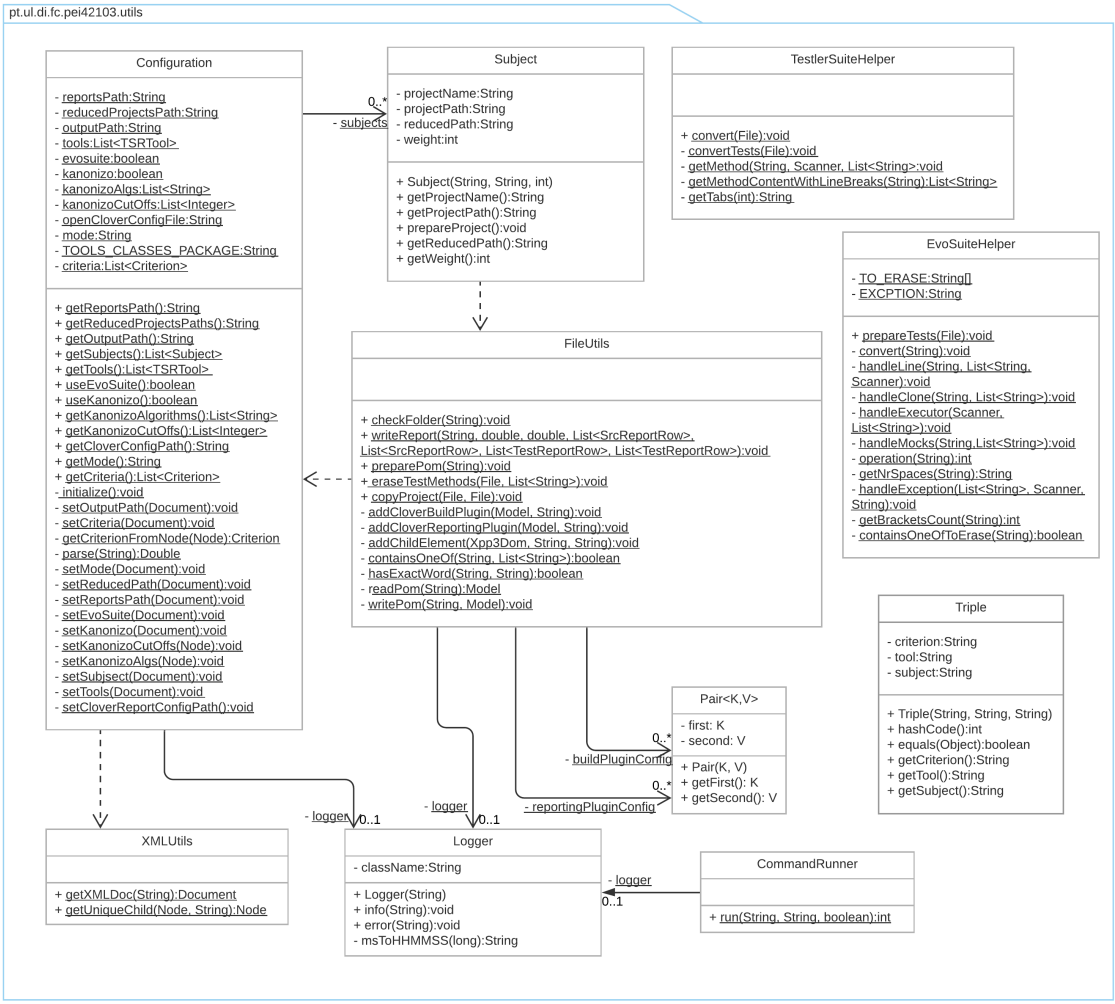

Figure B.5: Package pt.ul.di.fc.pei42103.utils class diagram.

Appendix C

Results

	commons-email	commons-lang	crunch-core	crunch-kafka	java-library	lambdaj	jfreechart	pmd-core	tika-xmp	tudu-lists	xmlsec	Total
Weights	1	1	1	1	1	1	1	1	1	1	1	
kanonizo#randomsearch#30	0.77603	0.89226	0.44499	0.90908	0.81879	0.90985	0.79976	0.93381	0.81474	0.95290	0.92666	0.83444
kanonizo#randomsearch#25	0.77902	0.90606	0.46529	0.90961	0.79417	0.89968	0.60661	0.92067	0.90740	0.93194	0.93679	0.82338
EvoSuite	0.99645	0.79803	0.34112	0.57145	0.97995	1.00000	0.93222	0.69066	0.87051	0.90696	0.95761	0.82227
kanonizo#additionalgreedy#20	0.46740	0.88963	0.97222	0.93806	0.81073	0.89379	0.57739	0.92895	0.72208	0.89315	0.86818	0.81469
kanonizo#additionalgreedy#30	0.51277	0.97088	0.94745	0.81430	0.82063	0.91027	0.43568	0.95390	0.81474	0.88996	0.81201	0.80751
kanonizo#additionalgreedy#15	0.48187	0.89311	0.92807	0.67808	0.73559	0.88867	0.65881	0.91892	0.90740	0.92604	0.82373	0.80366
kanonizo#greedy#30	0.50720	0.98494	0.47113	0.97855	0.79789	0.90055	0.60565	0.94308	0.72208	0.99401	0.73994	0.78591
kanonizo#random#30	0.49980	0.83796	0.42394	0.74282	0.78521	0.93987	0.71916	0.98021	0.72208	0.95328	0.88579	0.77183
kanonizo#greedy#25	0.50194	0.96298	0.46637	0.78798	0.81108	0.88745	0.62109	0.83004	0.81474	0.90902	0.87532	0.76982
kanonizo#random#15	0.47100	0.79078	0.35195	0.79155	0.77621	0.93316	0.57435	0.89387	0.81474	0.89830	0.88250	0.74349
kanonizo#additionalgreedy#10	0.46732	0.77139	0.92732	0.88160	0.48661	0.87572	0.51070	0.65375	0.81474	0.91910	0.86424	0.74295
kanonizo#greedy#20	0.49810	0.95149	0.40594	0.88791	0.78818	0.88168	0.65600	0.81616	0.90740	0.94467	0.30495	0.73113
kanonizo#random#25	0.49744	0.76971	0.48363	0.84763	0.81936	0.93102	0.58954	0.96540	0.90740	0.33866	0.81646	0.72421
kanonizo#randomsearch#20	0.48835	0.88847	0.43931	0.85898	0.32636	0.90550	0.54311	0.91288	0.72208	0.94705	0.91989	0.72291
kanonizo#random#20	0.47828	0.80140	0.45019	0.83730	0.77496	0.93926	0.30908	0.94229	0.62942	0.92274	0.84899	0.72126
kanonizo#additionalgreedy#25	0.47054	0.95821	0.94379	0.83828	0.50875	0.32006	0.47509	0.94095	0.62942	0.95930	0.72738	0.70652
kanonizo#greedy#10	0.46692	0.75151	0.32540	0.88964	0.30958	0.88644	0.53992	0.88693	0.90740	0.90194	0.84873	0.70131
kanonizo#randomsearch#15	0.44899	0.87576	0.40357	0.80759	0.63525	0.88194	0.56966	0.89508	0.44410	0.93286	0.79104	0.69871
kanonizo#random#10	0.39615	0.73548	0.41658	0.82353	0.60746	0.87204	0.33694	0.75542	0.90740	0.81598	0.92512	0.69019
kanonizo#randomsearch#10	0.39131	0.76880	0.40169	0.80839	0.47085	0.88278	0.58583	0.89070	0.53676	0.92590	0.82906	0.68110
kanonizo#greedy#15	0.41386	0.90686	0.42085	0.90036	0.76695	0.83741	0.66604	0.28839	0.72208	0.87448	0.56135	0.66897
Testler	0.41290	0.83816	0.37256	0.29464	0.71884	0.72157	0.46661	0.86981	0.25878	0.80273	0.92958	0.60783
Randoop	0.28130	0.30561	0.37256	0.58420	0.61543	0.83804	0.32877	0.79294	0.44313	0.87373	0.91918	0.57772

Table C.1: Example of the results table.

	Coverage	**Time**	**Dimension**
Coverage	1	1/3	7
Time	3	1	9
Dimension	1/7	1/9	1

Table C.2: Criteria Pairwise Comparison Matrix (real-world scenario).

	File size	**# Test cases**
File size	1	1/5
# Test cases	5	1

	% Branch coverage	**% Total coverage**
% Branch coverage	1	1/7
% Total coverage	7	1

Table C.3: Sub-criteria Pairwise Comparison Matrices (real-world scenario).

	Evaluated criteria	**Other criteria**	**Other criteria**
Evaluated criteria	1	9	9
Evaluated criteria	1/9	1	1
Evaluated criteria	1/9	1	1

Table C.4: Criteria Pairwise Comparison Matrices (focused sub-criteria).

	Evaluated sub-criteria	**Other sub-criteria**
Evaluated sub-criteria	1	9
Other sub-criteria	9	1

	Other sub-criteria	**Other sub-criteria**
Other sub-criteria	1	1
Other sub-criteria	1	1

Table C.5: Sub-criteria Pairwise Comparison Matrices (focused sub-criteria).